INSTANT SYNCHRONIZATION OF STATES IN WEB HYPERTEXT APPLICATIONS

vorgelegt von

Diplom-Informatiker

David Linner

aus Berlin

Von der Fakultät für Elektrotechnik und Informatik

der Technischen Universität Berlin

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften

Dr. Ing.

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. Odej Kao

Berichter: Prof. Dr. Dr. h.c. Radu Popescu-Zeletin

Berichter: Prof. Dr. Axel Küpper

Berichter: Prof. Dr. Lutz Prechelt

Tag der wissenschaftlichen Aussprache: 27.01.2012

Berlin, 2012

D 83

Abstract

The World Wide Web (web) has become one of the most relevant platforms for distributed, networked applications. This includes applications that support direct interactions among users. This thesis provides an answer to the technological challenges arising from the realization of real-time interactions. The real-time interaction is an approximation of the interaction wherein the moment of a user action and the moment the corresponding effect can be perceived are equal. Considered is the spectrum between interaction as means to achieve a common objective and interaction as means to achieve individual objectives. For this purpose i) ways to minimize the time-span between user action and play-out of the corresponding effect are investigated; ii) the impact of concurrent user actions is addressed with regard to user intentions, especially where user actions refer to application objects (e.g. a document or an auction). The findings are summarized to a software framework. The software framework shall help to significantly reduce human resources in the engineering process of interactive applications.

The international technology standards around the Hypertext Markup Language are introduced as background of this thesis. They define the technological fundament for the solution to be found. Starting from the resource-centered architectural style of the Web, state-of-the-art methods for accessing the state of a shared data resource are described. The methods are discussed with regard to their suitability for the realization of collaborative and competitive real-time interactions. A combination of *Operational Transformation* and *Bucket Synchronization* method are identified as appropriate.

The concept presented here brings both methods together. A communication schema based on operations and an algorithm for the resolution of conflicts between concurrent operations with respect to user intentions is created. Additionally, a general-purpose model for the representation of states for application object is proposed. As base for a validation, the concept is completely implemented. The validation comprises evaluation and benchmark tests as well as a case study. The evaluation test confirms the correct realization of the concept in the implementation. The benchmark tests reveal shortcomings of implementation with regard to scalability. The case study confirms that the thesis objective, a less resource intensive software engineering process, is achieved.

Zusammenfassung

Das World Wide Web (kurz Web) hat sich als Plattform für verteilte, vernetze Anwendungen etabliert. So auch für Anwendungen, die die direkte Interaktion von Anwendern miteinander unterstützen. Diese Arbeit beschäftigt sich mit den technologischen Herausforderungen, die aus der Umsetzung von Echtzeitinteraktionen im Web erwachsen. Echtzeitinteraktionen sind eine Annäherung an Interaktionen, in denen der Moment einer Benutzeraktion und der Moment der Wahrnehmbarkeit des Effekts durch andere Anwender übereinstimmen. Betrachtet wird das Spannungsfeld zwischen Interaktionen die der Erlangung eines gemeinsamen Ziels und Interaktionen die der Erlangung individueller Ziele dienen. So wird zum einen untersucht wie die Zeit zwischen Benutzeraktion und Wiedergabe des dazugehörigen Effekts so kurz wie möglich gestaltet werden kann. Zum anderen werden die Auswirkungen der Gleichzeitigkeit von Aktionen verschiedener Benutzer mit Blick auf deren Absichten behandelt, insbesondere für den Fall, dass sich die Aktionen auf ein Anwendungsobjekt (z.B. ein Dokument oder eine Auktion) beziehen. Es entsteht ein Softwarerahmenwerk, das die Realisierung von interaktiven Anwendungen mit Echtzeitanspruch beschleunigt und Entwicklungs- und Programmieraufwände somit deutlich verkürzt.

Als Hintergrund der Arbeit werden die internationalen Technologiestandards um die Hypertext Markup Language und gängige Praktiken für deren Einsatz vorgestellt. So wird zunächst der technologische Rahmen definiert in dem die Lösung zu suchen ist. Ausgehend von dem auf Ressourcen ausgelegten Architekturstil des Webs werden dann bekannte Methoden aufgezeigt um auf den Zustand einer gemeinsam genutzten Ressource zuzugreifen. Interaktionen werden mittels wechselseitigem Lesen und Schreiben auf Ressourcen abgebildet. Entsprechend werden die aufgezeigten Methoden mit Blick auf ihre Tauglichkeit für die Realisierung von kooperativen und konkurrierenden Echtzeitinteraktionen diskutiert. Als probate Lösung, die sowohl Unmittelbarkeit als auch Fairness gewährt, wird eine Kombination aus *Operational Transformation* und *Bucket Synchronization* Verfahren identifiziert.

Im Konzept werden beide Verfahren zusammengeführt. Es entstehen ein Kommunikationsschema für Benutzeraktionen auf Basis von Operation und ein Algorithmus für die Auflösung von Konflikten zwischen gleichzeitigen Operationen unter Berücksichtigung der Benutzerabsichten. Darauf aufbauend wird ein Model für die Darstellung von Zuständen für Anwendungsobjekte eingeführt. Als Basis für die Evaluierung wird das Konzept vollständig implementiert. Die Implementierung bildet die Grundlage für Korrektheit- und Geschwindigkeitstests. Außerdem wird damit eine Fallstudie durchgeführt. Die ersten Tests bestätigen die korrekte Umsetzung des Konzepts in der Implementierung. Die Geschwindigkeitstests zeigen Schwächen in der Skalierbarkeit der Implementierung. Die Fallstudie bestätigt, dass die Ziele mit Blick auf den Softwareentwicklungsprozess erreicht wurden.

Acknowledgements

First of all, I would like to thank my supervisor Prof. Dr. Radu Popescu-Zeletin, who gave me the opportunity to work at both the Technische Universität Berlin (TUB) and the Fraunhofer Institute for Open Communication Systems (FOKUS) and thus let me experience an environment that gave me room for fundamental as well as applied research. I'm grateful for his continual openness, patience, and guidance during the past years.

I'm deeply thankful to all my colleagues at TUB and FOKUS, and those who worked with me on the Real-time Web project and the Communication Cockpit project over the last years. Their comments and criticisms during many fruitful discussions were of tremendous importance and help.

I would like to thank all the students who have graduated in the field of real-time interactive applications on the web. Special thanks go out to Rafael Grote, Niko Pfeifer, and Stanley Schragl for their veritably outstanding work.

I would not have been able to complete my work without the continuous support, encouragement, and warmth of my family, Anne and Valerie. I'm grateful for their ability to cheer me up and to refill my strength by simply being there.

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Definitions

List of Interfaces

1 Introduction

The introduction starts with the objectives and scope, and justifies the approach chosen to achieve the objectives. Finally the structure of this thesis is outlined.

1.1 Motivation and Objective

More than 1.3 billion computing devices worldwide, from smart phone to desktop computer, support the playback of rich, digital content encoded with the family of industry standards around the Hypertext Markup Language (HTML) [1]. More than 1 trillion unique offers of such encoded content are counted on Internet hosts [2]; they form the World Wide Web. More than 70% of the offers on the web are applications in the sense that user inputs define the presented content or the kind of presentation [3]. The web is thus one of the most relevant platforms for networked and distributed applications today.

The rise of the social web (formerly known as web 2.0 [4]) demonstrates the maturity of the web as platform for distributed interactive applications, i.e. applications that enable users "to act on or in close relation with each other" [5]. The objectives of application users define whether an interaction is competitive or collaborative. If two or more users pursue individual objectives, for example in an auction, the interaction is competitive. If all users pursue a common objective, as for instance in a wiki, the interaction is collaborative. Most interactive applications base on examples in the real world. Challenging, however, is the reproduction of the natural timing of interactions, especially when all interacting users should perceive effects of actions at the moment they originate [6], i.e. a reaction could happen instantly. The approximation of this ideal is referred to as *real-time interaction* in the following chapters. This definition is close to the traditional definition of soft real-time systems as e.g. provided in [7], while deadlines are defined by the limits of human cognition.

For the support of collaborative and competitive real-time interactions in a distributed application the following requirements are considered crucial in this thesis:

a) A user action has to take effect as soon as possible after it is issued.

b) A user has to be able to perceive an action, i.e. the resulting effect, as soon as possible after the action is issued.

c) An effect of an action has to be durable, i.e. it must not be rolled back.

d) For any two users the perceived effect of an action has to be the same.

e) For any two users the moment an effect becomes perceptible has to be the same. No user must be able to draw an advantage from an earlier perceptibility.

f) Any instant reactions of different users to the same action have to take effect at the same time.

g) Any instant user reactions to the same action that have mutually exclusive effects must be resolved equitably.

While the requirements a)-c) are relevant for both, competing and collaborative interactions, the requirements d)-g) provide a definition of the concept *fairness*, as utilized in this thesis.

The industry standards for the web imply one architectural style and one communication pattern for all web applications. The architectural styles require mapping all user actions in an application to read, write and execute operations on a centralized resource [8]. Accordingly, the communication pattern is limited to a request-response scheme terminated at the central resource. To design a web application that supports the direct interaction of two or more users, a mutual read-write scheme has to be applied to notify about user actions. As a result, engineers of an application that supports real-time interaction face two major challenges:

i) The mutual reading and writing has to be optimized with regard to the time interval between a user action and a corresponding effect perceptible by other users.- requirements a) and b)

ii) The shift between action and effect perception causes concurrent user actions. Concurrent user actions conflict if the expressed user intentions collide. If user actions address an application object, such as a document or a trade, conflicts should be resolved by the application with respect to the user objectives. In a collaborative context the intentions behind two or more user actions can be reconciled. In a competitive context the resolution of conflicts has to be fair at least. - requirements c) to g)

As requirements and corresponding challenges suggest, considering traditional properties for reliable resource access only, i.e. atomicity, consistency, isolation and durability (ACID), is not sufficient in the context of real-time interactive applications. Additionally the properties instantaneity and simultaneity in terms of [9] have to be respected. Collaboration tools, e.g. collaborative text editing, drawing or co-browsing applications, successfully address the problem of message concurrency when users interact while pursuing a common objective. Solutions for handling interactions where users pursue individual objectives are studied in the context of multi-player games [10-12]. However, such approaches have not yet reached the web in a variety comparable to collaboration tools.

The objective of this thesis is to create a software framework that provides a broad base for experimenting with innovative applications for collaborative and competitive interactions in close to real-time. The software framework is supposed to relieve software engineers of the burden of realizing solutions for action notification and conflict resolution. A strong fit with the architectural style of the web and the application model are intended to lower the expense of adaption. The software framework shall help to reduce the resources for software engineering and to minimize the risks in the realization of applications in an algorithmically and technologically challenging area.

1.2 Scope

The responsiveness of distributed interactive applications, i.e. the action-perception time interval introduced above, is influenced on network-level and application-level. Network delays are mainly caused by the propagation speed of signals on the links, the load on the links and the processing of packets at the routers. An optimization of network topology, capacity and equipment has a positive impact on the responsiveness of distributed interactive applications, but is beyond the scope of this thesis. Optimizations on application level aim at i) the communication algorithms, ii) the information management, or iii) at the system architecture [13]. The communication algorithms define the meaning of events and the patterns for the exchange of events between hosts. The optimization of algorithms aims at best satisfying the user interaction model implemented by an application, such as message passing, publish/subscribe or document sharing. The information management addresses the handling of data and the utilization of data semantics to optimize the responsiveness of applications. For this reason data compression is a means of information management as well as dead reckoning. Dead reckoning is utilized to predict events in a limited space of possibilities. The system architecture describes the components required to realize an application. Optimizations of the system architecture mainly focus on reducing the number of network hosts involved in user interaction and on shortening the physical distances between these hosts. The impact of the network characteristics is supposed to be reduced. This work addresses communication algorithms and system architecture for distributed interactive applications on the web. Information management techniques are not addressed beyond the state of the art.

Apart from numerous combinations like Publish-Subscribe Systems [14, 15], Data Spaces [16-18], Blackboards [19] and Remote Procedure Call [20], the two basic means for communication between two or more sites of a distributed application are the messages and shared resources. Other methods are either derivations or combinations. Both methods are suitable for the realization of interactive applications. Web applications base on mutually writing to and reading from shared resources [8]. The communication method is thus resource-centric. The software framework presented in this work is intended to respect the technological principles of the web and complement existing technologies. Message-centric approaches for the realization of user interactions are thus out of scope and are not considered in detail.

Interactions among users are generally classified as competing or collaborative. The web as application platform does not make a difference between these classes, but for real-time interactive applications they determine the appropriate family of algorithms. In order to retain the generality of the platform, the solution presented here addresses both classes of user interactions.

1.3 Methodology

The work conducted in the scope of this thesis comprises eight major steps documented in the following. Figure 1-1 show the dependencies between these steps.

1. *Study web as a platform:* Classify the role of the web as application platform, research the models for realizing applications on the web and methods to enable the communication between multiple sites of a web application with low-latency.
2. *Study instant state synchronization techniques:* Understand and qualify the range of algorithmic solutions for handling shared resources in time critical environments. Identify appropriate algorithms for mapping collaborative and competitive user interactions to operations on shared resources.
3. *Research instant state synchronization solutions for the web:* Identify and analyze current approaches to realize real-time interaction in web applications, software solutions as well as theoretical work. Clarify how this thesis distinguishes from related work.
4. *Algorithmic and conceptional framework:* Design a communication scheme that respects the algorithmic requirements as well as the characteristics of the application platform web. Define state access mechanisms, constraints and conflict resolution methods for a generic resource state model.
5. *All-purpose resource state model for web applications:* Define the basic data structure for a state model applicable in web applications and define primitives to modify this data structure. Specify a configuration to adapt state model and primitives to the conflict resolution methods of the algorithmic and conceptional framework.
6. *Abstract software system specification:* Define how algorithmic and conceptional framework can be realized in a software system independent from the technological solutions applied for this realization. Specify a system architecture with components and component interfaces.
7. *Implementation:* Develop a fine-grained software specification, select technologies for the realization and implement the specification. Evaluate the implementation to verify that the algorithmic and conceptional framework is implemented correctly. Determine the performance of the implementation.
8. *Empirical Evaluation:* Research if the created software system and the underlying algorithms and concepts actually help to realize a real-time interactive application. Determine if the software engineering process is positively influenced by concepts and software. Provide feedback for a revision of theoretical and practical work.

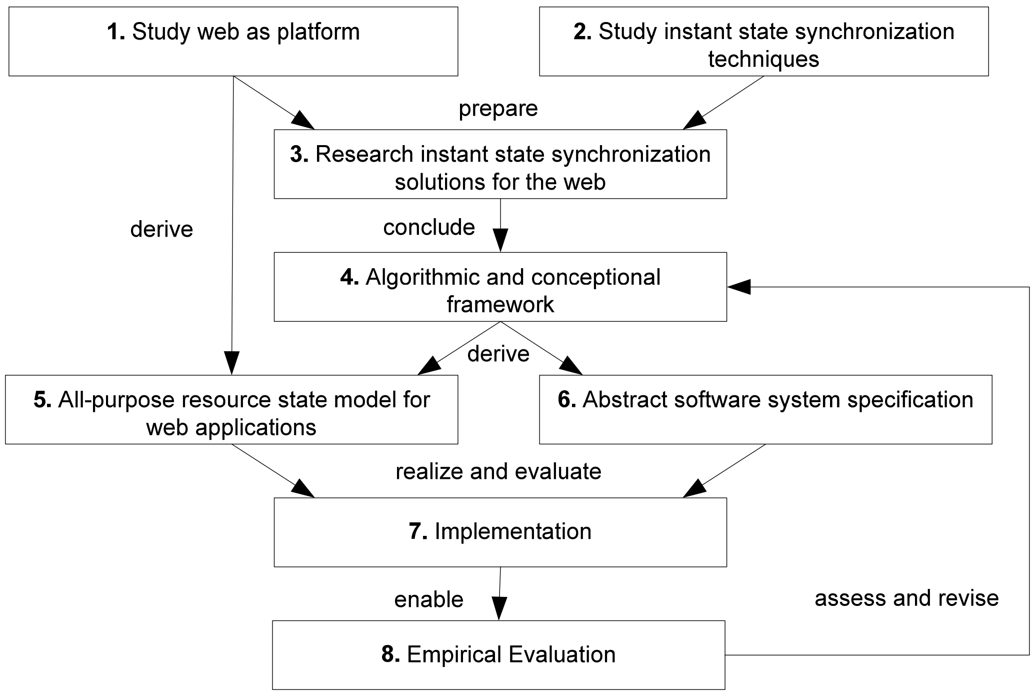

1. Study web as platform
2. Study instant state synchronization techniques
prepare
3. Research instant state synchronization solutions for the web
derive
conclude
4. Algorithmic and conceptional framework
derive
5. All-purpose resource state model for web applications
6. Abstract software system specification
realize and evaluate
7. Implementation
enable
assess and revise
8. Empirical Evaluation

Figure 1-1: Methodology of this thesis

1.4 Contribution

The contribution of this thesis to the current state of research and engineering is threefold:

First, a synchronous method for accessing shared resources on the web is specified. The method is inspired by bucket synchronization and operation pipelining approaches. No comparable approach has been presented for the architectural and technological landscape of the web by now. The method enables fair interactions of users, while fair refers to the timing of data presentation and action processing. In addition to the theoretical work, a binary network protocol and a text-based network protocol are specified for this method. Both protocols are designed to complement current and upcoming technology standards for the web.

Second, a novel client-server operation transformation procedure is introduced. In contrast to state of the art approaches, the procedure enables clients to continuously originate and receive operations without the need for sequenced access to the server. The control algorithm builds on the theory of proven and established algorithms and algorithmic properties. New is the bulk-processing approach for operation sequences that reduces the overall time required for state convergence and enables the concurrent client access to the server. Novel is also to determine the synchronization mode, i.e. pessimistic or optimistic, per operation based on a schema. The operation transformation method introduced in this thesis is not limited to text. Instead it is applicable to any state model that bases on a continuous namespace.

Third, a tree-based state model tailored to the needs of web engineers is specified. In contrast to comparable approaches adjacent nodes can be ordered and unordered in this model. This option enables a direct mapping between objects utilized in the modern application model of the web and the representation of a shared resource. In addition to the state model, operations for state modification and a corresponding operation transformation function are provided.

1.5 Outline

This thesis has eight chapters, a conclusion and an appendix. An overview of the contents is given in the following:

The Web as a Platform introduces the architecture and the application model for application on the World Wide Web. With regard to the instant synchronization of resource states, standards and techniques are introduced that serve as starting point for the work conducted in this thesis.

Resource-centric Interaction introduces state of the art principles for resource sharing in distributed systems. Read replication and full replication techniques are identified as relevant in the context of real-time interaction on the web and all algorithms that originated from that research area are explained in detail, as well as their advantages and shortcomings.

Related Work introduces related research, community activities and software projects. Advantages and disadvantages are explained and discussed with regard to the objectives of this work.

Instant Synchronization of a Shared State contains the fundamental concept of this thesis. An operation pipelining approach for the exchange of user events between internet hosts is described as communication scheme. Based on this communication scheme a generic operational transformation concept for state convergence is defined.

Instant State Synchronization System (IS3) introduces the specification of an abstract system architecture based on the communication scheme and the state convergence algorithm introduced in the previous chapter. The specifications address client and server components, as well as component interfaces.

Web Object State Model introduces a configuration for the generic IS3 that satisfies the needs of WHAs. In addition to a partially ordered tree as state model, functions are specified for the customization of the operational transformation algorithm.

The Pulsar System describes the whole reference implementation for IS3 as a middleware for web applications. The description includes specifications for the network interfaces, component implementations, the robot-based customization model and the application programmer interfaces.

Evaluation and Benchmark: The correct operation of communication scheme and stack implementation as part of the reference implementation is shown in an evaluation. A benchmark is documented that analyses the limits of the reference implementation in practical scenarios.

Case Study documents the result of an empirical evaluation. A case study is conducted to verify the achievement of the thesis objectives by applying Pulsar in the context of software development projects. The first case addresses the realization of a collaborative rich text editor. The second case describes the realization of an online adaption for the classic console game Bomberman.

Conclusion summarizes the achievements and discusses whether the objectives were met. Also this chapter includes a brief overview of future prospects.

2 The Web as a Platform

Although the Web started as a ubiquitous library for rich text content, it is one of the most relevant application platforms today. This chapter comprises descriptions of traditional and modern application models. Additionally, methods are presented for realizing low-latency bidirectional communication based on this application model.

2.1 Background

In the early 1990s Berners-Lee et al. started an internet application technology for the access to remote documents. In April 1993 the Conseil Européen pour la Recherche Nucléaire (CERN) released the technology specification and its reference implementation for a hypertext information system. The rapid distribution and the word wide emerging content for this technology soon formed a virtual network referred to as the World Wide Web [21]. In the remainder of this work the term *web* is used as short form for World Wide Web.

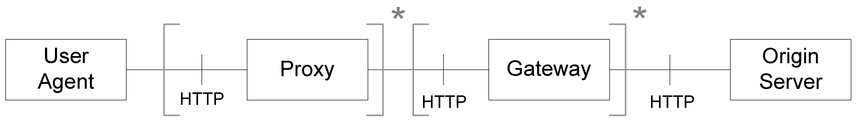


Figure 2-1: Web Architecture

The technology developed by Berners-Lee et al. bases on a client-server application pattern. Documents to be shared are stored on the *origin server*. A document viewer called *user agent* accesses these documents through the internet. Today the term *web browser* is the more common term for the document viewer. Throughout this work both terms are used. User agent and origin server communicate through the Hypertext Transfer Protocol (HTTP) [22]. HTTP realizes a request-response communication pattern. Solely the user agent is allowed to start a request. The origin server can only respond to requests. Request and response message of HTTP are text-based. The request contains at least the identifier of the document to be viewed. The response at least contains a status code, a field indicating the length of the content and the content itself. A document identifier is encoded as Uniform Resource Locator (URL) [23], a format which was later generalized as Uniform Resource Identifier (URI)[24]. The URL is not only an abstract name; rather it has to be resolvable to the physical address of an internet host. Berners-Lee et al. also developed a platform-independent transport format for document copies, the Hypertext Markup Language (HTML), which has now reached pre-recommendation status in version 5 [25]. Document copies in HTML representation are also called *web pages* throughout this work. The innovation of HTML is the hyperlink. With hyperlinks the HTML syntax allows referencing any other document available on an internet host implementing HTTP. The reference has to be given as URL. When the user selects a hyperlink from a document in the User Agent, the User Agent has to request the referenced document from its origin server and to show the contents of the response.

The basic architecture of the web is optionally extended with *proxies* and *gateways*, as depicted in Figure 2-1. The gateway is introduced to cache documents in their HTML representation on behalf of the origin server. Additionally, a gateway can be used to balance the request load between multiple origin servers serving the same content, or to enforce security constraints. The intention of the proxy is to cache documents on behalf of the user agent or the local network of the user agent (e.g. an enterprise network). Additionally, proxies are used to implement content filters (e.g. black list, white list).

Fielding et al. later generalize the notion 'document' as *resource* and introduce the term *representation* for document copies [8]. A representation is thus a context-dependent snapshot of a resource state. Moreover, they introduce with *Representational State Transfer (REST)* a formal model for methods to access resources through HTTP and add methods to write and delete resources. The remainder of this work uses the terminology of Fielding et al.

2.2 Application Models

2.2.1 Traditional Application Model

With the quick growing size of content available on the web, means are required to concretize requests and to individualize responses. Since the parameters for concretizing a request are content-specific, an inflexible extension of HTTP or the user agent is inappropriate. Instead HTML is extended with so called forms, which represent graphical input elements. A special submit button contained in each form terminates the user input. Once terminated, the user agent takes the names and values of form input fields, encodes them as list of name-value pairs according to [24], attaches this list to a prefix URL and starts a request to the just created URL. Forms are thus nothing else than hyperlinks with placeholders for user inputs.

As a counterpart to forms, the Common Gateway Interface (CGI) [26] is introduced at the origin server. CGI takes an HTTP request, in particular the URL, and translates it to the name of a program and a list of execution parameters for this program. The program is started while the program outputs are captured and forwarded as payload to the HTTP response. Figure 2-2 shows an example of a CGI program that simply adds two numbers. With the help of a form, the user provides two parameters *a* and *b*. The parameters are encoded and attached to the request URL. The origin server forwards the request to CGI. CGI parses the request and derives a program name and execution parameters in a customizable way. The program is started with the parameters and produces HTML encoded content as output. The content is tailored to the request parameters. Also the CGI program could output HTML that again contains a form. As the example suggests CGI programs may not only filter a number of static resources according to user parameters, but also generate data on demand. To create a coherent application, with a look and feel like desktop applications, a number of CGI programs can be chained to a continuation.

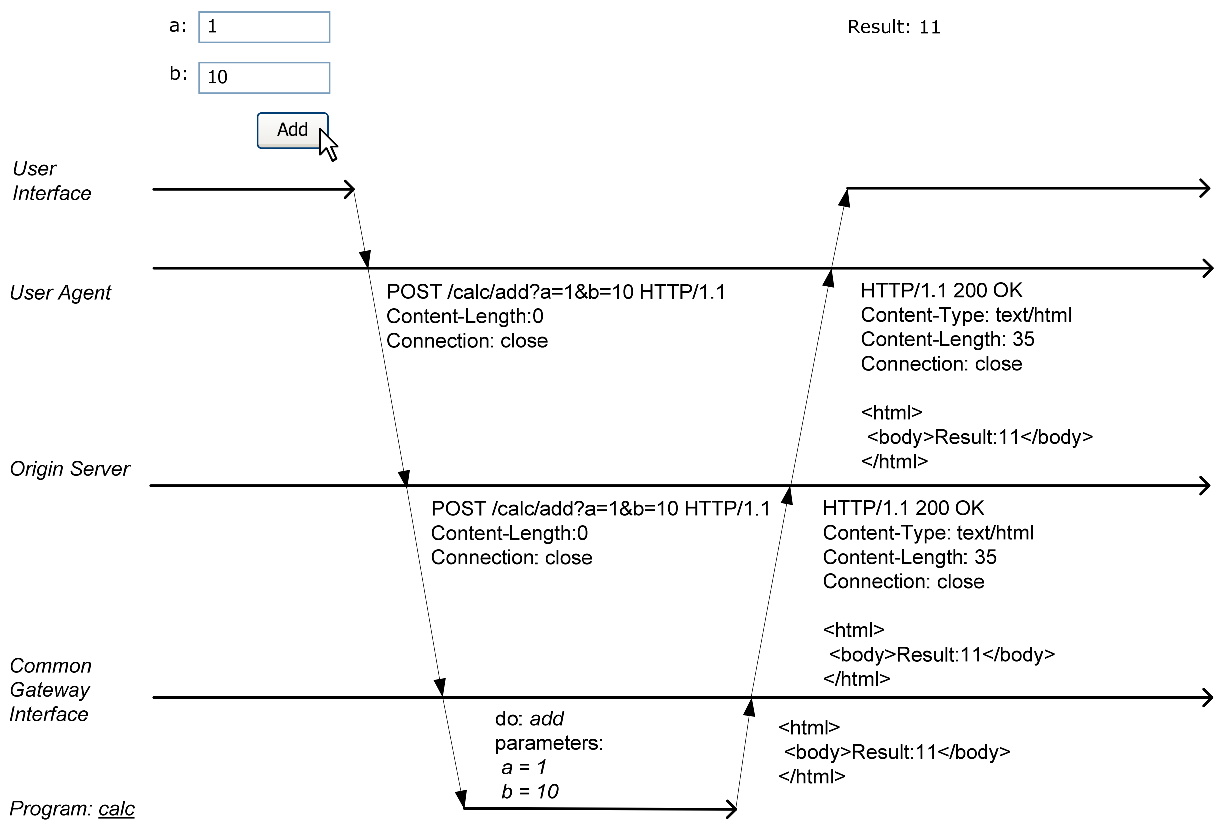


Figure 2-2: Example CGI application

HTTP is stateless. To work with the same data in a number of succeeding requests (e.g. of a continuation), these data have to be looped through from the last response to the next client. Alternatively the data can be stored at the server temporarily and a reference to these data – a so called session identifier - looped through instead. Although commonly applied, the latter solution is unacceptable with regard to the REST principles. The loop through of data however, is supported by the *cookie*, a special protocol feature of HTTP [27]. As alternative to cookies, request parameters are used to communicate the context of a request.

Modern alternatives for server-site programs do not necessarily build on CGI, but the principle of connecting origin server and programs for the dynamic creation of content is the same. Some representatives are PHP [28], Java Servlets and Server Pages (JSP) [29], Active Server Pages (ASP) [30], Ruby on Rails (RoR) [31] and the Google Widget Toolkit (GWT) [32]. A comprehensive survey of recent platforms and frameworks for server-site programs is presented in [33]. A peculiarity of modern CGI successors is the support for the modularization of server-site programs according to the Model-View-Controller (MVC) software engineering pattern. This basically postulates a clearly separation of the data to be presented (Model) from the presentation (View) and the application logic for selecting data and presentation dependent on user inputs (Controller). Views are templates (mostly HTML pages with placeholders), which are only populated with data when the data are requested. Application frameworks like GWT go a step further and allow software engineers to design a complete web application with a GUI framework for desktop applications, while performing the required mapping to web technologies themselves.

In the remainder of the document the term *web server* is used denote the set of static and dynamic resources for one application on the origin server. The term *web client* is used to denote the set of representations for all resources in the web server. For example, a CGI program adding two numbers is part of the web server. The markup created by this program and shipped to the user agent is part of the web client.

2.2.2 Extensions to the Traditional Application Model

The traditional application model of the web supports the realization of applications which look similar to simple business applications for standalone computers. In the early days of the web, however, HTML was limited to the presentation of decorated text, images and tables. Graphic elements (e.g. charts, directory trees), dynamic elements (e.g. video) or interactive elements (e.g. games) were not supported. For that reason the company Netscape enabled its user agent to understand HTML markup for an active object, also called *applet*. Applets are precompiled programs with an optional user interface. Comparable to images, they are served by the origin server and are referenced from a web page by URL. When the user agent renders an active object in a web page, it requests the program from the origin server, starts the program on the user host and embeds the user interface of this program with the web page.

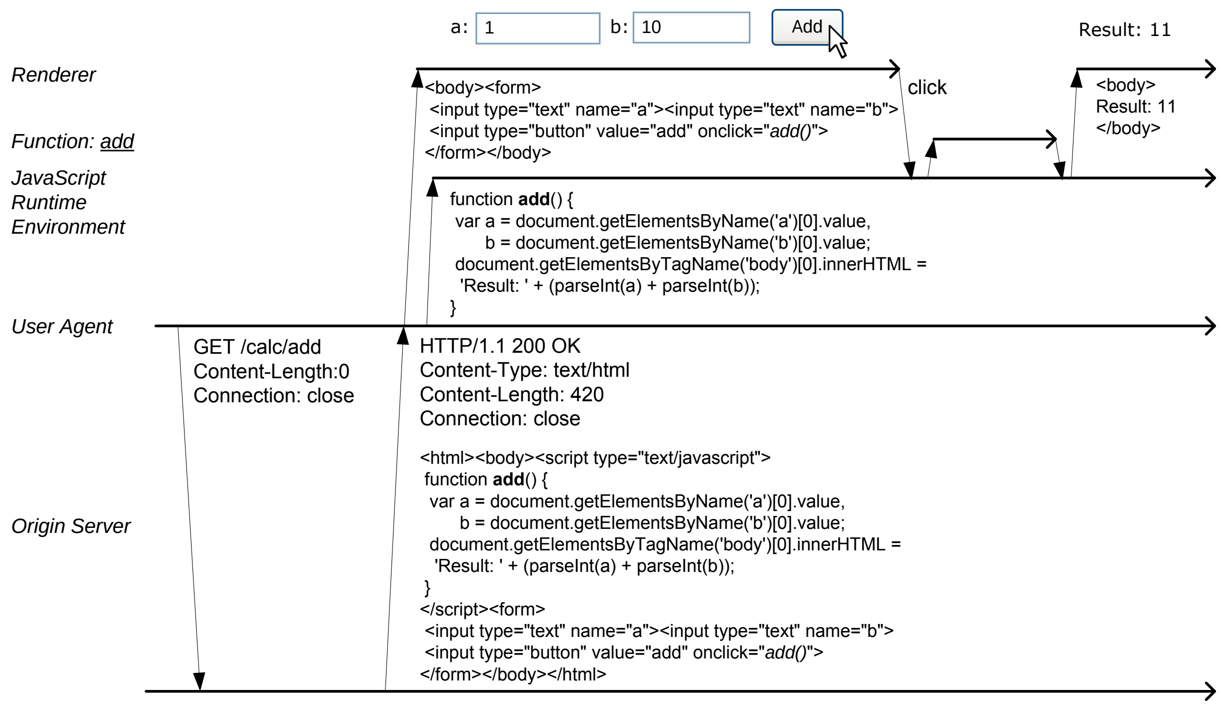
Obviously user agent or user host have to be able to execute the program. The Netscape user agent therefore requires the programs to be represented in Java byte code and the user host to run a Java Virtual Machine (JVM). Support for embedded active objects is found in most user agents today. Successors of the JVM as execution environment are for example Adobe Flash Player, Adobe Shockwave Player or Microsoft Silverlight runtime environment. The runtime environments differ not only in the syntax of code they execute, but also in the capabilities they offer. Flash and Shockwave are tailored to rich, interactive, graphical presentations including audio and video. Silverlight aims at making the user agent a fully fledged platform for any kind of application which was formerly found only on standalone computers.

Active objects pose a number of security risks for applications on the web. For example, the code obtained from a web server can be used to access file system, peripherals and network interface of the user host during execution. Another critical point of active objects is their poor conformance to basic ideas of the web. For example the code used to represent programs is encrypted or the parser code protected by copyrights and patents, which constrains the openness of contents in active objects and thus the searchability.

2.2.3 Modern Application Model

The introduction of applets reveals a weak point of web pages and the traditional application model of the web. Interactivity is bought with latency. Each user action that means a progress in the continuation inherent to a web application requires the user agent to start a request and the user to wait for the response. The answer to this weak point is to perform multiple succeeding steps of the continuation at the client, i.e. to strengthen the role of the user agent in the application model [34].

Originally intended to create a connection between applets and the surrounding web page, Netscape introduced a script programming language called *JavaScript (JS)*, which is today standardized as ECMA Script [35]. JS code is contained as source in a special markup element of an HTML page or referenced as source by URL. The user agent interprets the code when it renders the web page. Modern user agents also compile the code just-in-time and execute it natively. In this work, the term JS runtime environment denotes the component of a user agent that either interprets or compiles and natively executes JS code.



```
<body><form>
 <input type="text" name="a"><input type="text" name="b">
 <input type="button" value="add" onclick="add()">
</form></body>
```

```
<body>
Result: 11
</body>
```

```
function add() {
 var a = document.getElementsByName('a')[0].value,
     b = document.getElementsByName('b')[0].value;
 document.getElementsByTagName('body')[0].innerHTML =
  'Result: ' + (parseInt(a) + parseInt(b));
}
```

```
GET /calc/add
Content-Length:0
Connection: close
```

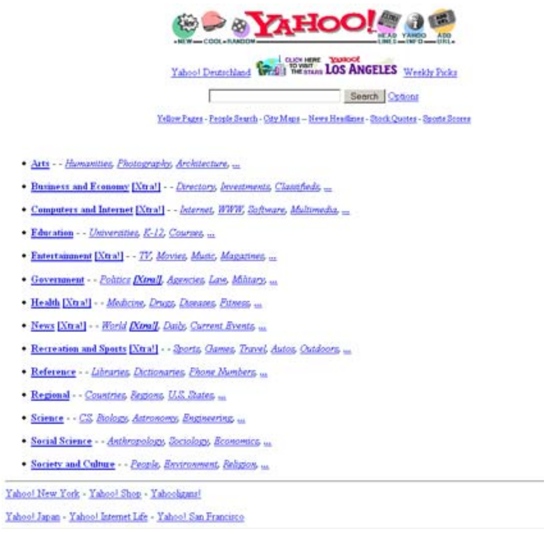
```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 420
Connection: close

<html><body><script type="text/javascript">
 function add() {
  var a = document.getElementsByName('a')[0].value,
      b = document.getElementsByName('b')[0].value;
  document.getElementsByTagName('body')[0].innerHTML =
   'Result: ' + (parseInt(a) + parseInt(b));
 }
</script><form>
 <input type="text" name="a"><input type="text" name="b">
 <input type="button" value="add" onclick="add()">
</form></body></html>
```

Figure 2-3: Example JS application

The JS runtime environment exposes a number of objects to the JS code as global variables. One of them is an object named *document*. The document object is the root element of the so called *Document Object Model (DOM)*, today standardized in version 3 [36]. The DOM is a leaf tree representation of the HTML page that contains the JS code. In this representation each markup element is a node, while nested markup elements are child nodes. Starting from the document object, the JS code can access and change each part of the HTML page by object methods and attributes. Changes, e.g. added nodes or modified text contents, are rendered and presented to the user immediately in the user agent. Code execution is triggered by user events [37] such as key presses and mouse actions on DOM nodes, or timer events. All JS events are collected in the same queue. The queue and thus the event callbacks are processed sequentially. All JS function calls are synchronous. Accordingly, no two functions access the same resource concurrently. The JS language does not support parallel computing at all. Pseudo-concurrent processing has to be emulated with continuations and timer events. The tandem of DOM event observation and DOM manipulation is also referred to as *DOM Scripting*.

The simplified example in Figure 2-3 shows the same use case as Figure 2-2, but is realized with DOM Scripting. An initial request to the origin server is replied with an HTML page containing JS code in addition to the form. The code includes a function which reads the values of the form input fields, adds them and writes the result to the DOM while replacing the form. The function is first triggered when the user raises a click event on the form button. When the event processing is finished, the DOM is rendered again and the result presented to the user.

DOM Scripting can thus be used to implement a number of successive use cases without request to the origin server. Practical examples are the validation of form fields, the collection of data in so called wizards, or skimming through large data collections. The exclusive handling of validation and usability use cases at the client reduces the number of requests an origin server has to process per second. DOM Scripting moreover improves the responsiveness of the user interface with regard to the user.



Yahoo! Deutschland · Weekly Picks

Yellow Pages - People Search - City Maps -- News Headlines - Stock Quotes - Sports Scores

- Arts - - Humanities, Photography, Architecture, ...
- Business and Economy [Xtra!] - - Directory, Investments, Classifieds, ...
- Computers and Internet [Xtra!] - - Internet, WWW, Software, Multimedia, ...
- Education - - Universities, K-12, Courses, ...
- Entertainment [Xtra!] - - TV, Movies, Music, Magazines, ...
- Government - - Politics [Xtra!], Agencies, Law, Military, ...
- Health [Xtra!] - - Medicine, Drugs, Diseases, Fitness, ...
- News [Xtra!] - - World [Xtra!], Daily, Current Events, ...
- Recreation and Sports [Xtra!] - - Sports, Games, Travel, Autos, Outdoors, ...
- Reference - - Libraries, Dictionaries, Phone Numbers, ...
- Regional - - Countries, Regions, U.S. States, ...
- Science - - CS, Biology, Astronomy, Engineering, ...
- Social Science - - Anthropology, Sociology, Economics, ...
- Society and Culture - - People, Environment, Religion, ...

Yahoo! New York - Yahoo! Shop - Yahooligans!

Yahoo! Japan - Yahoo! Internet Life - Yahoo! San Francisco

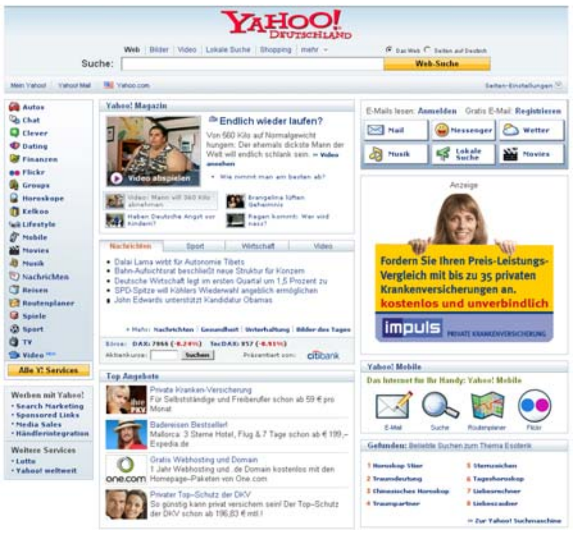


Figure 2-4: Yahoo user interface in 1996

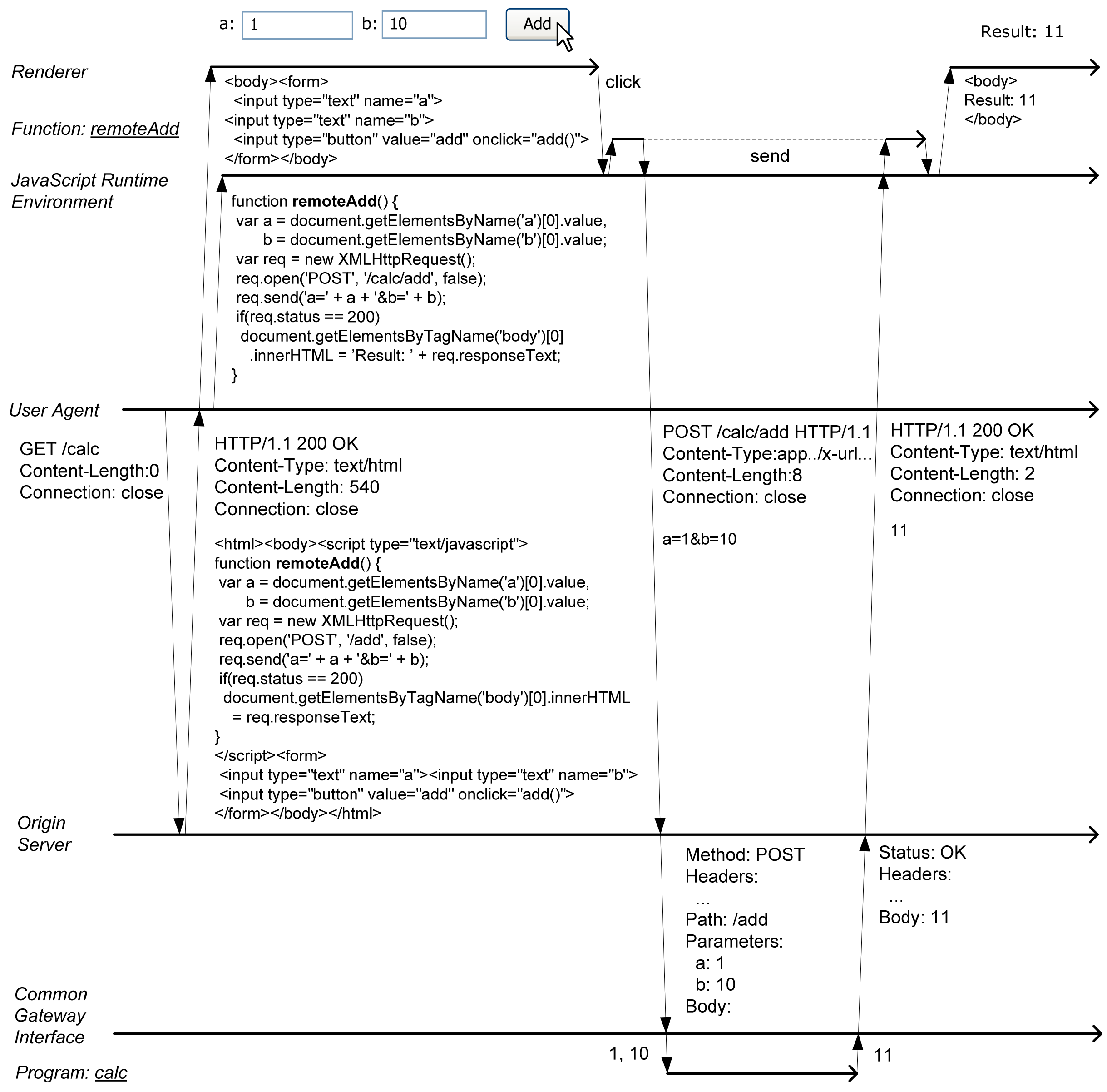
Figure 2-5: Yahoo user interface in 2008

In parallel to the development of the modern application model, HTML is complemented with the meta language *Cascading Style Sheets (CSS)*. CSS defines how the user agent has to render HTML elements to the screen (e.g. size, position, color, font, etc.). CSS and DOM Scripting encourage web designers and software engineers to create new, appealing user interfaces for application on the web. The rapid evolution of computing hardware ensures that user agents can still keep up with rendering and JS execution. Figure 2-4 shows the user interface of the web service Yahoo in 1996. Figure 2-5 shows the same user interface 12 years later. In 1996 the service bases on the traditional application model. The markup for the whole interface including images has a size of some 9,000 Bytes. The 2008 version of the interface has a size of 450,000 Bytes, including images, CSS and JS code. Although this version bases on the modern application model, requests to the origin server are still required, for example if the user wishes to follow a hyperlink. User agents cache contents like images and JS files referenced from a web page. In the worst case however, the presentation of a web page like the one in Figure 2-5 requires the user agent to request approximately ½ MB of data from the origin server. If the interface is broke up into multiple components, i.e. CSS files, JS files and images files, as many requests to the origin server are required. Even with a fast internet connection the user will experience the time for loading and rendering a web page as delay.

Modern user interfaces for application on the web are composed of multiple sections, (e.g. navigation, main content, search, quick links, advertisement, etc.). In most cases a user action results in the replacement of one section only. Consequently, there is no need to reload and render the whole page. This observation leads to the introduction and standardization of a new global JS prototype to the JS runtime environment, the so called *XMLHttpRequest (XHR)* [38]. An instance of this prototype is used for one request to the origin server of the requesting JS code. With XHR the application decides which events (e.g. caused by a user action) result in a new request to the origin server and also handles the data of the response.

The interface of the XHR prototype includes functions and attributes to send requests, monitor the request state and obtain response data. The application engineer has to decide whether to utilize XHR in synchronous or asynchronous mode. In the synchronous mode the function for sending the request blocks until the response is received or an error occurrs. In the asynchronous mode an application engineer has to provide a callback function, which is triggered when the response is received. Sending is non-blocking in asynchronous mode. A function that starts a request can terminate, the user interface remains responsive. The data of a response can have any text format. JS runtime environments natively support parsing and processing of HTML, XML and *JSON* format today. JSON, short for JavaScript Object Notation, is the minimal subset of the JS syntax required for the description of data structures excluding functions [39].

Figure 2-6 depicts a simplified example for an XHR application. The example again adds two numbers. The web client is initialized with a request to the calculator application. The HTML markup in the response contains a JS function. The function is called when the user clicks to the add button on the screen. First, the function reads the values of two form input fields. Then it creates and starts an XHR to its origin server in synchronous mode. The web server includes a CGI program that adds the two numbers. In this case the CGI program only returns the result of the operation, but no markup. When the response with the result reaches the user agent, the function processing is resumed. The function reads the result of the CGI program from the XHR instance, appends a label to it and adds the resulting text to the DOM. When the function terminates the JS runtime environment tells the user agent to render the DOM again. The example is chosen for its simplicity. In practice, requests to the web server are only used to access resources unavailable at the user agent, such as databases or special services.



```
<body><form>
  <input type="text" name="a">
<input type="text" name="b">
  <input type="button" value="add" onclick="add()">
</form></body>
```

```
<body>
Result: 11
</body>
```

```
function remoteAdd() {
 var a = document.getElementsByName('a')[0].value,
       b = document.getElementsByName('b')[0].value;
 var req = new XMLHttpRequest();
 req.open('POST', '/calc/add', false);
 req.send('a=' + a + '&b=' + b);
 if(req.status == 200)
  document.getElementsByTagName('body')[0]
    .innerHTML = 'Result: ' + req.responseText;
}
```

```
GET /calc
Content-Length:0
Connection: close
```

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 540
Connection: close

<html><body><script type="text/javascript">
function remoteAdd() {
 var a = document.getElementsByName('a')[0].value,
       b = document.getElementsByName('b')[0].value;
 var req = new XMLHttpRequest();
 req.open('POST', '/add', false);
 req.send('a=' + a + '&b=' + b);
 if(req.status == 200)
  document.getElementsByTagName('body')[0].innerHTML
    = req.responseText;
}
</script><form>
 <input type="text" name="a"><input type="text" name="b">
 <input type="button" value="add" onclick="add()">
</form></body></html>
```

```
POST /calc/add HTTP/1.1
Content-Type:app../x-url...
Content-Length:8
Connection: close

a=1&b=10
```

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 2
Connection: close

11
```

```
Method: POST
Headers:
  ...
Path: /add
Parameters:
  a: 1
  b: 10
Body:
```

```
Status: OK
Headers:
  ...
Body: 11
```

Figure 2-6: Example XHR application

The example reveals the movement of MVC design to the web client. Controller, a JS function, and View, an HTML template, are shipped with the first response from the server. On demand of the Controller the web server provides the Model via XHR. In principle, one initial request to a web server is sufficient to initialize the web client for the entire user session, assumed the session is not affected by external events (e.g. disconnection). All successive requests can be handled by the web client itself through XHRs. This extreme of web application is also called *single page application* and marks the state of the art for web application engineering.

DOM Scripting, XHR and CSS do not fully exploit the possibilities offered by active objects such as applets. The *Web Hypertext Application Technology Working Group (WHATWG)*, the driving force behind HTML version 5, aims at extending HTML, CSS and the JS runtime environment to close this gap. The term Web Hypertext Application Technology is chosen to clearly distinguish the efforts of the working group from those around the integration of third party technologies with the web application model, such as Silverlight, Flash or JavaFX [40]. In the remainder of this work the WHATWG terminology is adapted to refer to web applications that exclusively utilize HTML, CSS and JS. Such an application is called *Web Hypertext Application (WHA)*.

2.3 Low-Latency Bidirectional Communication

The key requirement of resource-centric real-time interaction between two or more users is the notification of changes to resources and consequently their representations with low latency. WHA clients write to resources via XHR. When the client initiates an XHR, a TCP connection is opened instantly if needed, and an HTTP request sent. The WHA server can accept the write access when receiving the request and update the resource. The web application model, however, provides no standardized means to notify the update to other WHA client that requested a representation of the same resource before. Under the name *Comet* the engineering community has documented three techniques to work around this drawback, i) *frequent polling*, ii) *streaming*, iii) *long polling*. Based on these efforts, a WHATWG spin-off drives the standardization of a clean, long-term solution called *WebSocket* in the context of HTML5.

2.3.1 Frequent Polling

The frequent polling technique requires a WHA client to periodically initiate requests to the WHA server. If the server has an update notification for the client, this notification is piggybacked on the response. Ideally the polling interval is chosen short to prevent notifications from being delayed while waiting in an output queue at the WHA server. However, the more frequent the WHA client polls for notifications, the more likely are empty responses. As Figure 2-7 illustrates, updates to resources are requested at the server by separate XHRs. The figure uses trapezoids to represent communication phases. The left flank of each trapezoid symbolized the beginning of a request, the right flank the end of the corresponding response. Light-grey trapezoids depict communication on behalf of the WHA server, dark-grey trapezoids communication on behalf of the WHA client.

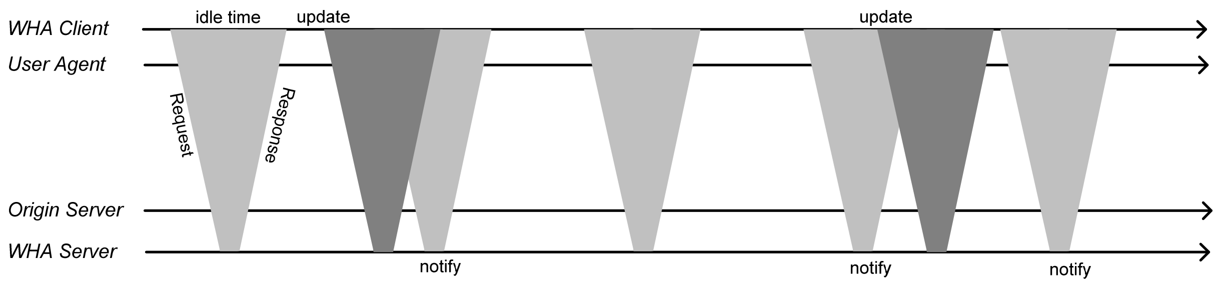


Figure 2-7: Comet technique frequent polling

Frequent polling is the most stable Comet technique, but also the least efficient with regard to resource consumption and latency. Each poll to an empty notification queue at the WHA server means overheads. Nonetheless, the polling interval has to be short in order not to delay notifications. If alternatives are not applicable, the polling technique can be improved by so called *guestimate patterns*, i.e. meta information about the probability of new notifications or the notification frequency.

2.3.2 Streaming

Streaming is the oldest and least reliable technique to deliver notification from WHA server to WHA client. The technique builds on capabilities of HTTP 1.1 to control the underlying TCP connection and transmit payload of initially undefined size. Additionally, a special layout element in HTML is used, the *iframe*. Like a hyperlink, an iframe references another resource by URL, but instead of causing the user agent to replace the currently shown representation on a user action, the iframe already contains this representation. Thus, iframes can be used to nest representations of different resources. If those representations are provided as HTML and originate from the same host, the corresponding DOMs are integrated in the DOM of the root page. To establish a stream the WHA client requests a prepared resource. This request is not handled with an XHR instance. It is caused indirectly by adding an iframe to the DOM that references the prepared resource. The WHA server responds with HTML, but omits the markup to indicate the end of the page. Moreover, it tells the user agent to keep the TCP connection alive and uses a chunked transfer coding on HTTP instead of transmitting the response payload as block. After receiving the beginning of the response, the user agent remains in the state of waiting for more data. Figure 2-8 depicts this characteristic with a continuous, shaded area.

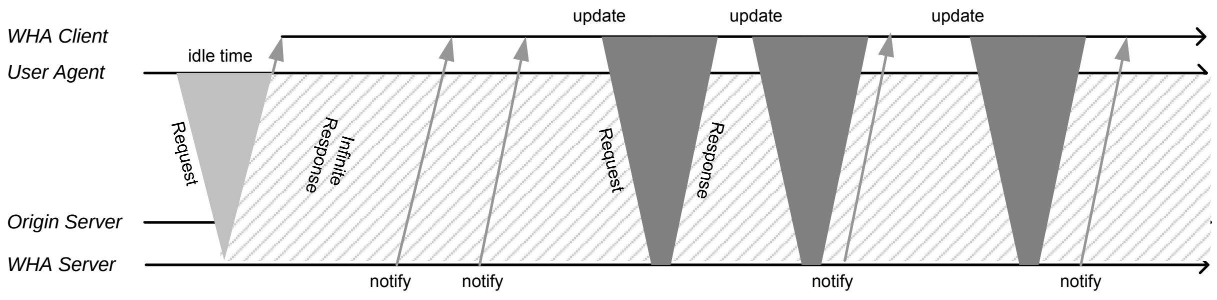


Figure 2-8: Comet technique streaming

When there is a new update notification for the WHA client, the WHA server sends another chunk of HTML via the pending connection to the client. The chunk of HTML contains a JS code segment with a call to a *receive* function, which was provided on initialization. The receive function works as callback function for the WHA client, while the parameters for its invocation represent the actual data of the update notification from the server. The user agent parses and interprets this chunk of HTML immediately and keeps the connection open. With every new server notification the procedure is repeated. To request updates the WHA client uses XHRs.

HTML was not designed with regard to the streaming technique. As a consequence, the technique does not work properly in every user agent and may cause irritating side effects. For example, some user agents do not process received markup or display a load indicator (hour glass as mouse pointer or progress bar) as long as a response is outstanding. Another problem is the monitoring of the connection. If the TCP connection carrying the streamed HTTP response is interrupted, the user agent renders an error message to the terminating iframe instead of raising an exception. The development of an error detection mechanism is thus complicated.

2.3.3 Long Polling

Long polling is an improvement of frequent polling. The frequent polling technique requires the WHA client to periodically request update notifications from a respective queue at the WHA server. The WHA server responds immediately. If the queue is empty, the response is empty. Like streaming, long polling utilizes the capability of HTTP to suspend a response. Still the WHA client has to provide the opportunity to deliver a notification by requesting it, but if the queue is empty the WHA server sends a partial response with chunked transfer coding and tells the WHA client to keep the underlying TCP connection open. The response is suspended until a new update notification is sent to the WHA client. In contrast to streaming, long polling closes the response when a notification is delivered. Immediately after the WHA client receives a notification, it has to start another request to the WHA server. Thus, the WHA server continuously has the opportunity to deliver notifications, as depicted in Figure 2-9. The requests of the WHA client to the notification queue as well as the update requests are handled with XHR instances. Like in frequent polling and streaming, update requests are handled separately.

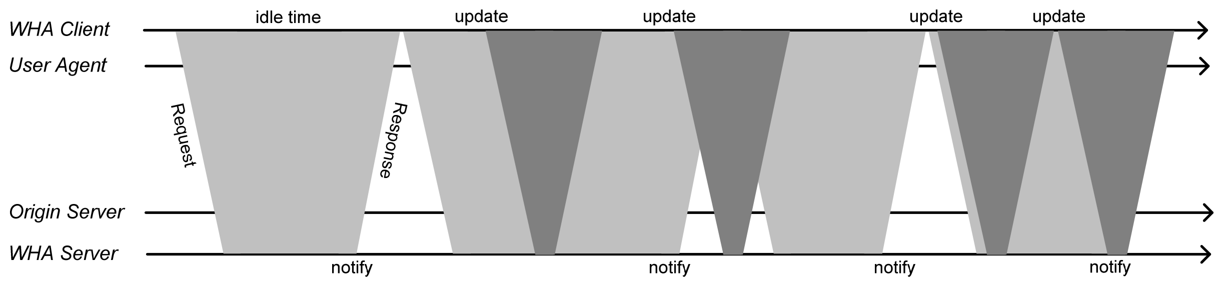


Figure 2-9: Comet technique long polling

To prevent timeouts on the underlying connection or achieve a better load balance at the origin server, the WHA server can close a suspended response after a predefined idle time and thus provoke the WHA client to start a new request. Long polling is more robust than streaming. If the TCP connection of a suspended response is lost, the corresponding XHR instance terminates with a proper exception message. The WHA client then has the chance to analyze the cause and recover the long polling scheme appropriately. Long polling can be seen as a trade-off between robustness (frequent polling) and efficiency (streaming). Since long polling bases on suspended responses like streaming, it also shares one of the drawbacks. Some user agents show a load indicator to the user as long as a response is outstanding, hence, permanently in the case of long polling.

2.3.4 WebSocket

In contrast to the Comet techniques, the WebSocket is not a work around for the existing WHA technology landscape. The name stands for a new API to be introduced for the JS runtime environment and a new application protocol for bidirectional messaging exchange. The new API realizes a JS prototype called *WebSocket*. An instance of this JS prototype basically provides a function to send messages and a callback handler for the subscription to incoming messages. In the current version of the recommendation draft [41] only text messages are supported, but the working group also discusses a support for binary messages. In contrast to XHR, outgoing and incoming messages are not related (e.g. by a request-response pattern). WHA client and server can send and receive messages at any time and without precondition once a connection is established. Only the WHA client can initiate the connection. While the discussions about the WebSocket API have reached the final phase, the standardization of a network protocol serving this API continues. The solution currently implemented by user agents supporting the WebSocket API bases on a draft of Hickson [42]. This solution, however, is not a general consensus. The *HyBi* working group of the Internet Engineering Task Force (IETF) has the remit to find a final solution. Their current draft is documented in [43].

3 Resource-centric Interaction

Interactions in a resource-centric distributed system base on mutual write and read operations to shared resources. The operations are based on knowledge about the state of these resources. This chapter introduces state-of-the-art-methods for the keeping synchronous the representations of a resource state at multiple sites of a distributed application. The concept consistency is refined with regard to timeliness and fairness of synchronization.

3.1 Resource Access Paradigms

A declared objective of this thesis is to present a solution that integrates with the resource-centric architectural style of the web. Accordingly, interactions of users have to be mapped to shared resources. The term resource can denote data, services or peripherals [44]. The term resource is always used to denote a data resource. The four basic access paradigms for shared resources in distributed systems are remote access, migration, read replication and full replications [45]. Figure 3-1 explains these with the help of finite state machines. Each state machine represents the user site. The state machines not only outline the local transitions, but also when and which messages are sent and received by the site.

The remote access paradigm is most common and also the foundation for REST. The accessing site does not maintain a full copy of the state of a shared resource, but rather a subset obtained on reading from the resource. Reading requires sending a message and waiting for a response with the portion of the state requested, as depicted in Figure 3-1a). If the site intends to write to the resource it also needs to send a message. A write operation is not completed until a message with a corresponding reply is received by the site.

Figure 3-1b) shows the migration paradigm. Migration reads and writes locally, i.e. without sending messages and waiting/blocking until a response arrives. The entire resource and the exclusive read and write access rights migrate from site to site on demand. Only a site which holds the resource can read from and write to it.

The remaining paradigms in Figure 3-1c) and d) base on replication. In the initialization, the site obtains a representation of the entire state resource. A read replication approach enables the accessing site to perform any read access locally without issuing messages. If the site has to write to the resource, a proceeding like in remote access is required. The site sends a message and waits for an acknowledgement of the operation. Additionally, the site may be requested to update the local copy of the state and acknowledge the update. A full replication approach supports local writing in addition to local reading. Synchronization of the state copy is done within a separate synchronization step which is decoupled from writing and reading, but involves sending and receiving of messages.

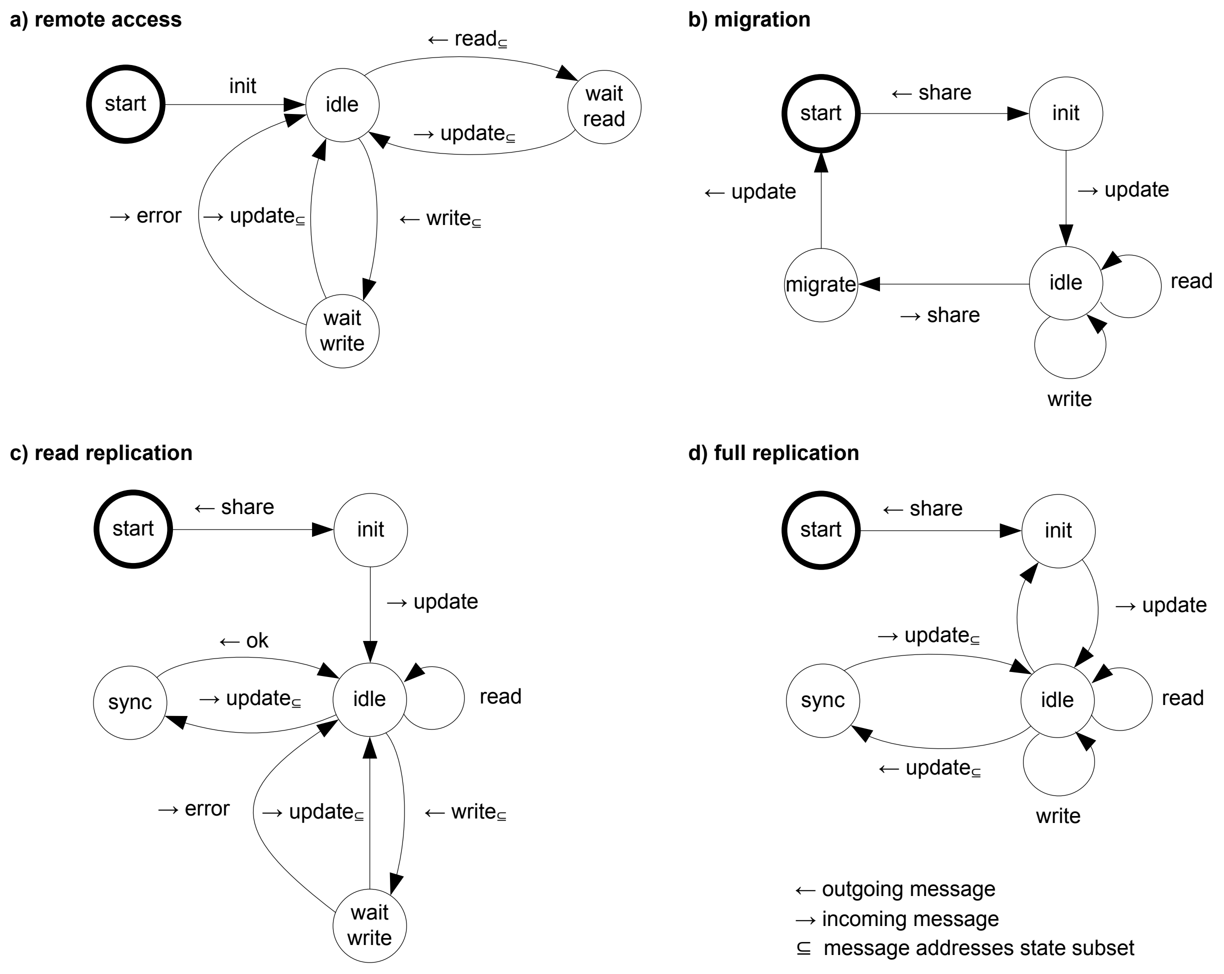


Figure 3-1: Basic access models for shared state

3.2 Consistency in Real-time Interactive Applications

The key objective the access paradigms is to keep the views to a shared resource consistent, i.e. unify the state representations even though they reside at different sites. In the context of interactive applications, this process is referred to as state convergence. The term consistency thus refers to convergent state representations of a resource and not the integrity of the resource itself. Remote access, read replication, and full replication allow multiple sites to access the same subset of a resource state concurrently, while concurrency and precedence are not necessarily related to physical time. They can also be defined with respect to the causal relations between accesses[46]. In the causal view of time, accesses are said to be concurrent if they are performed in the same context. If one access is performed on a state that already reflects another access, the first one is said to depend causally on the second one. Causal dependence is defined as the inverse of precedence in logical time. The order of accesses is essential for state convergence. If, for example, two sites share a state in accordance with the read replication paradigm and both receive the same two differential updates to their replica of the state, both have to perform the updates in the same order. Writing to a variable first the value 5 and afterwards the value 7 or vice versa results in different final values for this variable. The order among the accesses of multiple sites to the same resource can be maintained through continuous sequence numbers or timestamps taken from a global wall-clock time (e.g. realized with NTP [47]). Solutions explicitly addressing causal relation of accesses are logical clocks [48] and vector time. In any case the provision of an order index (timestamp, number, etc.) is required for each access. Logical time stamps have the advantage that they also serve as reliable reference to the current state of a site.

If two concurrent accesses are in conflict with each other they are said to be competing [49]. For a reliable handling of competing accesses they have to be atomic or applicable in isolation in terms of the ACID properties [50]. As described above, finally, a total order for all accesses is required throughout the distributed system. This can be realized with locks. Each site thus has to wait for all previous accesses of other sites to be committed before it can access the shared resource itself. The migration paradigm explicitly assumes locking. If concurrency is intended in the first place, e.g. for responsiveness reasons, competing accesses can be rejected when they occur. The issuing sites have to try again in that case. An alternative approach to rejection is inclusion. Inclusion is applicable if write accesses are not semantically exclusive. For example, two competing write accesses, both intending to increase the value of a variable by one, can be included to one write access increasing the variable value by two. Inclusion minimizes the number of rejections and rollback where required.

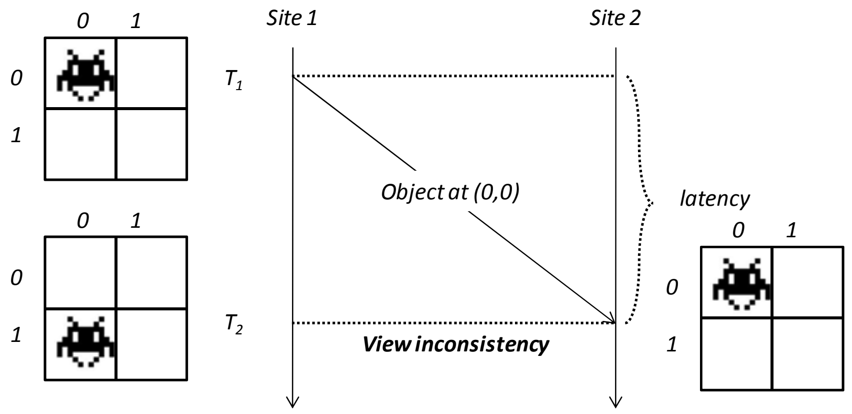


Figure 3-2: Consistency-latency tradeoff

Consistency here refers to the perceivable result of user interactions, the process of state convergence, however, requires to trade off simultaneity against instantaneity [9, 51]. Today, the basic means of communication for networked applications is the message. Interactive applications have to fill the gap between message passing on the network and the abstract interaction paradigm they provide to the user. The transport of a message between any two sites of an application is characterized by the underlying physical connection of these sites. The predominant factors affecting the transport time of a message are throughput and packet delay. While the delaying effect of throughput can be limited by reducing the size of messages, the packet delay depends on network congestion, hops in the routes realizing the connection and the physical limits of the transport medium. From approximately 1ms in a local network the average packet delay adds up to 150ms on a global connection. The reference value for delays in distributed interactive systems is 300ms [52]. Thus, on wide area network connections there is almost no tolerance if each transaction includes at least one message roundtrip.

Any access to a shared resource that requires waiting for a message has a potentially negative impact on the perceived instantaneity of responses to user actions, claimed by the requirements a) and b) in section 1.1. A solution for this drawback is to decouple access to the resource and message passing, as for example in the replication paradigms. This decoupling implies that the representation of the state maintained at a site can diverge from the versions of other sites. User decisions made with regard to an old and possibly incorrect state have to be corrected, in the worst case by a cancellation, i.e. a write access is rejected. This behavior conflicts with requirement c) in section 1.1. To minimize the number of cancellations and roll backs, the versions of the shared state have to be kept synchronous across the sites. The actual degree of divergence between the different representations of a resource state depends on the message latency. Figure 3-2 shows an example. The movement of a game character is rendered with delay to the screen on *Site$_2$*. Any reaction of the user on *Site$_2$* is thus inherently late. The situation is unfair. Alternatively *Site$_1$* could delay the rendering of the effect resulting from the local access for the time of message transport. The play out of the effect would be simultaneous for both sites, but simultaneity is bought with a loss of responsiveness for the *Site$_1$*. Consequently, a trade-off between the requirements a) - b) and e)-f) of section 1.1 is needed.

In summary, for the maintenance of a consistent shared state in real-time interactive applications the following properties are required:

- *Ordering and Causality Preservation:* preserve natural cause-effect order
- *Competing Access*: concurrent access of multiple sites to the same subset of the shared state
- *Simultaneity of access effect play-out*: preservation of temporal relations between accesses of different sites
- *Instantaneity of access effect play-out*: unperceivable delay between access and play-out of the respective effect

The remote access paradigm requires the sending of a message for each access and is thus unsuitable for distributed, interactive applications. The migration paradigm allows fast read and fast write access, but interactions of multiple sites require the mutual reservation and migration of the state. Concurrent access is thus impossible and the migration of the entire state may require the transport of large messages on each access. This violates the requirements a), b), e) and f) in section 1.1. Accordingly the migration paradigm is not applied in the practice of real-time interactive applications. The replication paradigms represent the most suitable compromise between responsiveness and interactivity, since access to the state and maintenance of consistency are partially decoupled. The read replication paradigm is suitable for state sharing based on continuous consistency and simultaneous effect play out, as claimed by the requirements c), d), e) and f) section 1.1. The full replication paradigm satisfies applications that require instantaneous responses to user actions, i.e. applications where the requirements a) and b) of section 1.1 have a higher priority than the requirements e) and f). Implementations of the full replication paradigm base on the optimistic assumption that any divergence in the local state is resolvable while synchronizing with the other sites. In contrast, implementations of the read replication paradigm are pessimistic in this regard. For this reason full and read replication solutions are also said to realize either an optimistic or a pessimistic approach for state sharing in interactive distributed applications. In the following sections the basics of all relevant optimistic and pessimistic approaches are introduced and discussed with regard to the appropriateness for WHAs. A user action causing a resource state change is generalized as event. The term event is also used as synonymous to user action hereafter.

3.3 Lockstep Synchronization

Lockstep Synchronization (LS) is an implicit method for the realization of read replication. It complements algorithms that manage resource access pessimistically. In LS the state of the shared resource advances step-wise. Each site has to issue exactly one event per step and must not proceed with the next step before having received and acknowledged the events of all other sites for the same step. Since events are transported with messages, event delivery suffers from network induced delays. LS can thus be realized for decentralized and centralized systems. For the latter case an example is depicted in Figure 3-3.

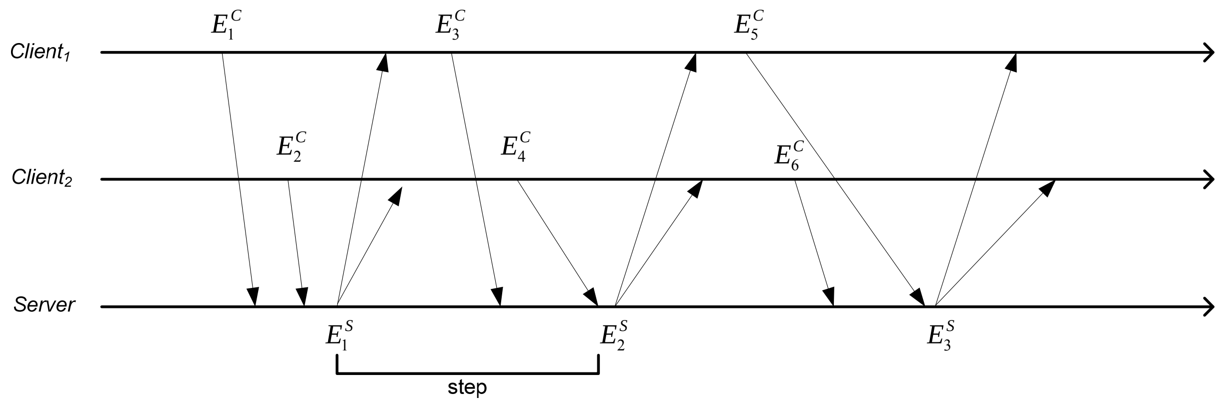


Figure 3-3: Processing events in Lockstep Synchronization

The request for a change of the state, for example in response to a user action, is not applied by the issuing client site to the local instance of the state immediately, but first send as event to the server. When all events are received at the server, the server aggregates the changes to the state. All events of the same step are considered concurrent. An event from step *s* is said to precede an event from step *t* if *s*<*t*. Competing changes are resolved or rejected by the server. For that purpose they can be ordered and applied sequentially or they can be applied in parallel by utilizing transactional memory concepts [53, 54]. After aggregation and conflict treatments, the server forwards the changes as new events to the clients. A client that receives a server event applies the changes to the local instance of the state, advances in processing and sends the next event. The next event implicitly serves as acknowledgement for the last step.

LSS ensures the presentation of events to clients is more or less simultaneous. The event messages are still exposed to different latencies depending on the network connection between client and server. The performance of the entire system depends on the latency of the last responding client. Additionally, the duration of steps can vary. This makes the fluent advancement of the state along fixed length real-time intervals challenging. LS is thus inappropriate for games based on fast user action and reaction. Client errors and resultant missing events have to be caught with timeouts. Improvements of LS especially with regard to cheating in competitive user interaction on a decentralized setup are presented in [11].

3.4 Bucket Synchronization

An improvement of LS is Bucket Synchronization (BS) [55]. Sites do not wait an arbitrary time for events like in LS, instead all sites advance processing in fixed length time intervals. BS is applicable for centralized and decentralized systems. Figure 3-4 depicts BS by the example of a decentralized configuration.

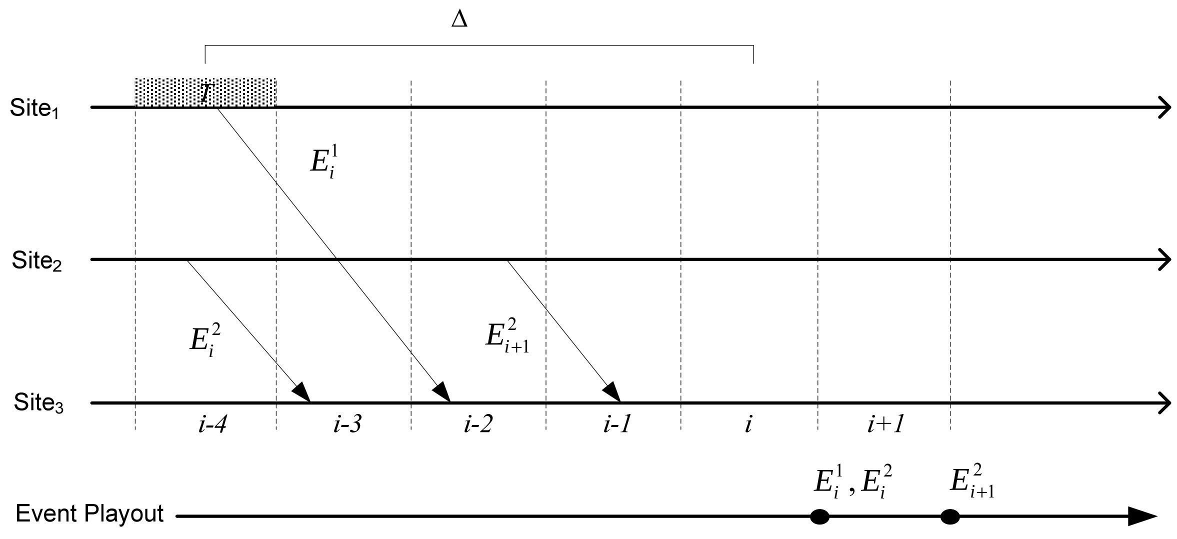


Figure 3-4: Processing events in Bucket Synchronization

A premise of BS is a global clock to synchronize the timing of the sites. Gautier et al. utilized Network Time Protocol [47] for this purpose. The time is subdivided into intervals of fixed length *T*. All events issued in the same interval *i-Δ* are marked to take effect at the end of interval *i*. Thus, each site delays local events by *Δ* intervals to ensure the messages transporting these events to other sites have reached their destinations on time for simultaneous play-out. The delay factor *Δ* has to be chosen with respect to the connection with the highest message delay. Message delay and jitter should be measured continuously. Events that do not reach a site on time are scheduled for a later bucket. For simplification Figure 3-4 only depicts the events sent from *Site$_1$* and *Site$_2$* to *Site$_3$*, although both sites would need to send these events to each other too. The second event of *Site$_2$* also arrives at *Site$_3$* before interval *i* takes effect. Nonetheless, the event is delayed at the receiver until *i+1* takes effect.

BS satisfies simultaneity as consistency objective. Accordingly, the concept is suited for the realization of pessimistic resource sharing. Like LS before, BS is not explicitly designed to synchronize representations of resources residing at different sites. Events that imply competing write accesses either have to be processed sequentially or addressed with, e.g. transactional memory concepts [53, 54]. Events that are received late at least at one site represent an unaddressed problem in BS. An event for interval *i* that is received at interval *i+1* could arrive after new events have already been issued. In that case the new events do not causally depend on the late event and thus refer to an inconsistent state. In a centralized realization of BS a server would process all concurrent events before an interval is due and forward the results to all sites on time. Late events could be moved to a later bucket. In this case the effect of an event may differ from the one intended by the user, but at least it is the same at all sites and the state representations do not diverge.

The simultaneous application of events makes BS suitable for use cases with competing interactions among users, where the play-out time of an event, i.e. a user action, has a significant impact on the experienced fairness. Stock price tickers or quiz games with a buzzer are examples of such use cases. The drawbacks of BS in this regard are probability and distribution of early received events. An early received event is an event that is received at least one interval before its intended interval. If a site is manipulated, users may obtain presentations of an early event and react ahead of time. They thus have an advantage. Since the time an event is received depends on the delay of the corresponding message, users with a less delayed connection always have an advantage. In this case encrypted messages and mutual rating and voting on sites can be applied to ensure a certain level of fairness [12].

3.5 Time Warp

A straightforward optimistic approach for state synchronization is to immediately apply a change locally and to communicate relevant events to all other involved sites without any delay. In the following this approach is called Event Forwarding (EF). EF results for *n* sites and only one event per site in at most n^2 messages to communicate these events. An improvement of EF utilizes a centralized architecture (clients and multiple servers) to reduce the number of messages [56]. In this later Advanced Event Forwarding (AEF) approach a server collects all events, summarizes them and forwards one event to each site. This limits the number of messages exchanged in worst case at the same time among *n* sites to *2n*. AEF can additionally filter the events forwarded to each site by relevance. The relevance is determined by the view each site maintains for the resource state. Only events about changes of the state subset that is currently part of the view are forwarded. For example in a distributed computer game only those events could be relevant that affect the current room of the user character. Event filtering results in smaller message sizes and less processing load at each user site.

Even if EF and AEF ensure that all messages sent from one site are received at any other site in the same order, they respect neither the causality of events across sites, nor competing changes to the state. They are thus only applicable if the subsets of the shared state changed by each site are disjoint. The basic optimistic approach that respects event causality and competition is Time Warp Synchronization (TW) [57]. TW is an early, decentralized mechanism and designed for multi-processor as well as networked, distributed systems. Changes to the state copy are executed in TW immediately when events are received. If an inconsistency is detected, for example caused by the out-of-order application of events, a state replica is rolled back to the last known consistent state and all subsequent changes are applied again in the correct order.

A state in TW is represented as a set of variables. These variables are modified by actions. All actions originated by the same site at the same time are summarized as one event. For usage within a distributed system, each event is shipped with one message. TW is based on sequential consistency. The order of events throughout the system is preserved by a concept Jefferson calls virtual time. In principle, this virtual time and all implications for the relations of events are derived from Lamport [46]. Therefore, events are not only annotated with sender and receiver name, but also with a timestamp in one-dimensional, logical time. The timestamp represents the moment the event should take effect at the resource and thus any replications. Following Lamport, the execution of an event stamped with a time later than the current local time at the receiver results in the correction of the local time to the time of the time stamp. The next locally issued event increases the local time by one and the corresponding message is sent with the new timestamp.

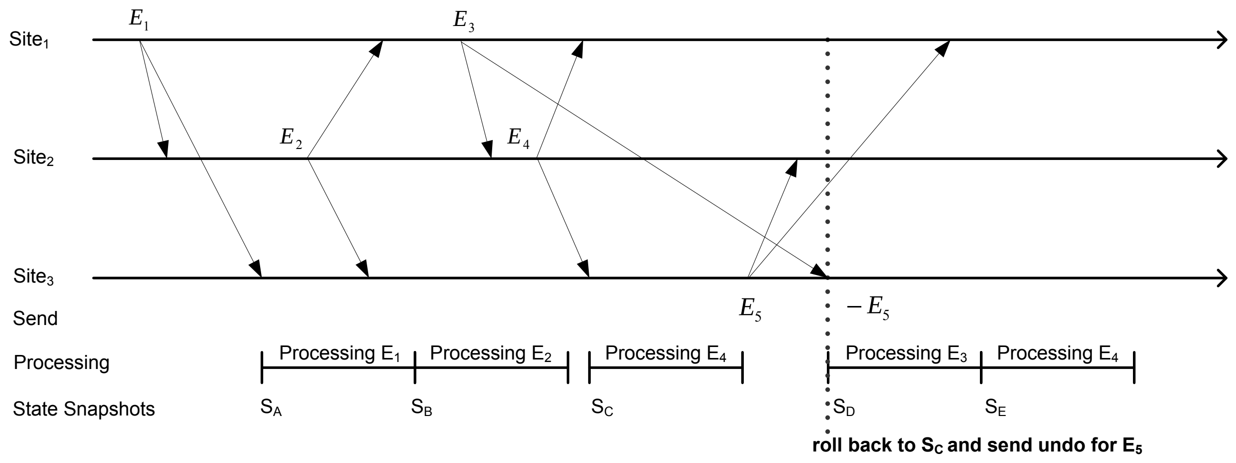


Figure 3-5: Processing of events in Time Warp mechanism

Figure 3-5depicts the processing of events at the receiver by the example of *Site₃*. Normally, all received messages are collected in an input queue and sorted by their timestamps in ascending order. The events in the queue are processed after they arrive as soon as the site is idle. Even if the event timestamp suggests its causal dependence on an as yet unreceived event the processing is not delayed. However, all events remain in the input queue even after being processed in case a roll back is required. For the same reason a snapshot of the current state is made before the execution of any event.

An event is considered late if it carries a timestamp which is before or equal to the current virtual time at the receiver. If a late event arrives, such as event E_3 at $Site_3$ in Figure 3-5, the system jumps back to the last consistent snapshot of the state, rolls back the input queue accordingly, inserts the late event and (re)applies all events in the input queue starting with the newly inserted one. Two subsequent events do not have to be causally related, but only the consequent rollback ensures the convergence of states across all sites.

The complexity of a roll back increases if the site that has to perform the roll back had already send events itself based in the incorrect state. For example $Site_3$ in Figure 3-5 already sent E_5 before recognizing an inconsistency. For this purpose, each site maintains an additional queue with undo messages for all messages it sent. When an out-of-order event is received that reveals an inconsistency, the undo messages stamped at a later time are sent immediately.

A site receiving an undo message has to delete the corresponding message from the input queue if the event has not yet been processed. If the contained event was already processed, i.e. the actions were applied to the local state, the site has to perform a rollback to the last state snapshot before the message to be undone was applied. This eventually includes the sending of own undo messages. Although the cascaded undoing of events does not result in live- or dead-locks, it may result in a considerable numbers of messages communicated among the sites.

Events originated between a virtual time and the moment of a rollback is required are discarded like event E_5. While in use cases like games a rollback could be experienced as a minor skip, the disappearance of previously typed characters or drawn lines in serious office applications could be critical. Jefferson argues that most undoes are performed before an event is processed, i.e. as long as the event is in the input queue. However, this only holds if the processing of an event takes considerably longer than its transport between any two sites. The rollback is the more likely default case if utilizing TW on wide area networks.

3.6 Trailing State

Cronin et al. introduced in [58] an improvement to Time Warp and called it Trailing State Synchronization (TS). TS maintains a delayed, consistent state that is used to patch the state presented to the user if needed. TS does not require rollbacks and corresponding undo messages. Additionally, it reduces the memory load caused by the state snapshots of TW. TS is designed for decentralized systems. Nonetheless, it is applicable in centralized systems too, while the central instances only serve as event hubs but do not have to process events themselves. The state representation in TS is a usually a set of unrelated variables or a set of objects with object variables. The notions of action, event and message are the same as in TW.

Also the ordering of events is ensured by logical clocks and timestamps on the messages communicated between the sites.

The central concept of TS is to maintain at least one copy of the application state that is intentionally delayed, the so called trailing state. All locally originated and received events are instantly applied to the main state, the state rendered to the user. Additionally these events are queued and applied with a fix delay to the trailing state. This concept is depicted in Figure 3-6. If an event like *E₃* arrives that causally precedes events waiting for application to the trailing state such as *E₄* and *E₅*, then *E₃* is scheduled for application to the trailing state before the events already waiting. When *E₃* was applied to the trailing state, an inconsistency to the rendered state is likely. Instead of rolling back events, all events in the queue waiting for application to the trailing state are applied immediately to the trailing state and the resulting state becomes the state rendered to the user. In practice, TS can utilize multiple trailing states with different delays to the rendered state. That way, very late events can still be processed, while inconsistency at a time close to the logical time of the rendered state can be fixed with less processing effort.

Cronin et al. tested the TS in very demanding multiplayer network games and found that most patches of the rendered state are not perceived by the users. Nonetheless, patches may invalidate local user actions that were only applicable in an inconsistent local state.

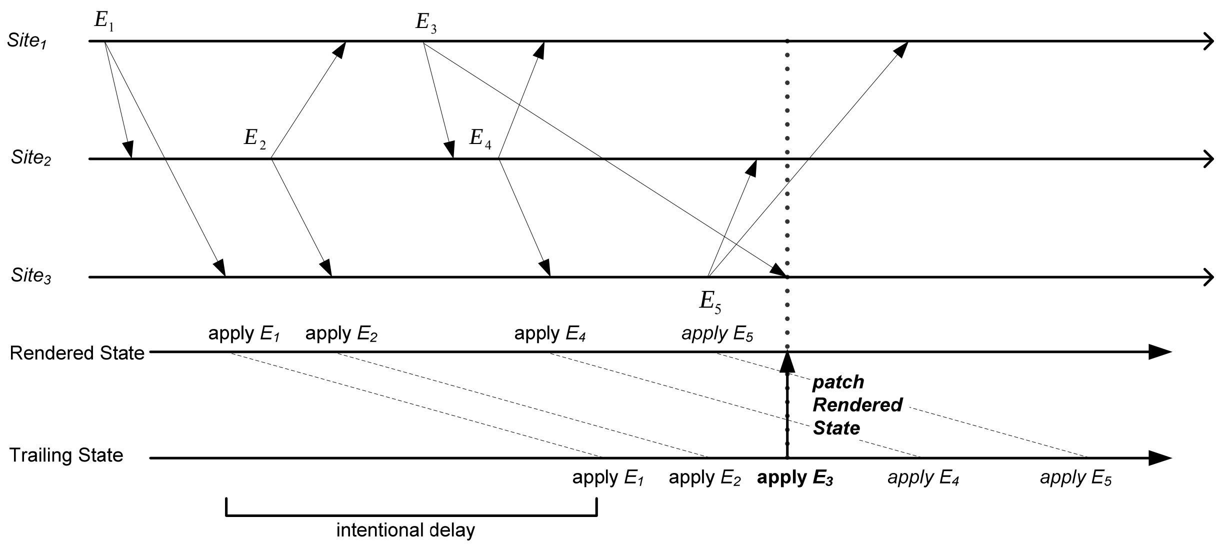


Figure 3-6: Processing of events in Trailing State

3.7 Operational Transformation

Operational Transformation (OT) is an optimistic approach for the synchronization of state copies in groupware systems. In particular OT is designed for real-time collaborative applications which allow multiple users to concurrently edit the same text document. OT systems monitor the actions of users such as insertion or deletion of letters. Actions are represented as operations which remote sites have to apply to their local state instance in order to achieve the same state as the originating site. An operation is in principle a parameterized function invocation. For example, the operations O_1=*delete*(*1*,"*a*") and O_2=*insert*(3,"x") transform a state "*abcd*" into the state "*bcxd*". The first parameter of the operations refers to the offset in the text; the second parameter contains the character to modify. If all sites in Figure 3-7a) start from the same initial state and apply the operations O_1 and O_2 in the same order, they should obtain the same final state. Since OT is optimistic, both operations are applied to the state instance of Site1 before they are sent to the other sites.

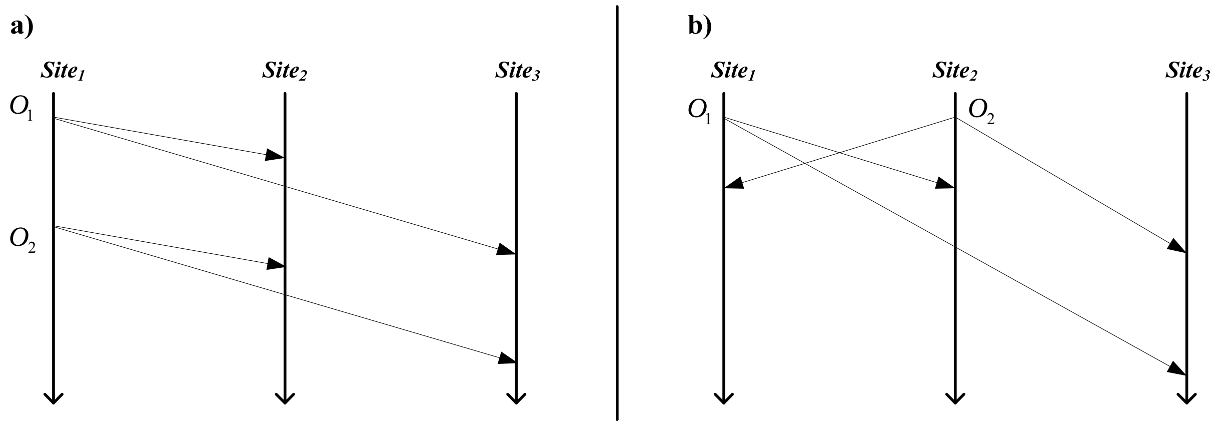


Figure 3-7: Preceding and concurrent operations

If in contrast O_1 and O_2 are issued concurrently like in Figure 3-7b), Site1 would obtain the final state "*bcxd*" while Site2 and Site3 would obtain "*bxcd*". Both operations are competing in this case and their order of application results in different final states. Most OT implementations utilize logical clocks [48] and vector times to maintain a global order of operations. Total order is achieved by additionally introducing priority rules, which for instance order concurrent operation by the indices of the issuing sites. Applying operations in the same order at any site however, is not necessarily sufficient to preserve the intention of users. Operations waiting for application also have to be adapted to the effects of competing operations. This process is called inclusion transformation or forward transposition. For example, $Site_1$ has to find an O_2' which has the same effect as O_2 at $Site_2$, but after the application of O_1. The intention of O_2 was to insert an "*x*" before the "*c*" in "*abcd*". After the character "*a*" was removed the "*x*" has to be inserted at position 2 to preserve the original intention. A correct transformation O_2' would thus be *insert*(*2*,"*x*"). A transformation O_1' at $Site_2$ would look like O_1, since O_2 does not affect the deletion at position 1.

A straightforward realization of OT is the GOT algorithm [59]. Operations are totally ordered and executed at all sites in this order. The operation processing is done like in TimeWarp algorithm. Once a remote operation O_{new} is received, all locally executed operations logically positioned behind this late operation are undone, O_{new} is applied to the state and the undone operations included with the new one and redone. The correct redo however, requires a special procedure. The inclusion of the new operation O_{new} to an undone operation O_i also has an impact on the undone successor O_{i+1} of O_i, which was either issued in awareness of O_i or later transformed to reflect the effects of O_i. Thus, the effect of O_i has first to be excluded from O_{i+1}

which results in O_{i+1}', then O_{new} can be included in O_i to O_i' and finally O_i' included in O_{i+1}'. This has to be repeated for all undone and redone operations. The exclusion transformation, also called backwards transformation, is the reverse of the inclusion transformation and a peculiarity of GOT and SOCT2[60].

The original idea of OT as realized in dOPT was to apply operations in the order they are received and to delay remote operations only until all causally preceding operations have been received and applied. For this purpose, the order of including and applying any two concurrent operations must not result in different states [61]. Ressel et al. formalize this property of the inclusion as *Transformation Property 1* (TP1) [62].

Definition 3-1: Transformation Property 1 (TP1)

> For any two concurrent operations O_1 and O_2 the inclusion transformation function *transform* has to satisfy the property:
>
> $transform(O_1, O_2) \equiv transform(O_2, O_1)$

The property requires that the inclusion transformation preserve the relation of the effects of O_1 and O_2, independent from the transformation order. The practical consequence of this definition is $apply\ (transform(O_1,\ O_2),\ apply(O_2,\ S)) = apply\ (transform(O_2,\ O_1),\ apply(O_1, S))$, where *apply* evaluates an operation in a state and returns the resulting state. TP1 alone however, is proven to be insufficient to realize OT correctly between more than two sites [60, 62, 63]. The reason can be found in the priority rules, which are used to establish an order among concurrent operations and complement the partial order provided by logical clocks. If sites diverge by more than one operation in state, i.e. one operation is concurrent to two or more operations of another site, the states may not converge.

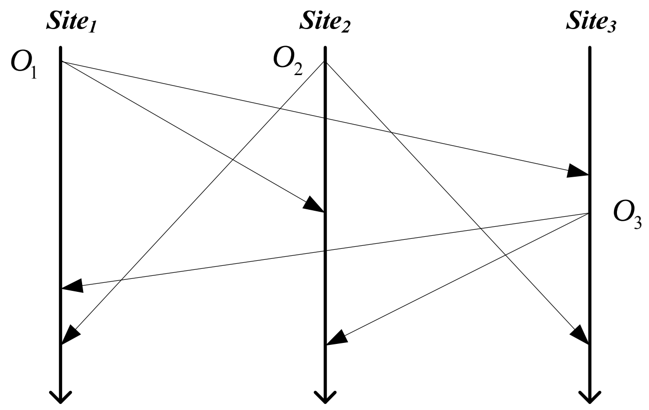


Figure 3-8: Concurrence of multiple operations

Figure 3-8 depicts an example. The operations O_1 and O_2 are concurrent, while O_3 causally depends on O_1. The initial state at all sites is “abcd”, O_1=*delete*(*1*,”*a*”), O_2=*delete*(*3*,”*c*”) and O_3=*insert*(3,”x”). According to dOPT O_3 would be applied to the state without changes at $Site_1$, while both operations have to be included with O_2 before its application. The final state at $Site_1$ is "bxd". The same happens at *Site*$_3$. *Site*$_2$ in contrast includes the effects of O_2 in the concurrent operations O_1 and O_3. Neither O_1 nor O_2 are changed, so O_3 and O_2 refer to the same offset when O_3 has to be transformed. If now a priority rule is used based on the site index, O_3 could be logically positioned before O_2 and therefore bypass a necessary modification. The final state at *Site*$_2$ would thus be "bdx", which diverges from "bxd".

An approach to prevent the above situation is to ensure a global order for concurrent operations. SOCT4 utilizes a central sequencer for this reason [64]. The sequencer provides all sites on demand with an operation number. The number is taken from a continuous sequence. Before sending an operation each site waits until all operations with a lower sequence number have been received, includes these operations into the operation to be sent and finally sends the operation. The sequence numbers replace logical timestamps. Another solution for the above problem is the introduction of a second property for inclusion transformations [60, 62]. Ressel et al. formalize this as *Transformation Property 2* (TP2).

Definition 3-2: Transformation Property 2 (TP2)

> For any three concurrent operations O_1, O_2 and O_3 the inclusion transformation function *transform* has to satisfy the following property:
>
> $transform\ (O_3, transform(O_1, O_2)) \equiv transform(O_3, transform(O_2, O_1))$

Like TP1 before, TP2 requires the inclusion transformation to be agnostic with regard to the order of transformations across multiple steps. Practically TP2 claims:

$apply(transform(\mathrm{O}_3, \mathrm{transform}(\mathrm{O}_1, \mathrm{O}_2)), apply(\mathrm{transform}(\mathrm{O}_1, \mathrm{O}_2), (\mathrm{apply}\ (\mathrm{O}_2,\mathrm{S})))$ =
$apply\ (transform(O_3, transform(O_2, O_1)), apply(transform(O_2, O_1), (apply\ (O_1, S)))$

Accordingly, the adOPTed algorithm bases on the idea to follow all potential orders for the inclusion of concurrent operations [62]. For this purpose the algorithm requires maintaining an *n*-dimensional interaction model with all these 'paths', where *n* is the number of sites. The interaction model is used to ensure all operations can be transformed in a context which equals the context of their origination. GOTO and SOCT2 base on the idea of restoring the situation where sites do not diverge by more than one operation in state and all preceding operation were executed in the same order [60, 63]. Concurrent operations which satisfy this condition are said to be context equivalent. The algorithms are similar to adOPTed, but require a site to explore an alternative order of inclusion for operations only on receipt of an operation that was executed in such an alternative context. Context equivalence is only required for transformation. Therefore GOTO and SOCT2 successively recover the executed operations by an exclusion transformation – similar to GOT – and include the resulting operations in the order required for a safe transformation of the received one. For example, when O_3 arrives at *Site*$_2$, O_2 is excluded from O_1', which is the version of O_1 already applied at *Site*$_2$. Afterwards O_1 is included with O_2 to O_2' and O_2' included with O_3. O_2' and O_3 are context equivalent.

Compared to GOT the algorithms adOPTed, GOTO and SOCT2 require less rollback since they make use of *TP1* and *TP2* and therefore perform better in practice. But while the algorithms are fine, the transformation functions proposed in combination with them do not satisfy *TP2* in all cases [65, 66]. Consequently, these transformation functions only work reliably in a setup with two sites, unless workarounds such as tombstones are utilized [66]. In a centralized system, a server can order concurrent operations uniquely and bypass the requirement of *TP2*. The Jupiter algorithm is very similar to GOTO and SOCT2 [67], but the interaction model is 1-dimensional only. Whenever a past operation is included with a recently received one, the past operation is also transformed with respect to the new one. All later transformations are thus always performed on context equivalent operations. If for example the interaction in Figure 3-8 were limited to $Site_1$ and $Site_2$, while $Site_1$ originates O_3 too after O_1, then $Site_2$ is required to include O_1 into the stored version of O_2 and later transform O_3 against the resulting operation O_2' in the store.

OT is applicable to any data structure with directed order, including graphs and trees. Data models and transformation functions have been proposed for text with tree-layout [68] and the Standard General Markup Language (SGML)[69].

3.8 Differential Synchronization

An approach similar to OT is Differential Synchronization (DS). DS emerged from version control systems in collaborative document editing and was not originally designed for interactive applications. Normally, state instances at different sites are allowed to diverge for hours, but DS approaches are also suitable for centralized interactive applications [70].

In contrast to OT, DS realizations do not monitor all user actions to derive inherent operations. At fixed intervals the user version of a state, usually a text, is compared to its version of the last interval. The changes that have been applied by the user are isolated as differences. In addition to each difference, meta data is collected which describes the context of the change in detail. The context is supposed to help the receiver of a difference to find the position in text where the changes have to be applied. For example, the context can include the text that was previously at the same position, the text before and after a new text fragment or the number of characters inserted.

The differences found per interval are sent to another user site. Since the other user may have applied changes in the interval too, the differences have to be reflected in the current version of the state and the version of the last interval. The resulting modifications which update both states to versions that include the remote changes are called patches. After the patches have been applied, the changes performed by the user are identified and sent back to the other site as differences again. The patching process repeats at the other site.

If more than two sites are involved, a server can be used to organize the patching procedure [70]. If the interval chosen for deriving differences and applying patches is short enough, DS applications are perceived as being distributed interactively by the user. DS realizations do not necessarily rely on timestamps. The context of a difference is sufficient to correctly patch a state. However, the text context alone may not be sufficient to create a correct patch. Especially if states diverge too much, additional information about user intentions and semantics of differences is required. For this reason DS is also combined with OT [71], while their improvement is the subject of ongoing research.

3.9 Discussion

Pessimistic synchronization approaches delay local events artificially, but ensure with small variations simultaneous play out of events at all sites. LS and BS are thus appropriate solutions if simultaneity of events is a higher priority than instantaneity of local events. This is usually the case if an advance in time also means an advance in interaction. Trading or gaming applications are examples for corresponding use cases. LS and BS prevent user perceivable roll backs, since all concurrent events are processed at once before playback. The pipelining of BS has over LS the advantage that it does not block for the time of a step. BS thus satisfies the requirements c)-f) in section 1.1 and is especially suited for competitive user interactions.

EF and AEF are optimistic approaches, but since they are not really suited for concurrent write access to the same state they are inappropriate for the realization of read and full replication systems. TW and TS are agnostic with regard to user intention. Events are applied in total global order at all sites. Virtual time only ensures partial order. The total order is obtained by artificially ordering concurrent events, e.g. by site indices. However, concurrent events could have a different appearance if they were issued in awareness of each other.

In TW local operations are even undone without replacement once a late operation has to be caught up. Users experience this undo as removal of their actions. Such a removal may be satisfactory in games, but not in collaborative editing use cases. TS could benefit from the findings of DS and advance to a suitable approach for collaborative interaction. However, DS cannot resolve inconsistency reliably, so a combination of DS and TS would require an extensive evaluation to find out whether both approaches actually compensate the shortcomings of each other or not.

Compared to TW and TS, OT has the advantage to not require the maintenance of trailing states, state snapshots and the execution of explicit patches or rollbacks. The transformation of operations is isolated from the actual state at site and only affects the operation parameters. Without the need of TP2, OT realizations are also reliable with regard to state convergence. OT thus satisfies the requirements a)-c) in section 1.1 and is well suited for collaborative interactions. For real-time interactive applications with mixed-use, collaborative or competitive cases, a combination of OT and BS is therefore the most promising approach. Such a combination could trade off between optimistic and pessimistic synchronization with respect to the current operation context. The requirement g) introduced in section 1.1 has to be addressed by an OT tie-breaking rule that is balances the different intentions of interacting users.

4 Related Work

In the following sections recent efforts in porting state replication techniques to WHAs are introduced and discussed. The introduction starts with theoretical efforts, continues with middleware solution and tools and ends with entire software frameworks comparable to the one resulting from this work. The chapter highlights positive aspects, but also identifies inadequacies.

4.1 Theoretical Efforts

WHA Technologies form a flexible platform for the seamless adaptation of most algorithms and concepts known from parallel and distributed computing. Accordingly, also the state of the art about optimistic and pessimistic synchronization of resource representations (i.e. replicated objects) is applicable to this platform with modifications.

In the research area of OT these modifications mainly address the centralized architecture, the polling-based link between user agent and origin server, as well as the scalability needs of applications on the Web.

Related work in the context of applying OT to WHA [72-74] bases on HTTP as sole available communication protocol. A necessary design objective is thus to minimize the number of messages exchanged between user agent and origin server. The approach chosen other authors is to buffer user operations at the client and send them as sequence to the server in predefined intervals. Operations of other clients are piggy-backed on the response from the server.

Differences among the related approaches can be found in the processing of operations at the server. While the work of Wang et al. [72] and Shen et al. [73] is based on a sequential handling of clients, Shao et al. [74] process messages in intervals. In [72] a client is not allowed to send new operations until the server has explicitly acknowledged a previously sent sequence. At any time the server processes only operations of one client. All operations are collected in a central history buffer. When an operation sequence arrives at the server, it is first transformed against all concurrent operations in the history buffer and then appended at the end of this buffer. The client receives as immediate response the list of concurrent operations from the history buffer transformed against the recently sent operation sequence. Individually transformed operations for clients are not stored. The acknowledgement mechanism ensures that a client always sends operations directly causally dependent on the last operation received from the server.

The maintenance of only one history buffer saves physical resources at the server and thus has a positive impact on the scalability as argued in [72]. For each client, the communication of operations appears as two-way exchange with the server. Instead of a vector timestamp per operation, a client references to the last operation obtained from the server to indicate where to continue in the next request. This kind of causality indicator keeps the messages exchanged between client and server short. A drawback of this algorithm is the loss of a temporal relation between operations from different clients. If requests of clients *A*, *B* and C arrive at the server concurrently and are served in the order *A, B, C*, then *A* will get the operations of *B* and *C* with a delay, *B* gets the operations of *A* immediately and the operations of *C* with delay, and *C* gets the operations of *A* and *B* immediately. These variations can lead to irritations in collaborative editing, especially if the users work on the same part of text with different intentions.

The concept and the algorithms of Shen et al. [73] equal the work of Wang et al. [72] in most noteworthy details. While Wang et al. documented their work in the context of the software project Google Wave, Shen et al. provide the descriptions with sophisticated formalisms.

Shao et al. abstain from a strict sequencing of client accesses and instead introduce server timing. The server continuously receives messages with operation sequences from all clients and buffers them in input queues. Operations from the input queues are processed at predefined intervals, the results buffered in output queues for the clients and piggy-backed on the response of the next client requests. The causal relations of operation are also restored by a reference to the last operation received. The server appears to the clients as another client. Batch transforming of concurrent operations from different clients preserves the temporal relations of operations experienced by the user. For example, concurrent operations from sites A and B are played out simultaneously at site C. This is a clear advantage over the sequencing-based approaches in [72, 73]. The need for input and output buffers represents a drawback. Although the buffering expense growth linearly with the number of clients, the server has to maintain the state of the communication session for each client. The need for session management at the server complicates load balancing in large-scale applications.

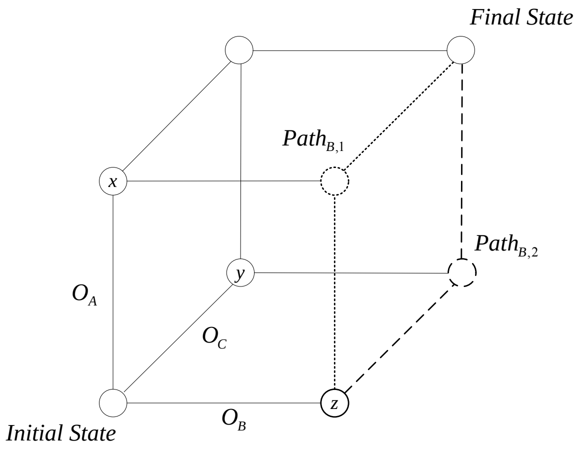


Figure 4-1: State space and transformation paths

If several users edit a document at the same time they produce concurrent operations and temporarily reach non-converged local states. This aspect is shown in Figure 4-1. Three clients *A, B* and *C* start at the same initial state, create two operations each and reach the intermediate states *x*, y and *z* in state space. To enforce the convergence of state copies, each client has to apply transformed versions of the remotely originated operations. As Figure 4-1 indicates, operations can be transformed and applied in different orders. Each order is a transformation path in the state space. Figure 4-1 highlights two of four possible transformation paths for the operations of A and C with respect to the state of B. When operation sequences of multiple clients are transformed, the server has to select an appropriate transformation path for each.

The OT control algorithm used in [72] and [73] bases directly on the findings of Nichols et al. in [67]. Nichols et al. take advantage of a central component all operations have to pass. They use the server to enforce a global order for all operations. The global order prevents two clients from receiving two operations in different orders and thus makes TP2 unnecessary. All operations can be transformed along the same path at all sites.

The OT control algorithm realized by Shao et al. bases on theoretical work of Li and Li [75]. Li and Li demonstrated how challenging the achievement of TP2 for inclusion functions is. They proposed a solution which avoids critical transformations by deriving transformation conditions from the operation history and establishing the transformation path with respect to these conditions. The work of Li and Li emerged from a multi-peer environment without a central point for the enforcement of a global order for operations. WHAs however base on a centralized infrastructure. The advantage of the algorithm proposed by Li and Li over the simpler algorithm of Nichols et al. in the application context is marginal. Using the same transformation path for all clients as in [72] and [73] rather reduces the number of transformations required for each client. Shao et al. do not limit the state model utilized in their work to character, word or token streams, as Wang et al. and Shen et al. Nonetheless, the algorithms of these two should also be utilizable for any data structure that can be mapped to a linear address space (e.g. arrays, trees).

The work of Fraser in the research area of DS [70] was conducted with explicit focus to the web. Based on predefined thresholds (i.e. timeout or character count) the local text copy is analyzed for differences at the web client and the results sent via XHR to the web server. From responses to XHRs the web client obtains patches for the local text copy. The theoretical results of Fraser are complemented with a number of programming libraries and tools for use in WHAs. Thanks to this practical support the algorithms of Fraser have been adopted by a number of applications, as the next section will outline. Although DS is an optimistic approach it is not primarily suited for the instant communication of state changes as already argued in Chapter 3. Documentation of theoretical efforts on porting TW and TS to the web has not been found.

The only pessimistic synchronization technique that reached the web by now is STMS. In [76] Noel describes an extensible software transactional memory approach which is also suited for application in WHAs. With XHRs transactions are send to the server and results as well as copies of updated memory portions transported back to the web client. Pessimistic synchronization techniques that focus on simultaneous event play-out have not yet found the way to the web. This is probably due to the lack of support for instant delivery of server generated messages in the pre-WebSocket era.

4.2 Middleware and Tools

Motivated by the business prospects of real-time applications on the web, since 2006 the engineering community has produced a number of software solutions for message passing between web clients, distributed publish/subscribe in web applications and object replication between web server and web clients. The innovativeness of these solutions can be found in the way of abstracting the technological realities to provide APIs for well-known communication patterns.

The communication pattern most often reflected in software solutions for the web today is message passing. Relevant representatives are CometD [77], Socket.io [78], Strophe.js [79], Kaazing [80] and Jwebsocket [81]. CometD, Socket.io, Strophe.js and Kaazing primarily bridge differences in the availability and implementation of communication APIs in different browsers. For example, they implement several of the push techniques introduced in Chapter 2 as well as WebSocket support, and select the most appropriate communication method at runtime. Jwebsocket is based on the WebSocket API only. In all solutions the software engineer programs an application against a unified, proprietary interface. CometD, Socket.io, Kaazing and Jwebsocket are complemented by a backend part. CometD, Socket.io and Jwebsocket provide server implementations compliant with Node.js and Java. Kaazing includes a gateway which maps messages to technologies more common in server programming, such as XMPP and JMS. Stophe.js is a client-only software library which implements BOSH [82], an XMPP binding for HTTP. Stophe.js has to be used with either a BOSH-XMPP gateway or BOSH compliant XMPP servers.

All named solutions are extendable and in addition to the basic message passing pattern the support publish/subscribe communication patterns in all facets of [14]. Although event channels are a foundation for synchronization techniques in object replication, real support for replicated objects is out of scope for cometD, socket.io, strophe.js, kaazing and jWebsockets.

Software solutions supporting object replication on the web are XSTM, Google-diff-match-patch (abbreviated as *GDMP* in the following) [83] and Jinfinote [84]. XSTM is software transactional memory implementation. A JS library enables web application engineers to create transactions at the web client and send them to the server to be committed. A strength of XSTM is the wide platform coverage. Implementations are, for instance, available in Java, .Net and Objective C. The theoretical foundation is presented in [76]. GDMP implements the DS algorithms of Myers [85] and thus is tailored to text objects. GDMP does not include a solution for the communication with a backend and is limited to a number of JS libraries for text processing. The integration with the sites of a web application has to be done by the application engineer. Jinfinote is a JS frontend for the OT library and tools in the Infinote package [86]. Jinfinote is seamlessly integrated with any WHA that requires collaborative text editing. According to available descriptions, Infinote implements the OT algorithm of Ressel et al. [62] although it utilizes centralized architecture. A TP2 hard inclusion function is not documented. GDMP and Jinfinote address plain text only. Both lack considerations about rich text objects.

4.3 Software Frameworks

While this thesis was being prepared, two holistic solutions for the fast realization of real-time interactive WHAs emerged. The first is Google Wave [87]. Originally introduced as a new service, the software is today available under open source license from the Apache Foundation. The second is IBM Cooperative Web Framework (CoWeb) [88]. CoWeb is available under open source license today from the Dojo Foundation.

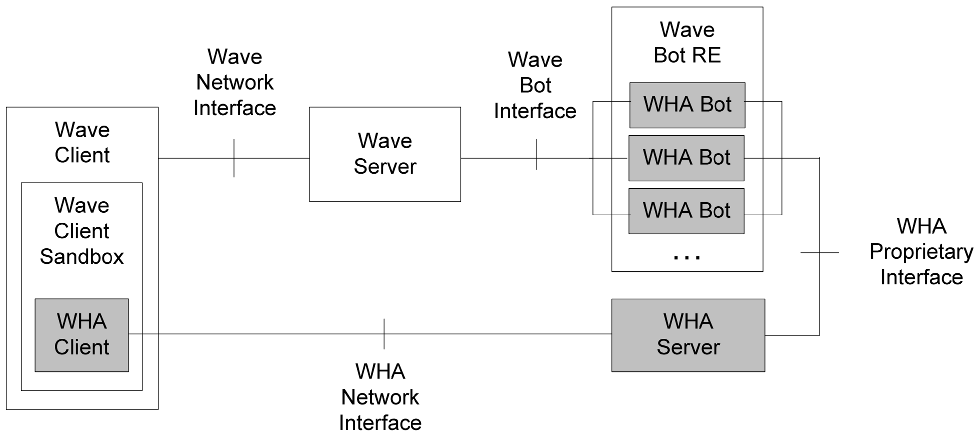


Figure 4-2: Architecture of Wave

Figure 4-2 outlines the software architecture of Wave. Basically, the *Wave Server* maintains a number of sessions. Each session reflects the communication of a group of users. Users may create, collaboratively edit and delete documents within a session. Documents can be ordered, e.g. to form a message thread. *Wave Client* and Wave Server are connected by a virtual, bidirectional message channel which is realized with the techniques introduced in section 2.3.3. The message exchange is used for the realization of the OT concept described in [72]. The documents of Wave sessions are the rich text objects to be replicated.

Wave is a software framework and application platform. Applications created with the framework require the Wave platform as runtime environment. Framework and platform, however, not only enable novel collaboration use cases but also limit the design and presentation options for WHAs. The Wave platform has a user interface for the presentation of Wave sessions at the user agent. This user interface is strongly coupled with the Wave Client and supports a number of use cases from collaborative editing of rich text to search. Using the Wave Client directly from a WHA client is not envisioned. WHA clients have to be designed as layout elements for a presentation within documents of a Wave session. Such a layout element is referred to as *Gadget*. A WHA client must not access the Wave Client and its data directly. Instead, it can access data about the session it is utilized in, as well as a special associative array. The associative array is distributed between all instances of the same Gadget and reflects a shared Gadget state. The array is synchronized instantly between all gadget instances on an update. Concurrent write operations to the same field are serialized at the server. The value of a field always reflects the value of the last update. Combined updates of multiple fields substitute transactions. The WHA client representing a Gadget cannot read from or write to the documents of the session it is utilized in. For that purpose the WHA engineer has to provide a software agent, called *Bot* in Wave terminology. The Bot has its own life-cycle and is executed in a dedicated runtime environment (Bot RE), which is attached to the Wave Server. In contrast to the Gadget, the Bot can listen to changes in Wave documents, create documents and modify document contents. The *Wave Bot Interface* provides the Bot with the same options as the Wave Client does for the user. Additionally the Bot can access several communication interfaces, for example to contact a WHA Server in a proprietary way. A Bot, however, has no means to communicate directly with a Gadget from the same WHA.

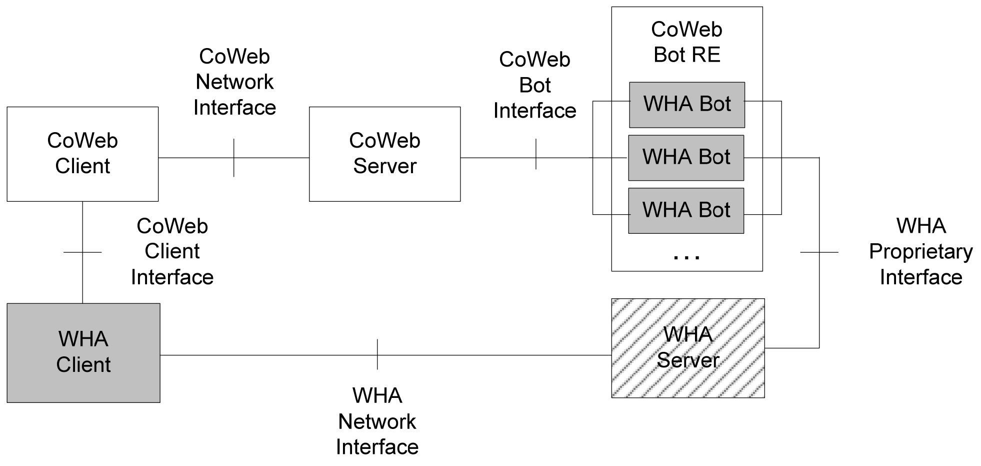


Figure 4-3: Architecture of CoWeb

In contrast to Wave, the CoWeb framework does not tie an application down to a special platform. WHAs are rather developed around CoWeb as depicted in Figure 4-3. The basic components are still similar to their Wave equivalent. The CoWeb Server manages collaboration sessions, where a session groups a number of shared objects. Through a message passing channel realized with Cometd [77], a *CoWeb Client* sends operation requests to and receives updates from the CoWeb Server. CoWeb generalizes documents as named collections. A collection can contain characters and tokens as well as any object. A rich text document is, for instance, represented as a collection of tokens. A shopping list is represented as a collection of objects describing goods. The OT algorithm to modify collections is not documented. In addition to collections, CoWeb directly supports the synchronization of named objects within a session. The algorithm to synchronize attributes of objects equals the algorithm that Wave utilizes to synchronize the associative arrays representing a Gadget state. CoWeb does not support the transactional modification of multiple object attributes at once.

A *WHA Client* embeds the CoWeb Client like any other JS library. The *CoWeb Client Interface* provides broad access to session life-cycle management, session data and even the *WHA Bot*s attached to the CoWeb Server. This Bot concept is very similar to that of Wave, except that a WHA Client and a WHA Bot can communicate directly via a request-response pattern. This peculiarity makes a *WHA Server* obsolete as a separate component in most use cases. However, the CoWeb Server does not provide any other interface a WHA Server could use to access sessions or session data, e.g. to store them persistently or export them. Such interfaces would have to be realized with Bots.

4.4 Summary

In section 3.9 a combination of OT and BS is identified as appropriate to cover a broad range of online interaction use cases. With OT as algorithmic base Wave and CoWeb reinforce this conclusion. Since both solutions are tailored to cooperative work use cases, they do not address competitive interactions as supported by BS algorithms.

The adaptation of OT algorithms for applications on the web is quite advanced. Lessons learned from solutions for mapping full-duplex message passing to a request-response based communication link like CometD and Socket.io simplified the porting of state-of-the-art OT algorithms. OT libraries like Jinfinote serve as notable proof for this development. The efforts of Wang et al., Shen et al. and Shao et al. bring recent findings in OT theory to the web. Although the algorithms of Wang et al. and Shen et al. are well tailored to the engineering practices for WHAs, their strict sequential processing of client requests breaks the temporal relations of operations. Users may have problems understanding which operations are reactions to theirs. Shao et al. propose a bulk transformation of operations from different clients at fixed-length intervals. This approach basically ensures preservation of temporal relations. The synchronization of the server interval with the client intervals would further improve this. Their transformation algorithm uses transformation paths individually computed per client. A performance comparison to an algorithm for transformations along a common path is lacking.

An adaption of BS algorithms for WHAs is not documented. A possible reason is the latency-prone communication of the pre-websocket era.

The software frameworks of Wave and CoWeb, the architecture as well as the component interfaces, follow the same, almost obligatory pattern. The strong coupling of Wave to the integrated platform however, makes Wave the less appropriate choice for the development of a standalone cooperative work application. The missing support for communication between client-site and server-site extensions (Gagdets to Bots) emphasizes this point. CoWeb supports the communication of WHA client and WHA server surrogates (Bots) by shared objects and a request-response mechanism similar to legacy HTTP communication. Also CoWeb is not limited to the synchronization of text like Wave. Shared resources are either treated as simple objects (primitive types and associative arrays) or collections of simple objects. OT operations like 'insert' and 'delete' are applicable to any collection. Missing is the support for the transactional modification of multiple objects attributes at once. Also CoWeb does not support custom data models for shared resources, custom operations and custom inclusion functions. Software engineers thus have to map their required data model to the available data model if at all possible.

5 Instant Synchronization of a Shared State

This chapter describes the concept of the thesis. It introduces a communication scheme for communicating changes to a shared resource. Also the chapter contains algorithms for the resolution of conflicts among concurrent operations, including definitions of basic terms and concepts.

5.1 Overview

The intention of the system proposed here is to support an application engineer in realizing real-time interactive applications. *Ordering and causality preservation, competing access, simultaneity of access effect play-out* and *instantaneity of access effect play-out* are identified as relevant consistency properties in a resource-centric environment in section 3.2. The properties instantaneity and simultaneity cannot be achieved at once for the access effect play-out if the communication of the involved sites suffers from human-perceivable delays, as typical on the Internet. This peculiarity will be covered in the described system by two complementary operation modes a software engineer can make use of for an application accessing a shared resource in the context of an interaction. These modes define whether an access is issued in a pessimistic or an optimistic context, i.e. a competitive interaction or a collaborative interaction. The operation modes are:

(1) **Continuous convergence and effect simultaneity:** Simultaneous play-out of effects resulting from simultaneously issued actions requires all sites, even the site that issued the action, to wait with the play-out until the last site is ready to do so. As a benefit, effects of user actions can be played out in the same order at all clients and checked for concurrency constraints before. Users experience the state to be convergent after the play-out of any event. This operation mode satisfies the requirements a)-g) introduced in section 1.1.

(2) **Convergence on quiescence and instantaneity:** The effect of a local user action is played out instantly. At this moment the site has no information about simultaneous and potentially competing events from other sites. Accordingly, convergence cannot be achieved until the system is quiescent; i.e. all messages have been delivered and processed, no more messages about user actions are on their way. Users experience the system state to be consistent when no other user is active. Intermediate states can diverge, to the extent that a particular state presented to the user at one site is never achieved at the other sites. This operation mode satisfies the requirements a), b), c) and g) introduced in section 1.1.

Although both operation modes are fundamentally different, their combined use within the same application makes sense if the objects comprising the application state have different characteristics and constraints. An assisted browsing application, for example, can allow two users to collaboratively edit text in operation mode (2), while a shared mouse pointer they move is synchronize under operation mode (1).

In the following, the system architecture, a suitable communication scheme for the interacting sites and an algorithm to achieve convergence according to operation mode (1) and (2) are introduced. The communication scheme realizes a BS approach. The algorithm that ensures state copies convergences bases on the theory of OT. The rationale for BS and OT is provided in section 3.9.

5.2 State Representation and Synchronization

The system introduced here bases on variations of the client-server paradigm to simplify the integration with the existing web architecture and its ecosystem. In contrast to peer-to-peer systems a centralized architecture constitutes a single point to supervise restrictions, security policies and execute trusted application logic. This point cannot be bypassed without breaking the application itself and is thus a reasonable requirement of a web application operator. From a technological point of view a client-server architecture simplifies state convergence. Any communication is limited to a message exchange between the two sites. No potentially late or hidden messages have to be reflected in access ordering. The client-server paradigm does not prevent the distribution of the server, for example to scale the system.

The web architecture is designed to obtain the states of resources and mutually change resource states. To cope with the consistency-latency tradeoff discussed in section 3.2 a hybrid read/write replication paradigm is applied. The server maintains a resource state S_{Lead} (also called lead state of the resource). All clients maintain copies of a subset of the lead state. A copy of such a subset is denoted S_{Copy}. In case any non-resolvable inconsistency is detected in a copy, the copy is replaced by the most recent version of the subset in the lead state.

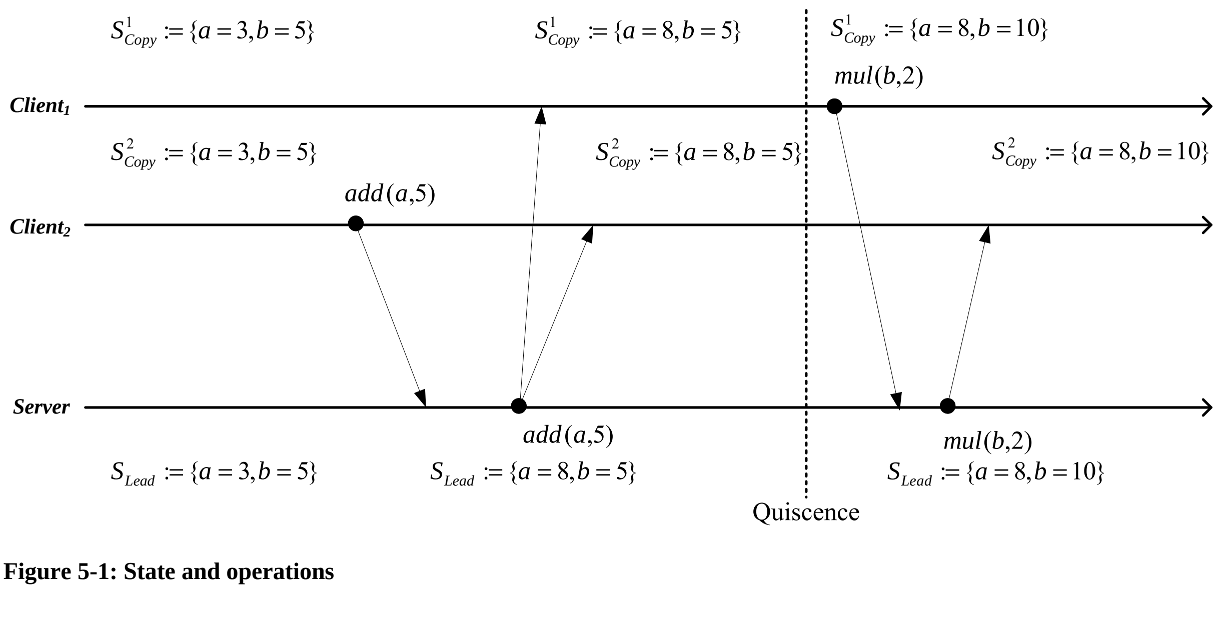
Example 1: Set of numbers	**Example 2**: Tree
{a=5, b=8, c=10}	div
	ul
	li
	hello
	li
	world
	hr

Table 5-1: Examples for state representations

The state has to be representable by any set of uniquely addressable, discrete, interactive objects, e.g. objects which can be changed by an action, but do not change themselves as a function of time. The following examples show two possible state representations, a set of natural numbers and a tree, such as used for the DOM. The objects in the first example are the variables *a* and *b*, which can be addressed by their names. The objects in the second example are the nodes of the tree, such as *div*, *ul* and *li*. The value of a node is addressable by its path in the tree, where *div/ul/li[2]/* could refer to the second *li* node and its value *"world"*.

A user action is represented as an operation. An operation comprises a deterministic function which maps a given state and a list of parameters to a new state, i.e. modifies the state of a resource. Figure 5-1 depicts a strongly simplified example for the communication flow between clients *Client*$_1$, *Client*$_2$ and the *Server*. The state representation used in this example is a set of numeric variables. Operations to change the resource state are defined by the two functions *add* and *mul* in this example. The *add* function adds a constant to the value of a variable in the state. The *mul* function multiplies the value of a variable with a constant and stores the result to the variable. The server does not create operations itself, but all operations requested by clients are communicated through the server. The server does not simply dispatch operations, but may also rewrite operations, for instance in the event of access conflicts between two concurrent operations. In Figure 5-1 the operation *add(a,5)* is applied according to operation mode (1), the operation *mul(b,2)* according to operation mode (2).

Figure 5-1 already points out two major differences to pure read replication: i) an update of the copies initiated by the server does not transmit the entire copy, but only the difference to the last copy sent to the client, ii) clients can apply changes to their state copy before sending them to the server. Property ii) is required to fulfill the operation mode (2), i.e. convergence on quiescence and instantaneity. The dotted line in Figure 5-1 denotes a moment when the system is quiescent.



$S^1_{Copy} := \{a=3, b=5\}$
$S^1_{Copy} := \{a=8, b=5\}$
$S^1_{Copy} := \{a=8, b=10\}$
$mul(b,2)$
Client$_1$
$S^2_{Copy} := \{a=3, b=5\}$
$S^2_{Copy} := \{a=8, b=5\}$
$S^2_{Copy} := \{a=8, b=10\}$
$add(a,5)$
Client$_2$
Server
$add(a,5)$
$mul(b,2)$
$S_{Lead} := \{a=3, b=5\}$
$S_{Lead} := \{a=8, b=5\}$
$S_{Lead} := \{a=8, b=10\}$
Quiscence

Figure 5-1: State and operations

5.3 Communication Scheme

Simultaneity of event play-out, i.e. the effect of applying an operation to the user-visible state requires the synchronization of clients. This synchronization has to respect the different characteristics of the communication channels between client and server, in particular the (throughput-induced) latency of operations. The system proposed here utilizes an adaptation of the bucket synchronization algorithm. Bucket synchronization is a more appropriate choice than lockstep algorithms discrete interactive objects whose states are not dead-reckoned by default while waiting for updates. The following sections explain the basic synchronization process then discuss details of the synchronization process with regard to simultaneity approximation and cheating.

5.3.1 Operation pipelining

The synchronization mechanism subdivides time into so called bucket intervals of length T. These intervals help to cluster operations into groups for simultaneous application. All operations O_i that shall be applied to the local state copy of all clients within interval i have to be issued in the interval $i – a - 1$. The parameter a, called pipeline-depth, determines the magnitude of the play-out delay. The play-out delay itself is defined as $\Delta:=aT$. The play-out of any operation effect to the user is thus delayed by at least the time Δ. An overview of these symbols and their relation to parameters is given in Figure 5-2.

All operations $O_1, \ldots, O_k$ created at a client C during interval i-a-1 are added to a new request message R_i in the order of their creation. The message itself refers to the interval i, for which the contained operations are intended to be applied to the state copies of all clients in the system. R_i must be sent by the client to the server by the end of interval i-a-1. A request message has to be created and sent by each client at the end of each interval, even if no operations have to be communicated. Frequent request messages serve as indicator of the client liveliness and the latency on the channel connecting the client.

The time the server receives a request message R_i has no impact on the play-out interval of the operations contained. The server continuously receives request messages of different client for various intervals. The server orders the received operations, applies them to the lead state and sends the resulting operations $O'_1, \ldots, O'_n$ for the adaption of state copies to all clients. The resulting operations are not necessarily a simple summary of the operations received from all clients. For the purpose of conflict resolution, the server can and may have to change them so that the state copies at all clients converge.

The operations a client is required to apply to its local copy of the state are contained in the update message U_i. The update message refers to the interval i in which the operation should take effect at the client. At the moment client C receives U_i the interval i begins for this client. This moment is referred to as time t_i. The client has to apply the contained operations to the copy of the state. The state change is required to take effect in the presentation to the user without further delay. To ensure simultaneity between clients, the server sends update messages individually delayed by time d. An update message triggers the interval increment and thus the release of the next request message with the operations for interval i+a. The pipelining of update and request messages in the communication scheme is depicted in Figure 5-3.

The latencies of request and update messages are reflected in Δ. Δ is the longest expected period of time that must elapse between the begin of transmission for R_i and the reception of U_i across all clients. It is determined by the latency of the slowest communication channel between server and clients and the time the server requires for processing client operations. Accordingly, for all clients the message roundtrip and the processing of operations at the server should take less time than Δ.

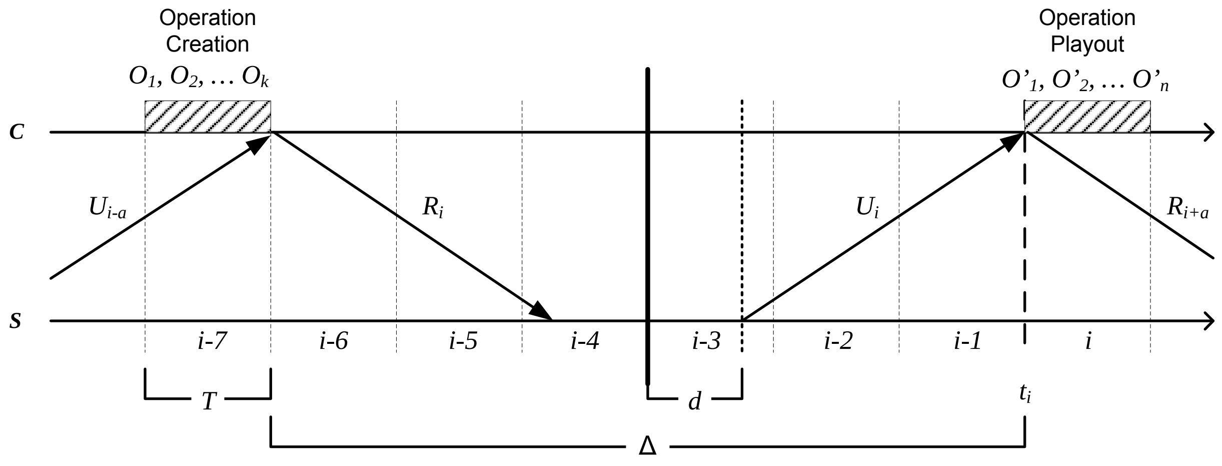


Figure 5-2: Adapted bucket synchronization

The server-controlled timing allows modifying Δ at runtime if request messages tend to be late. A messages is considered late if it is not finally received at time $t_i - \frac{\Delta}{2}$. Figure 5-3 suggests the time required for message reception is negligible. In fact, throughput and message length (number of contained operations) can lead to a significant reception time. To modify Δ the server can adapt the parameters *a* and *T* within predefined boundaries. These boundaries define the best and worst configuration for a distributed, interactive application and thus need to be chosen individually. If the system is already in the worst configuration and more than a predefined number of request messages in a sequence arrive late at the server, the sending client is disconnected. Even if a late message is still being processed, the server must send the subsequent update message on time. If an update message is late, clients are not affected unless they use the time span between any two subsequent updates to estimate the likely arrival time of the next message. This can be reasonable if the states of objects need to be dead-reckoned at the client. For this reason the server changes the parameters slowly over a number of intervals defined by the application developer.

The definition of operation lateness follows the definition of message lateness. However, an operation is not necessarily late if the message it was contained in is late. At the server, an operation for interval *i* is on time as long as its processing can begin at $t_i - \frac{\Delta}{2}$, i.e. it has been completely received at this time. Late operations for *i* are not considered for this interval any more. Depending on when they are received, they take effect in a later interval or they are refused if they have been substituted by operations of a later interval.

Request and update messages are required to be received in the order they have been sent. For the synchronization scheme, message delivery is assumed to be reliable, i.e. messages are neither lost nor are their contents corrupted. However, the consistency objective of the synchronization scheme is to enable convergence of state copies. This does not necessarily include the need for a consistent history of operations at all clients. In contrast, a necessary condition for the communication scheme is that messages can be received and processed interleaved at client and server. The communication channel must thus not be a queue, which blocks the processing of messages for interval *i* as long as still messages of interval $j < i$ have to be processed.

An example parameterization for the communications scheme that follows the definition of the IEEE standard for distributed interactive simulations [52] is $a = 3$, $T = 50ms$, while generally $a = \Delta / T$, $T = 50ms$ is a reasonable heuristic to determine the number of intervals based on the expected roundtrip time for messages between clients and server.

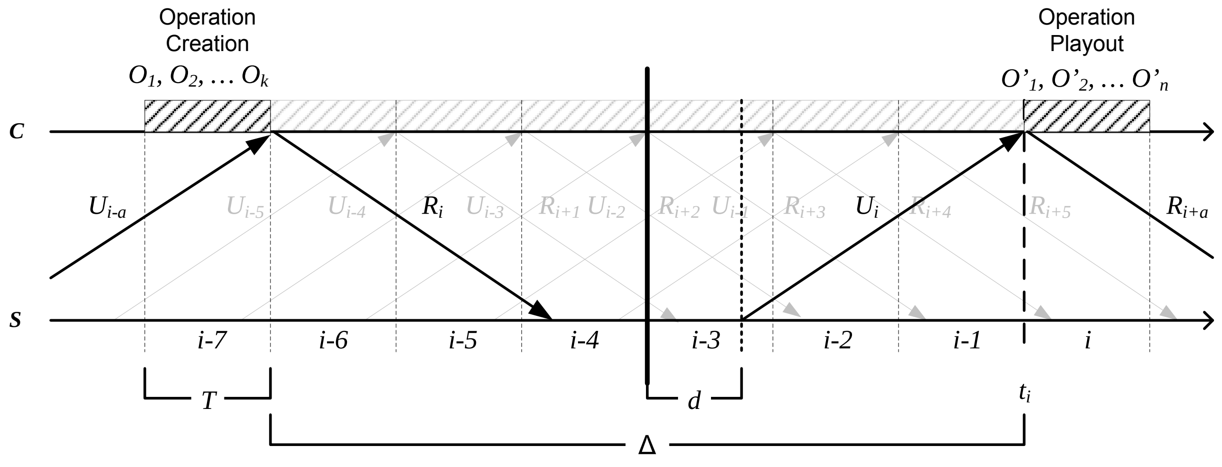


Figure 5-3: Pipelining of request and update messages

5.3.2 Latency estimation and fairness

Ensuring a simultaneous play-out of operation effects across all clients requires the simultaneous reception of update messages at the clients. The latencies of messages can vary from client to client. The main reasons for those variations on the internet are the geographical distance between communicating sites and the different characteristics of access networks (3G, Cable, etc.). Accordingly, the server needs to delay update messages for clients artificially by the expected latency the message will be exposed to. This delay *d* has to be determined individually for all clients as depicted in Figure 5-4. In practice, a grouping of clients with similar send delays is reasonable.

A simple estimation of latencies and the resulting delays d_C for sending updates to clients can be calculated with the roundtrip times for past messages. The roundtrip time ρ_C for a client C is measured at the server and determines the time between the beginning of sending an update messages and the beginning of receiving the subsequent request messages. Instead of only one roundtrip, the average for the w last roundtrip times is calculated to smooth out the jitter effects. Half the difference between this average value and the play-out delay Δ finally represents the value of the imposed delay for a client. This estimate for d_C is summarized in the following formula:

$$d_C = \frac{1}{2}\left(\Delta - \frac{\sum_{j=i-w}^{i} \rho_C^j}{w}\right)$$

This estimate does not ensure an absolutely simultaneous play-out of operation effects. The actual characteristics to which messages will be exposed are not predictable accurately. Additionally, the processing time of events may differ from client to client. Thus, a variation remains between the moments of play-out across clients. Nonetheless, the latency estimation can hide variation of play-out times to a degree that makes them imperceptible for the user. If latencies differ from one communication direction to the other, as for instance in satellite-added access networks, a global clock for physical time and message timestamps have to be utilized to calculate the differences between latencies of both directions.

Supporting the simultaneous application of operations to copies and the simultaneous play-out of effects aims to enable the fair state-based interaction of users. This is required in trading or gaming use cases, where users compete with each other instead of collaborating. Simultaneous play-out of operation effects ensure that all users have the same information at the same time and thus have the same time to react to this information. The reactions are collected in the same interval and should be treated as concurrent by the server.

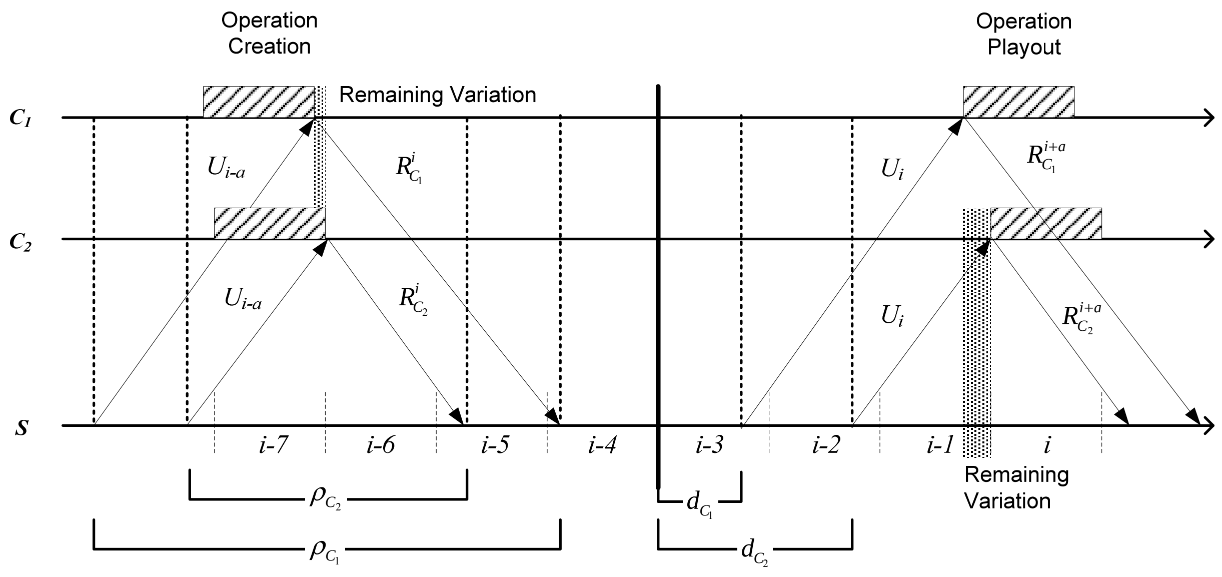


Figure 5-4: Individually delayed update messages

The latency estimate discussed above assumes all clients send their request messages R_i as soon as they receive the update message U_{i-a}. Where users compete with each other, one user can take advantage of manipulating the message timing in the client implementation. An example is given in Figure 5-5. Client C_1 intentionally delays sending message $R_{C_1}^{i}$ by time t_{hold} to make the server believe the round-trip time and consequently the message latency is higher than it actually is. The incorrect value is used by the server to determine d_{C_1}. Consequently, the server sends U_i long before it actually needs to. The message arrives at the client by time $t_{advantage}$ earlier than intended by the server. If the time $t_{advantage}$ is greater than the interval length T, the client can send reactions to the operations contained in U_i one or more request messages earlier. For example in Figure 5-5, client C_1 has enough time to include its reactions to U_i in $R_{C_1}^{i+a-1}$, while the other clients can included their reaction first in $R_{C_x}^{i+a}$. If this reaction of C_1 is a bid in an auction application and the client repeats its behavior continuously, other clients and the users would experience the application as unfair.

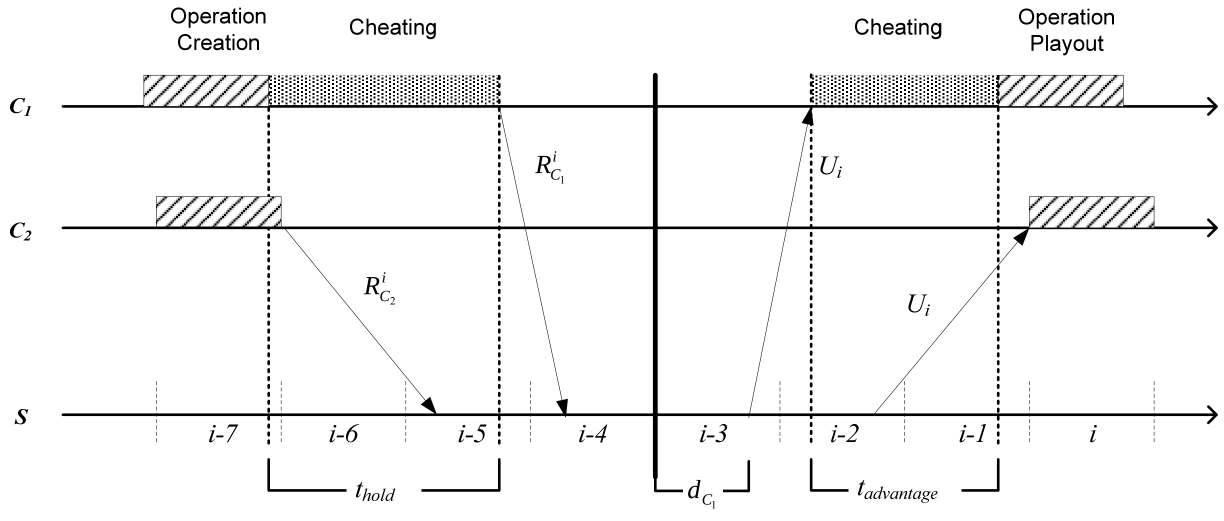


Figure 5-5: Taking advantage of delayed request messages

For applications that have to cope with the above scenario two options are suggested for the implementation of the system proposed here:

a) The cheating problem arises if a client can include its reactions to an update message in an earlier request message than other clients. To remove this possibility, the pipelining of operations can be eased by setting parameter a:=1 and thus $\Delta = T$. In this configuration the proposed algorithm operates like round-based approaches.

b) The client achieves the possibility to cheat by delaying its own request messages and thus disturbs the latency estimation at the server. If the server uses latency information of a third party the client data can be checked for plausibility. Krishna et al. present in [89] a suitable algorithm to determine the latency of an internet host by the latency of the DNS server closest to this host.

5.4 State Copies Convergence

With the communication scheme, a basic algorithm for communicating state changes between clients and server is introduced that preserves the temporal relation between events and thus satisfies the synchronization consistency objective. The sequencing of messages and ordering of operations within messages are additionally the foundation for the maintenance of a global operation-effect order. But how is operation concurrency addressed in order to achieve state convergence? To motivate the challenges for both configurations, continuous convergence and convergence on quiescence, Figure 5-6 shows an example.

For simplicity, the messages and intervals are hidden in the figure. Each operation is contained in a single message. The state is represented as a set of the numeric variables *a:=2* and *b:=4*. The state variables can be modified by the two functions *add* and *mul*. The clients *Client*$_1$ and *Client*$_2$ simultaneously create the operations *O*$_1$ and *O*$_2$, later *Client*$_1$ creates operation *O*$_3$. The *Server* receives the operation in the order O$_2$, O$_1$, O$_3$. To correctly process the operations the *Server* needs to know that apart from the order of reception, O$_1$ and O$_2$ emerged from the same initial state, while *O*$_3$ bases on the assumption *a=4*.

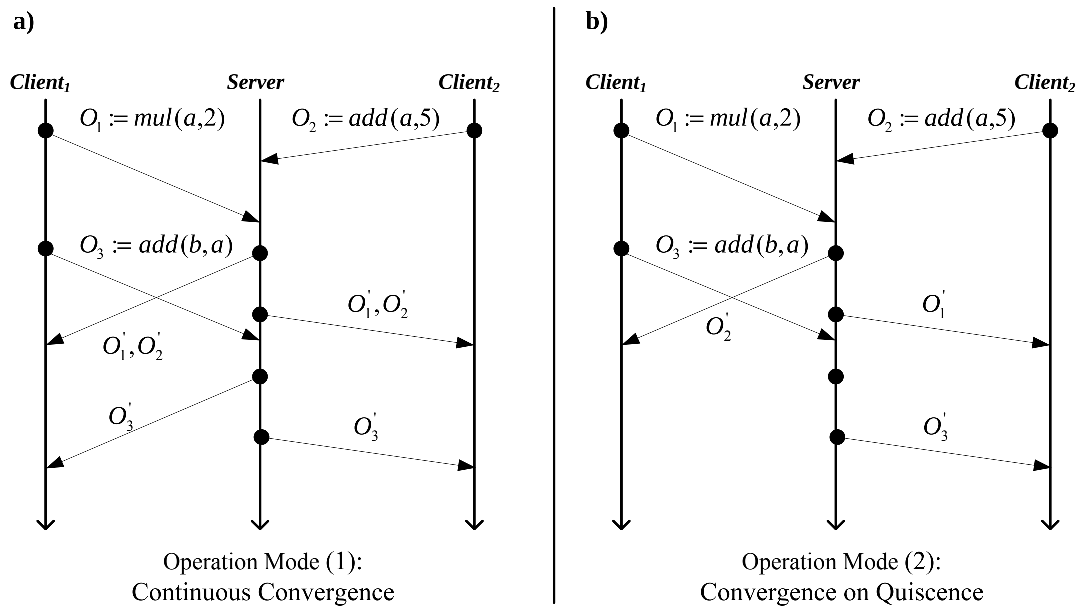


Figure 5-6: Challenges in copies convergence

Considering the causal relations of operations at the *Server* only is not necessarily sufficient. Depending on the order of applying *O*$_1$ and *O*$_2$, the lead state is finally either S$_{\text{Lead}}$ = *apply*(*O*$_3$, *apply(O*$_1$*, apply(O*$_2$*)))* = {a=14, b=18} or S$_{\text{Lead}}$ = *apply*(*O*$_3$*, apply(O*$_2$*, apply(O*$_1$*)))=* {a=9, b=13}. If the entire system is operated in mode (1) as depicted in Figure 5-6a), i.e. updates are requested but first executed when the corresponding updates from the server are received, the server can decide on the global order of operations. Nonetheless, the server needs to know that *O*$_3$ depends on *O*$_1$, rather than on O_1' O_2'. If, additionally, the system operates in mode (2) as depicted in Figure 5-6b), *O*$_3$ has already been applied to the local state copy. *Client*$_1$ has to consider O_2'

concurrent to O_3 and needs to know whether O_1 was respected by O'_2 or not. In this case the decision of the *Server* on the order of applying O_1 and O_2 significantly affects the convergence of copies.

The deterministic application of all operations in the same order at all sites can ensure state convergence, but not the preservation of operation semantics. The sequential application of competing operations, for instance two operations changing the same variable in the system state, can finally result in the loss of operation effects. In this case users can get the impression that their intentions are not reflected in the converged state. The following section will describe how causally dependent and concurrent operations are recognized in the system described here. Additionally, conditions are defined for determining competing operations and an algorithm is introduced for resolving competitions while preserving the original intentions of users.

5.4.1 Preservation of Operation Causality

The first step towards state convergence is to enable all sites in the system to bring operations into the same order before applying them. For ordering operations in the system partial ordering is favored over total ordering for performance reasons. Accordingly an operation O_1 can precede an operation O_2 or both operations can be concurrent. An operation O_1 causally precedes O_2 if the process which issued O_2 was aware of the effect of O_1 on the state from which O_2 was created. Two operations O_1 and O_2 are said to be concurrent if neither precedes the other. Thus, both operations were created independently.

To enforce partial ordering vector time is utilized, a variant of logical time [46]. A peculiarity of the system described here is the architecture that hides clients from each other. Any communication is pairwise between client and server. Accordingly, the vector used to indicate the logical time at the sites has only two components, one for the client, and one for the server. For shipping with messages, each operation is annotated with an instance of this vector as reference to the state the operation originated from.

Definition 5-1: State Reference

> The state reference is a two component vector of natural numbers with the form $Reference := \binom{c_{client}}{c_{server}}$ whose components c_{client} and c_{server} are addressable by the constants *client* and *server* and the infix function [], so that $[client] \stackrel{\text{def}}{=} c_{client}$ and $[server] \stackrel{\text{def}}{=} c_{server}$. The vector is initiated at each site with the value $\binom{0}{0}$. Sites count operations locally and increment their component in the local reference. The vector component of the remote site always contains the index of the most recent causally preceding operation known from the remote site. If the state references are equal, the operations must be equal.

Figure 5-7 depicts an example how state references are utilized. When the client creates the first operation O_1 no operation of the server has been received, so the client annotates O_1 with $\binom{1}{0}$. The first operation of the server O_3 is created in causal dependence to O_1. This dependence is reflected in the client component of the vector, which has the value 1. Additionally the counter for local operations at the server is increased by 1 and the value reflected in the state reference of O_3, which is now $\binom{1}{1}$. O_3 is received at the client after the client sends its second operation O_2. Accordingly O_3 does not reflect O_2 in its state reference, the respective vector component still has value 0.

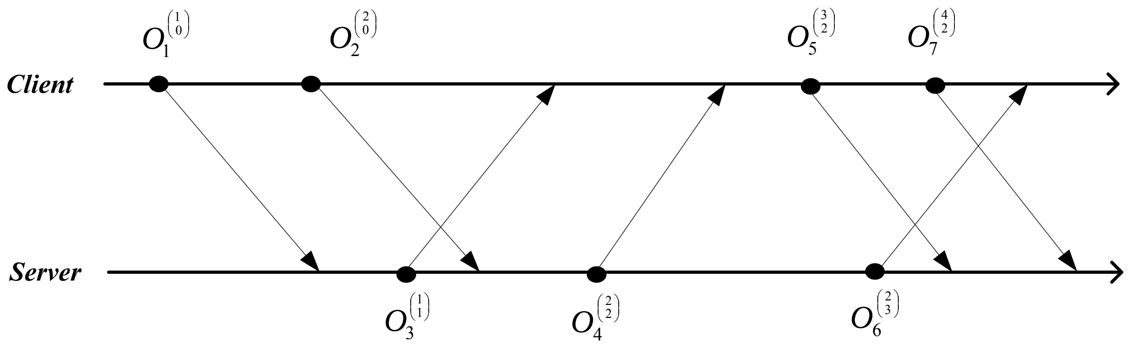


Figure 5-7: Example of handling state reference vectors

In practice, the state reference can be derived from the interval of the message an operation is shipped with and the number of this operation in the sequence of operations shipped with the same message. However, to simplify the following definitions, the reference is considered explicit. With the reference as ordering criterion, the operation can be defined formally.

Definition 5-2: Operation

> An operation is data structure of the form *Operation* := <*f*, *p*> where *f* is a the function with the signature *f*: *Parameter* × S^* → S^* to be applied to the state $S \in S^*$ and $p \in$ *Parameter* is the vector of parameters for the application of *f* to a state *S*. S^* is the superset of all reachable states.

The parameters of each operation should refer to a continuous namespace. For convenience, in the remainder of this work the abbreviated notation $f(p_1, p_2, \ldots, p_n)$ is still used whenever the operation instance $Operation(f, (p_1, p_2, \ldots, p_n))$ is meant, where $(p_1, p_2, \ldots, p_n)$is the component notation of *p*. Additionally, the helper function *reference*: *Operation*→ *Reference* is introduced, which returns a state reference for an operation. With the help of this notation, the server component of the state reference for an operation O_x is addressed with the notation *reference*(O_x)[*server*]. The client component of the state reference is addressed with *reference*(O_x)[*client*]. Causal precedence and causal concurrency in the system are formally defined in the following.

Definition 5-3: Precedence

An operation O_x precedes an operation O_y denoted $O_x \rightarrow O_y$ if all components in the state reference of O_x are less than or equal to their respective components in the state reference of O_y.

$$O_x \rightarrow O_y \equiv reference(O_x) \neq reference(O_y) \wedge \forall s \in \{client, server\}\, reference(O_x)[s] \leq reference(O_y)[s]$$

Definition 5-4: Concurrence

Two operations O_x and O_y are concurrent, denoted $O_x // O_y$, if neither O_x precedes O_y nor O_y precedes O_x.

$$O_x \parallel O_y \equiv reference(O_x) \neq reference(O_y) \wedge O_x \nrightarrow O_y \wedge O_y \nrightarrow O_x)$$

The definitions allow server and client to restore the partial order of operations. For the example given in Figure 5-7 the partial ordering of operations is $O_1 \rightarrow (O_2 \parallel O_3) \rightarrow O_4 \rightarrow ((O_5 \rightarrow O_7) \parallel O_6)$. To conclude with the same state, both sites need to resolve the partial order to a total order. Utilizing a priority rule for the order of applying concurrent operations, such as "server operations first", enables both sites to conclude with the same total order for applying the operations.

5.4.2 Preservation of Operation Semantic

The conclusion of a total order for all operations in the system is a necessary condition for state convergence. A convergent state, however, is not necessarily experienced as correct state by the user. Especially the prioritization of concurrent operations has a significant impact on the correspondence of the intended and actual states. Two exemplary system configurations and possible operation sequences are discussed in detail in the following.

Example1: The first example configuration is already used above. The state comprises a set of labeled numeric variables with initial values S = {a, b | a=2, b=4}. The functions for utilization in operations are *add* and *mul*, to either add a numeric value to a variable in the state or multiply a variable in the state by a numeric value. The two operations O_1 and O_2 are exchanged between clients and server as depicted in Figure 5-8 The server forwards the operations as O_1' and O_2'. If O_1 || O_2, the operation priority is decided based in the site index, so operations of Client2 are executed first.

			$O_1 \rightarrow O_2$ (Figure 5-8a)			$O_1 \parallel O_2$ (Figure 5-8b)		
Case	O_1	O_2	S_{Lead}	S_{Copy1}	S_{Copy2}	S_{Lead}	S_{Copy1}	S_{Copy2}
A	add(a,2)	add(a,3)	a=7, b=4	a=7, b=4	a=7, b=4	a=7, b=4	a=7, b=4	a=7, b=4
B	add(a,2)	mul(b,3)	a=4, b=12	a=4, b=12	a=4, b=12	a=4, b=12	a=4, b=12	a=4, b=12
C	mul(a,2)	mul(a,3)	a=12, b=4	a=12, b=4	a=12, b=4	a=12, b=4	a=12, b=4	a=12, b=4
D	mul(a,2)	add(a,3)	a=7, b= 4	a=7, b= 4	a=7, b= 4	a=10, b= 4	a=10, b= 4	a=10, b= 4

Table 5-2: Operation allocation for example 1 in operation mode (1)

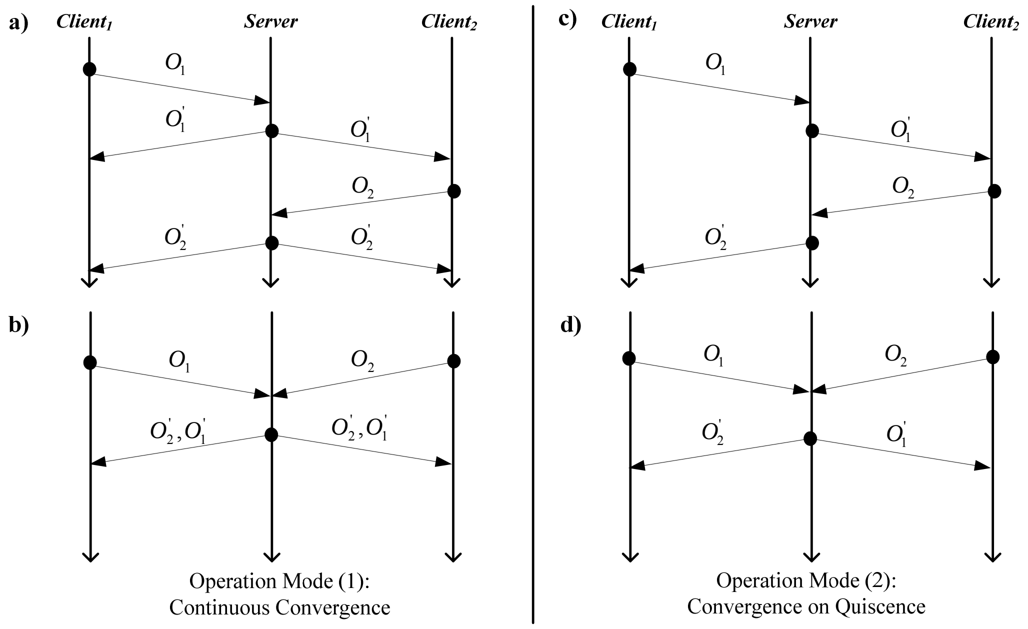

a) Client1 Server Client2
O1
O'1
O'1
O2
O'2
O'2
b)
O1
O2
O'2,O'1
O'2,O'1
Operation Mode (1): Continuous Convergence
c) Client1 Server Client2
O1
O'1
O2
O'2
d)
O1
O2
O'2
O'1
Operation Mode (2): Convergence on Quiscence

Figure 5-8: Operation ordering

Table 5-2 summarizes four cases for the allocation of the two operations in operation mode (1). S_{Copy1} and S_{Copy2} refer to the state at *Client₁* and *Client₂* respectively. If $O_1 \rightarrow O_2$ the user at client1 is aware of O_1 when O_2 is created. This awareness is reflected in the decision for the parameterization of O_2. The state convergences at all sites and the intentions of users are respected. Even if $O_1 \parallel O_2$, the state converges. Nonetheless the question remains for Case D of Table 5-2, whether 7 as final value for *a* is closer to the original user intentions than the actual value 10. Instead of using a static property for operation prioritization, the priority rule could respect operation semantic, such as operation with *mul* function first, or balance decisions fairly over the interacting users. A fair balance of priority decisions across all clients is especially of interest if the state change is absolute instead of relative. For example, a *set* function which sets a variable in the state to a given value would necessarily result in a final state that does not reflect the intentions of all users.

			$O_1 \rightarrow O_2$ (Figure 5-8c)			$O_1 \parallel O_2$ (Figure 5-8d)		
Case	O_1	O_2	S_{Lead}	S_{Copy1}	S_{Copy2}	S_{Lead}	S_{Copy1}	S_{Copy2}
A	add(a,2)	add(a,3)	a=7, b=4	a=7, b=4	a=7, b=4	a=7, b=4	a=7, b=4	a=7, b=4
B	add(a,2)	mul(b,3)	a=4, b=12	a=4, b=12	a=4, b=12	a=4, b=12	a=4, b=12	a=4, b=12
C	mul(a,2)	mul(a,3)	a=12, b=4	a=12, b=4	a=12, b=4	a=12, b=4	a=12, b=4	a=12, b=4
D	mul(a,2)	add(a,3)	a=10, b= 4	a=10, b= 4	a=10, b= 4	**a=10**, b= 4	**a=7**, b= 4	**a=10**, b= 4

Table 5-3: Operation allocation for example 1 in operation mode (2)

Table 5-3 summarizes four cases for the allocation of the two operations in operation mode (2). If $O_1 \rightarrow O_2$ the user at client1 is again aware of O_1 when O_2 is created and reflects this awareness in O_2. If $O_1 \parallel O_2$ the clients apply their own operation before sending them to the server. In case D of Table 5-3 this proceeding results in a conflict. S_{Copy1} is divergent to S_{Lead} and S_{Copy2}. *Client₁* would be required to apply O_2 after rolling back O_1 for this purpose. Alternatively, convergence could be achieved in case D if *O_2'* reflected the context change of *O_1*, i.e. the duplication of *a*, which would result in *O_2'* = add(a,6) rather than in *O_2'* = add(a,3). A change of the priority rule would only move the problem from *$Client_1$* to *$Client_2$*. The conflict does not arise in cases A-C. In these cases the operations either refer to different variables in the state or the application order of operations does not matter since the functions are commutative.

Example 2: The second example configuration is a state is a set comprised by strings S = {v, w | v= "xb", w = "xz"}. The function *ins* is introduced to insert a character at a given position in a string. The first position in a string is 0. So, ins(v, 1, "b") inserts "b" at the second position of v, which results in v="abcd". Again the two operations *O_1* and *O_2* are considered in the constellations of Figure 5-8. The allocations for the operations are summarized in Table 5-4 and Table 5-5.

			$O_1 \rightarrow O_2$ (Figure 5-8a)			*$O_1 \| O_2$* (Figure 5-8b)		
Case	*O_1*	*O_2*	*S_{Lead}*	*S_{Copy1}*	*S_{Copy2}*	*S_{Lead}*	*S_{Copy1}*	*S_{Copy2}*
A	ins(v,0, "a")	ins(v,0,"a")	v="aaxb" w="xz"	v="aaxb" w="xz"	v="aaxb" w="xz"	v="aaxb" w="xz"	v="aaxb" w="xz"	v="aaxb" w="xz"
B	ins(v,0, "a")	ins(w,1,"y")	v="axb" w="xyz"	v="axb" w="xyz"	v="axb" w="xyz"	v="axb" w="xyz"	v="axb" w="xyz"	v="axb" w="xyz"
C	ins(v,2, "e")	ins(v, 0,"a")	v="axbe" w="xz"	v="axbe" w="xz"	v="axbe" w="xz"	v="axeb" w="xz"	v="axeb" w="xz"	v="axeb" w="xz"

Table 5-4: Operation allocation for example 2 in operation mode (1)

In Table 5-4 the state converges for all cases in operation mode (1) if *$O_1 \rightarrow O_2$*. The user at *$Client_2$* is aware of *O_1* and thus *O_2* reflects the effect of *O_1* to *$S_{client2}$*. If *$O_1 \| O_2$*, again the state converges, but Case C unveils a problem with the final state. Caused by the priority rule *O_2* is applied first and changes the value of *v* so that the effect of *O_1* in the new value of v is different. The character "e" is inserted before the "b" and not behind as intended. If *v* is document in a collaborative text editor, the user who issued *O_1* would experience the value of *v* as incorrect. To preserve the semantic equivalence of *O_1* and *O_1'* the server would need to modify *O_1'* to reflect the change of *O_2* to the state. The value of the forwarded operation should be *O_1'* = ins(v, 3,"e"). Changing the priority rule would only move the problem from one site to the other.

			$O_1 \rightarrow O_2$ (Figure 5-8c)			*$O_1 \| O_2$* (Figure 5-8d)		
Case	*O_1*	*O_2*	*S_{Lead}*	*S_{Copy1}*	*S_{Copy2}*	*S_{Lead}*	*S_{Copy1}*	*S_{Copy2}*
A	ins(v,0, "a")	ins(v,0,"a")	v="aaxb" w="xz"	v="aaxb" w="xz"	v="aaxb" w="xz"	v="aaxb" w="xz"	v="aaxb" w="xz"	v="aaxb" w="xz"
B	ins(v,0, "a")	ins(w,1,"y")	v="axb" w="xyz"	v="axb" w="xyz"	v="axb" w="xyz"	v="axb" w="xyz"	v="axb" w="xyz"	v="axb" w="xyz"
C	ins(v,2, "e")	ins(v, 0,"a")	v="axbe" w="xz"	v="axbe" w="xz"	v="axbe" w="xz"	**v="axeb"** w="xz"	**v="axbe"** w="xz"	**v="axeb"** w="xz"

Table 5-5: Operation allocation for example 2 in operation mode (2)

Although in Case A the operations refer to the same variable in the state, the final state correctly reflects the semantic of both operations. The parameterization of O_1 and O_2 in Case A makes the operations commutable, although function *ins* is not commutative. In Case B the operations are isolated since they refer to different variables in the state.

In Table 5-5 the state converges for all cases in operation mode (2) if $O_1 \rightarrow O_2$. Also the semantics of operations is preserved; for all operations the context is not changed between the moment it was issued and the moment it is applied. If $O_1 \| O_2$, there is again a problem if O_1 and O_2 refer to the same variable in the state while not being commutable. *Client₁* applies O_1 to the local state before sending it to the server. Accordingly "e" is positioned behind the "b" in *v* and not before it as for $Client_2$ and the *Server*. A rollback of O_1 at $Client_1$ is no option, because the semantic of O_1 reflects the intention of the user at $Client_1$. The only alternative to achieve convergence is to ensure that the context change caused by O_2 at server and $Client_2$ is reflected in O_1' before applying it at the *Server* and forwarding the operation to $Client_2$. The operation should thus be O_1' = ins(v,3,"e"). Case A and B are unproblematic for the same reasons as in operation mode (1).

Both examples discuss the handling of operations from different clients at the server. However, in operation mode (2) also the clients have to resolve the semantics for concurrent operations. A forwarded operation received at the client can be concurrent to an operation issued and applied at this client since the last forwarded operation was received. The cases of the above examples do not differ in this situation and are therefore not explicitly discussed here.

The examples illustrate that semantic violation between two operations O_1 and O_2 are caused if $O_1 \| O_2$ and if O_1 and O_2 are not commutable in the order of application without modification. This observation is used to define the competition of two operations as follows:

Definition 5-5: Competition

Two operations O_x and O_y are said to be competing, denoted $O_x \# O_y$, if these operations are concurrent and if their order of application to any state S ∈ S* results in different final states.

$$O_\mathrm{x} \# O_\mathrm{y} \equiv \exists_{S \in S^*} O_\mathrm{x} \parallel O_\mathrm{y} \wedge apply\left(O_\mathrm{x}, apply(O_\mathrm{y}, S)\right) \neq apply\left(O_\mathrm{y}, apply(O_\mathrm{x}, S)\right)$$

The function *apply* applies an operation to a given state *S*. The function has the signature $apply\colon operation \times S^* \rightarrow S^*$ and is defined with $apply(O, S) \overset{\mathrm{def}}{=} function(O)(parameters(O), S)$.

As state of the art, different mechanisms for handling competing operations are discussed. In addition to pessimistic mechanisms such as locking, optimistic mechanisms are introduced such as rollback, patching and inclusion. Inclusion mechanisms have the advantage that they fulfill the instantaneity requirement on the one hand, while on the other hand they do not require the rollback of operations whose effects were already presented to the user. The objective of inclusion mechanisms is to achieve commutability of two competing operations by transformation.

Definition 5-6: Inclusion Transformation

The transformation function $transform: operation \times operation \rightarrow operation$ includes the effect of an operation $O_y \in operation$ into a competing operation $O_x \in operation$ written $O_x' = transform(O_x, O_y)$. Function and parameters of the operation O_x are changed relative to function and parameters of O_y, so that the effect of applying O_x in S is the same as in S' with $S' = apply(O_y, S)$. The transformation function must not require modifying operation function and parameters relative to the current state S. The transformation function is defined in accordance to the system configuration, i.e. state representation and operation functions, but has to satisfy the following property:

$$apply\left(transform(O_x, O_y), apply(O_y, S)\right) = apply\left(transform(O_y, O_x), apply(O_x, S)\right)$$

Precondition (1): $\nexists_{O_z \in operation} O_z \rightarrow O_y \wedge O_x \# O_z$

Precondition (2): $O_x$are $O_y$context equivalent

The property of the inclusion transformation corresponds to the *TPI* defined in [62]. The property ensures that the execution of operations in a different order results in the same final state. Otherwise a finally transformed operation could result in an unexpected effect. The preconditions eliminate the need for *TPII*. Precondition (1) ensures operations are transformed against the correct order of concurrent operations. For this reason, operations are sequentially exchanged between client and server, no operation O_z with $O_x \parallel (O_z \rightarrow O_y)$ is hidden from either site.

In addition to Precondition (1), context equivalence of the two operations O_x and O_y is enforced by Precondition (2). This means, that for any three operations $O_x \# O_y \# O_z$ that arose from the same state, the transformation $transform(transform(O_x, O_z), O_y)$ is invalid. The result of transforming O_x against O_z appears like an operation that arose from a state, in which O_z was already applied. Even though this transformed operation is practically still competing with O_y, the result of a transformation against O_y does not have the same effect as $transform(O_x, O_y)$. The correct transformation would be $transform(transform(O_x, O_z), transform(O_y, O_z))$. A similar situation is given if $(O_x \rightarrow O_y) \parallel O_z$, where $O_x \# O_z$ and $O_y \# O_z$. In this case the transformation of O_y after O_x requires a version of O_z that also reflects the effect of O_x. The correct transformation of O_y would be $transform(O_y, transform(O_z, O_x))$.

Definition 5-7: Context Equivalence

> Two competing operations O_x and O_y are context equivalent if they arose from the same state or reflect the same history of operations since that state (by transformation or direct causal dependence).

An inclusion function for example 1 could be defined as follows:

$$include_{math}\left(\left(f_1, (p_{1,1}, p_{1,2})\right), \left(f_2, (p_{2,1}, p_{2,2})\right)\right)$$
$$\stackrel{\text{def}}{=} \begin{cases} \left(mul, (p_{1,1}, p_{1,2} \times p_{2,2})\right), if\ f_1 = mul \wedge f_2 = mul \wedge p_{1,1} = p_{2,1} \\ \left(add, (p_{1,1}, p_{1,2} + p_{2,2})\right), if\ f_1 = add \wedge f_2 = add \wedge p_{1,1} = p_{2,1} \\ \left(mul, (p_{1,1}, p_{1,2})\right), if\ f_1 = mul \wedge f_2 = add \wedge p_{1,1} = p_{2,1} \\ \left(add, (p_{1,1}, p_{1,2} \times p_{2,2})\right), if\ f_1 = add \wedge f_2 = mul \wedge p_{1,1} = p_{2,1} \\ \left(f_1, (p_{1,1}, p_{1,2})\right), else \end{cases}$$

This inclusion function satisfies the conditions of the transformation function, but does not respect the operator precedence rules for the prioritization of the mathematical operations *mul* and *add.* If operations with *mul* should have higher priority in the inclusion function, a new operation function would be required in the system that first rolls back the addition (i.e. subtracts the previously added value), multiplies the referenced value in the state and adds the subtracted value again. This procedure is required since the inclusion transformation must not refer to the current state *S* of the system.

A possible inclusion function for example 2 could be defined as follows:

$$include_{text}\left(\left(f_1, (p_{1,1}, p_{1,2}, p_{1,3})\right), \left(f_2, (p_{2,1}, p_{2,2}, p_{2,3})\right)\right)$$
$$\stackrel{\text{def}}{=} \begin{cases} \left(f_1, (p_{1,1}, p_{1,2} + 1, p_{1,3})\right), if\ p_{2,1} < p_{1,1} \vee p_{2,1} = p_{1,1} \wedge p_{2,3} < p_{1,3} \\ \left(f_1, (p_{1,1}, p_{1,2}, p_{1,3})\right), else \end{cases}$$

This inclusion function simply increases the offset for inserting the character in the targeted string if the other operation added a character at a lower offset of the string before. If both operations refer to the same position in the string, the values of the characters to be inserted are compared. Characters with a lower value are always written first.

The second inclusion function reveals a problem with regard to fairness of tie breaking. If the intentions of two operations cannot be combined as in the example, a fair tie breaking solution has to be provided. Fair would mean either to disable both operations or to ensure an equal distribution of success between the operation originators if tie breaking situations occur recurrently. Disabling operations is the simpler approach, but means that the application state does not advance. This can provoke the same tie breaking situation. For an equal distribution of success between operation originators a random operation priority would be required. The physical time of operation origination is inappropriate in this regard. Network delays affect the time operations are transformed. Since delays mainly depend on the physical distance between two machines, the distance between client sites and server site would determine whether and operation originator loses or wins conflicts. The equal distribution of success between operation originators has to be enforced. Therefore, the session state is exposed to the inclusion function in the form of the bucket interval index later.

5.5 Operation Mode Determination

The decision for the introduction of two operation modes follows the study of use cases for application on the web and the clear distinction of requirements for competitive and collaborative user interactions. While in competitive use cases, simultaneity is higher order magnitude than instantaneity, in collaborative use cases instantaneity is higher order magnitude than simultaneity. Nonetheless, collaborative and competitive use cases do not appear in isolation. Instead of dedicated sessions, each with its own operation mode, sessions with a mix of both operation modes are required. To determine whether a local operation is allowed to be applied instantly or not, a global schema is introduced for operations.

Definition 5-8: Operation Schema

> The schema is composed by the two functions $synchronous, instant: Operation \rightarrow \{T, F\}$ that determine for any operation $O_x \in Operation$ if O_x is applied in operation mode (1) or (2). If and only if $synchronous(O_x)$ is true the operation has to be treated in accordance to operation mode (1). If and only if $instant(O_x)$ is true the operation has to be treated in accordance to operation mode (2). The functions have the same domain of definition and must satisfy the following properties:
>
> i) $\forall_{O \in Operation}\big(instant(O) \vee\ synchronous(O)\big) \wedge (\ instant(O) \neq synchronous(O))$
> ii) $\forall_{O_x \in Operation} \nexists_{O_y \in Operation}\ synchronous(O_x) \wedge instant(O_y) \wedge\ O_x \# O_y$
> iii) $\forall_{O_x \in Operation} instant(O_x)\ \nexists_{O_y \in Operation}\ synchronous\big(transform(O_x, O_y)\big)$

Property i) ensures an operation is either to be applied in operation mode (1) or (2), but never applicable in both at the same time. Property ii) states that any two competing operations cannot be classified in different operation modes. This condition is required to ensure convergence. An approach to achieve property ii) is to classify the entities of the state by operation mode and let the operation inherit the mode of the entity its parameters refer to. Property iii) requires the transformation not to change the schema of an operation. The schema bases on the determination that operation mode (1), continuous convergence, is the default operation mode. The schema functions have the same domain definition, so in principle one function is sufficient to determine both cases.

At this point all customizable system characteristics have been introduced. The following definition summarizes the basic System Configuration:

Definition 5-9: System Configuration

The System Configuration is a data structure containing all customizable system properties having the form $Configuration = \langle S^*, f^*, instant, include, T, a \rangle$ where:

S^*	is the set of all reachable states
f^*	is the set of all operation functions supported for transforming the state
instant	is the schema function that determines the mode for any operation
include	is the inclusion function required for operation transformation
T	is the initial length of an interval in operation pipelining
a	is the number of pipelines

5.6 Control Algorithm

The integration of communication scheme, state copies convergence and operation mode determination is defined by a control algorithm, distributed between client and server. WHAs base on a thin client application model. The control algorithm is thus designed to handle as much processing as possible at the server. The algorithm supports any system configuration and is thus applicable for all state representations and operation functions, as long as the corresponding schema and inclusion function conform with the above defined properties.

The communication scheme requires all clients to collect operations for an interval and send them to the server at once. The server thus has to transform *n* lists of operations per interval, where *n* is the number of clients. While the first operations of each list are context equivalent and thus transformable against each other, the remaining operations in the list are not. The correct transformation of the operations O_1 and O_2 against O_3, where $(O_1 \rightarrow O_2) \parallel O_3$, requires to transform O_2 against *transform*(O_3, O_1) as shown in [62]. For the transformation of any two lists of concurrent operations, Algorithm 5-1 is defined. This makes use of the transformation function *transform*$_i$ defined in Definition 7. The index *i* refers to the current bucket interval of the pipelining approach. For each bucket interval a new instance of the transformation function is created from the interval index. This mechanism allows binding the tie-breaking rules of the transformation function to the session state.

```
transformlist: $\text{List}_{\text{Operation}} \times \text{List}_{\text{Operation}} \rightarrow \text{List}_{\text{Operation}}$
transformlist (L1, L2) = L where
L = L1
for i = 0 to |L|
 for j = 0 to |L2|
  if L[i] # L2[j]
   L[i] = $\text{transform}_i$(L[i], L2[j])
   L2[j] = $\text{transform}_i$(L2[j], L1[i])
  endif
```

```
10      endfor
11    endfor
```

Algorithm 5-1: Transformation of operation lists

The algorithm makes use of the data structure *List*. *List* is an ordered set of elements and comes with the function *push*, *peek*, *concatenate* and **| |**, the operator [ ] and the constructor function *List*. The *push* function adds an element to the end of a list. The *peek* function retrieves the last element of the list without removing it. The function *concatenate* appends all items of one list to another. The **| |** function returns the number of elements in a list. The [ ] operator refers to an element in a list by its position, where the first element has position 0. For the instance of *List* *example* = *a*, *4*, *x*, *c* the notation *example*[*2*] refers to *x*, **|***example***|** is *4*, *push*(*example, 9*) results in *a*, *4*, *x*, *c*, *9*, *peek*(*example)*would then result in *9*, *concatenate(example, example)* results in *a*, *4*, *x*, *c*, *a*, *4*, *x*, *c*.

The order of transforming the lists of operations against each other directly affects the convergence of states at the clients as discussed in chapter 4. To be able to abstain from TP2, one central order for the operations from all clients has to be determined during transformation, the common transformation path. Before, context equivalence for any two lists of operations has to be achieved before transformation. The algorithm is explained by the following example.

The server receives four concurrent operation lists *A*, *B*, *C* and *D*, which only contain operations already applied at the clients. How the server decides for the order of the lists does not matter. For a better readability alphabetic order is taken in the example. To additionally shorten the formulas the abbreviated notations A^B = *transformlist(A,B)* and A^{CB} = *transformlist(*A^B*,C)* are introduced. In the end each client requires properly transformed versions of the operation lists of the other clients. It would be incorrect to simply send B^A, C^{BA} and D^{CBA} to the client who sent *A*, because C^A and *B* are not context equivalent and thus cannot be transformed against each other. Instead C^A has to be transformed against B^A, which results in $C^{B^A A}$. Also *D* has to be transformed against the transformed versions of *B* and *C*. However, the resulting list only works for the client who sent A. According to the argumentation above the client who sent *D* has to receive *A*, B^A and $C^{B^A A}$, but these lists do not yet reflect *D*, so they all additionally have to be transformed against their context equivalent counterparts of *D*. The correct operation lists the client who sent *D* should receive is A^D, $B^{D^A A}$ and $C^{D^{B^A A} B^A A}$.

A	A^B	A^C	A^D
B	B^A	$B^{C^A A}$	$B^{D^A A}$
C	C^A	$C^{B^A A}$	$C^{D^{B^A A} B^A A}$
D	D^A	$D^{B^A A}$	$D^{C^{B^A A} B^A A}$

Table 5-6: Exemplary transformation matrix

The lists the clients who sent B and C should receive have to be created in a way which combines the approaches for the cases A and D. Table 5-6 shows the complete compilation of required transformation results in the example. The operation lists for each client depend on the index of a client's operation list in the preselected order of the server. For example, if a client sent B, the server takes all lists from column B followed by the lists on the diagonal behind the component (B,B). The respective cells are shaded in Table 5-6. The generalized version of this matrix is referred to as *transformation matrix* in the following and produced with the help of Algorithm 5-2.

The algorithm introduces the data structure *Matrix* with the constructor function *Matrix:integer→Matrix,* and the operator [ ]. The constructor function *Matrix(n)* creates an empty matrix with $n \times n$ components. The operator [x, y] refers to the component in row x and column y of the matrix.

```
intervaltransform: $\text{List}_{\text{List}_{\text{Operation}}} \rightarrow \text{Matrix}_{\text{List}_{\text{Operation}}}$
intervaltransform (L) = M where
M = Matrix<|L|>
for i = 0 to |L|
 if i > 0 then M[0,i] =  transformlist(L[0], L[i])
 M[i,0] =  L[i]
endfor
for i = 1 to |L|
 for j = 1 to |L|
  if i=j then
   M[i,j] = transformlist(M[i,j-1], M[i-1,i-1])
  elseif (j > i) then
   M[j,i] = transformlist(M[j,i-1], M[i-1,i-1])
   M[i,j] = transformlist(M[i,i], M[j,i])
  endif
 endfor
endfor
```

Algorithm 5-2: Creation of the transformation matrix

The retrieval of operation lists to be sent to the clients is described by Algorithm 5-3. The algorithm first inspects the columns from the first row to the diagonal and afterwards follows the diagonal to obtain the lists for each client. In contrast to a pure optimistic solution, the algorithm also respects pessimistic operations, i.e. operations created in mode (1). Those are selected from the diagonal component in the matrix by the helper function $select: List_{Operation} \times Condition \rightarrow List_{Operation}$ in line 9. The set $Condition$ refers to any condition on the elements in the list. The *select* function returns elements according to the order of the original list.

```
extract: $\text{Matrix}_{\text{List}_{\text{Operation}}} \rightarrow \text{List}_{\text{List}_{\text{Operation}}}$
extract (M) = L where
L = List()
for i = 0 to |M|
  $l_{tmp}$ = List()
  for j = 0 to i
    $l_{tmp}$ = concatenate($l_{tmp}$, M[j, i])
  endfor
  $l_{tmp}$ =concatenate($l_{tmp}$, select(M[i, i], x with synchronous(x)))
  for j = i+1 to |M|
    $l_{tmp}$ = concatenate($l_{tmp}$, M[j, j])
  endfor
  push(L, $l_{tmp}$)
endfor
```

Algorithm 5-3: Creation of operation lists resulting from transformation

Server and client process operations in accordance with the pipeline scheme. Before an operation sent from client to server is processed at the server and forwarded for application in the target interval, the client can receive multiple messages with concurrent operations. The server faces the same situation, i.e. operations received from the client still concur with operations recently sent to this client. Accordingly both first have to update the incoming operations with regard to recently sent concurrent operations. Recently sent operations are stored continuously in a history buffer. Operations are deleted from the history buffer after a complete roundtrip of operations. The update function is described in Algorithm 5-4. The precondition of the algorithm is that all operations in the non-empty list of incoming operations L share the same value for the *reference* component s. That means, for example, all operations received from a client should point to the same last server operation as causal context.

```
update: $\text{List}_{\text{Operation}} \times \text{List}_{\text{Operation}} \times \{client, server\} \rightarrow \text{List}_{\text{Operation}}$
update(L,H,s) = P
P= List()
$h_{tmp}$ = select(H, x with reference(x)[s] > reference(L[0])[s])
P = concatenate(P, transformlist(L, $h_{tmp}$))
```

Algorithm 5-4: Updating operations for transformation in the current state

The control algorithm for the server i) updates the operations from all clients, ii) creates the transformation matrix, and iii) retrieves the operation lists to be sent back to the client. Since the state reference of each operation is limited to one client, the server has to overwrite the state references for outgoing operations. The control algorithm at the client is limited to updating incoming operations from the server with respect to locally created, concurrent operations.

The functions *intervaltransform* and *transformlist* mostly account for time complexity of the transformation algorithm. The time complexity of *intervaltransform* depends on the number of request messages per interval $n=|L|$ and the time complexity of *transformlist.* The interval transformation fills the $n \times n$ transformation matrix with results of list transformations. The algorithm initializes the first row of the matrix while skipping the first element. Hence, it performs $n-1$ list transformations. Afterwards it initializes the diagonal and again skips the first element, i.e. $n-1$ list transformations. In parallel to filling the diagonal from the left-upper to the right-lower corner, the algorithm fills the two triangles below and above the diagonal. Since the first row and the first column of the matrix are already initialized, only $(n-1)^2$ fields minus the diagonal $n-1$ have to be initialized. These steps result in the following complexity calculation: $n-1+n-1+(n-1)^2-(n-1)=n^2-n$. Accordingly the time complexity of the interval transformation algorithm can be estimated with $O(n^2)$. The time complexity of *transformlist* depends on the length of the operations lists to transform against each other. If the first list is of length k and the second list of length l, the algorithm performs $2kl$ inclusions. The time complexity can thus be estimated with $2max(k,l)^2$, which is also equivalent to $O(n^2)$.

6 Instant State Synchronization System

Chapter 5 describes communication scheme and algorithms in a theoretical context. This chapter adds the recommended system architecture for utilizing communication scheme and algorithms in a practical context.

6.1 Overview

The Instant State Synchronization System (IS3) is an abstract realization of the communication schema and the algorithms for real-time state synchronization introduced in chapter 5. The system design puts forward the principle of a generic state representation and respective operations to modify this state. It refines the communication scheme by naming the involved components and defining the interfaces between them. IS3 extends the legacy Web architecture by introducing new client and server components, as well as interfaces to connect to legacy components. The maintenance of the shared state in IS3 is positioned orthogonally to the continuation-based interactions of *WHA Client* and *WHA Server*. Nonetheless, the state is only kept as long as a WHA session continues.

A basic system overview is depicted in Figure 6-1. The *IS3 Client* realizes the client algorithms and the link to the *WHA Client*. The *IS3 Server* maintains the lead state and dispatches state synchronizing operations to all clients in order to achieve state convergence. Additionally the IS3 Server provides an interface to the *WHA Server* to enable data access, session control and client authentication. IS3 Server and Client are connected by a generic network interface which reflects the communication scheme introduced in section 5.3.

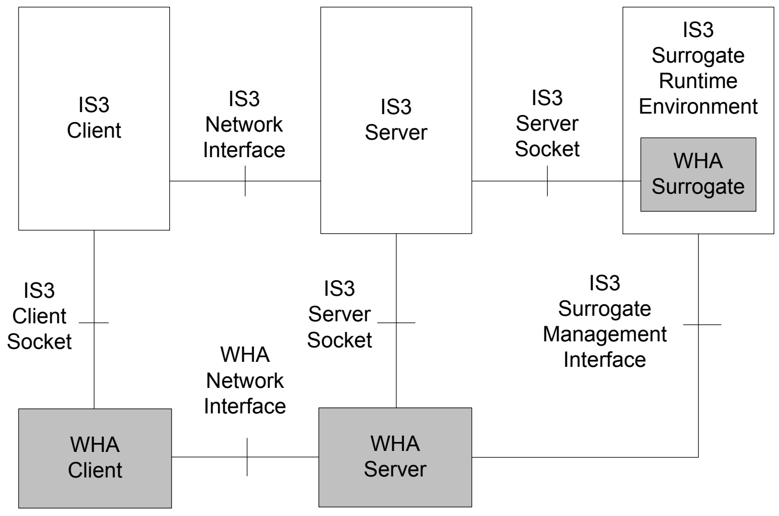


Figure 6-1: Basic System Overview

If WHA Server and IS3 Server are physically separated and the geographic distance negatively affects the latency of communication among both, a *WHA Surrogate* can be used to substitute the direct access of WHA Server to IS3 Server. A WHA Surrogate is an active component, implementing the essential WHA logic for IS3. Although not illustrated in Figure 6-1, WHA Surrogate and WHA Server can also maintain a proprietary communication channel. For safe execution of WHA Surrogates, the *IS3 Surrogate Runtime Environment (SRE)* is introduced. The WHA Server utilizes the *IS3 Surrogate Management Interface* to deploy WHA Surrogates and control their life-cycle.

In the following sections the basic profile for the interfaces of IS3 is explained in detail. The interface models are therefore described abstractly, while not respecting peculiarities of technologies that may be used for a later implementation.

6.2 Scopes

The consistency/latency trade-off says: the amount of data that needs to be communicated to achieve state convergence is directly proportional to the transmission time. For the system proposed here this consequently means there are a finite number of operations for each client that can be transmitted within the time Δ introduced in section 5.3 without making the operation late. The number of operations depends on the application, but most likely increases with the number of clients. If the application does not require all clients to share the entire state at the same time, a kind of state masking is useful as shown by [56]. For this reason, the network protocol here includes support for scopes to mask the shared state and communicate operations that affect the state within currently relevant scope only.

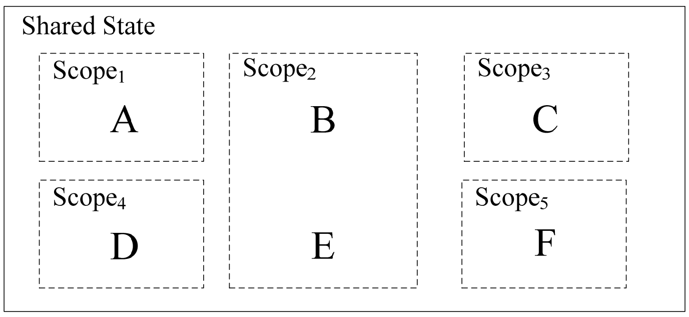


Figure 6-2: State and Scopes

A scope refers to an addressable subset of the shared state. Any two scopes have to be disjoint. In Figure 6-2 a state comprised of the six variables A-F and five scopes is depicted. In practice, scopes can refer to the pages of a document or the rooms in a game level. For the document editor, only the currently viewed page may be relevant. For the gamer, perhaps only visible objects in the same room are interesting. There is thus no need to copy the entire state to all clients.

Clients subscribe for scopes depending on the application context (e.g. currently document page), the server makes sure only operations are delivered to client which refer to the currently relevant client scope. Respectively only subsets of the state at the client related to the client scopes are kept up-to-date. if for example a client is subscribed to $Scope_2$ of Figure 6-2 only, the client does not recognize changes to variables A,C,D and F. However, clients are allowed to subscribe to multiple and even all scopes. Changing the scopes requires the initial update of the state subset referred to by the new scope. Scopes are predefined or can be declare at runtime. The entire shared state has to be covered by scopes.

Efficient scoping requires a fine grained design of the state representation and respective operations to modify the state. If the shared state cannot be segmented in terms of masks, one global scope has to be defined.

6.3 Network Interface

For a straightforward realization as network protocol, the network interface is described on the basis of messages. The communication scheme describes request and update messages. Both message types are similarly structured in the sense that they contain session related information such as the interval and a number of operations. The establishment and maintenance of the session itself requires the exchange of a number of additional messages and related information. Figure 6-3 therefore depicts a generic message format that reflects the needs of communication scheme and session life-cycle management. The generic message contains one *Head* and zero or more *Parts*.

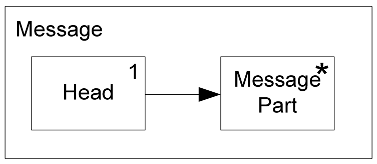


Figure 6-3: Generic message format

As depicted in Figure 6-4, the Head begins with a *Protocol Identifier* (format and syntax are to be defined on the level of protocol specification). The *Message Type* defines the meaning of the message and tells the receiver which kind of Data Units it can expect as message payload. Examples for message types are REQUEST and UPDATE.

The network interface is intended to support the multiple concurrent communication session on the same underlying communication channel. In this case the transport address (host address and process port) is not applicable for the unique identification of clients. In an IS3 session, a client is identified by an *Endpoint Identifier*. This identifier has to be unique for each IS3 Server. Clients must not join multiple sessions while using the same endpoint identifier. The endpoint identifier is selected by the client. To ensure the selection is unique, a client could for example combine physical address, location and time.

The field *Interval Number* contains the number of the interval in which the operations in the same message should take effect. This number was introduced as symbol *i* in section 5.3.1. The *Last Interval Number* contains the number of the last interval the message sending site received a message from the message receiving site. For example, if the last message the server received from a client contained the Interval Number 10, this number 10 has to be included in the field Last Interval Number of the next message the server sends to the client.

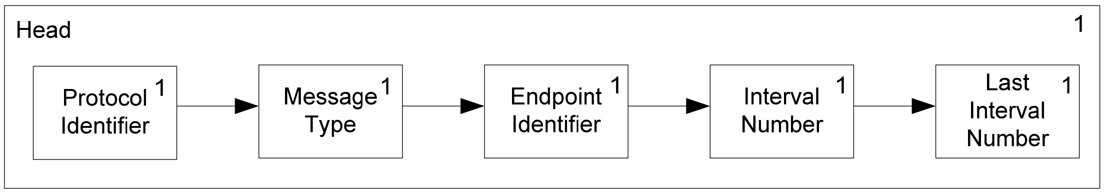


Figure 6-4: Fields in message Head

All fields of the Head are marked as mandatory in Figure 6-4. However, since the communication scheme requires that a client does not send a request before receiving an update, the Last Interval Number could be implemented as an optional field for messages sent from client to server. If, in contrast, clients estimate the time for sending the next message, e.g. based on the roundtrip of previous updates, the field is required.

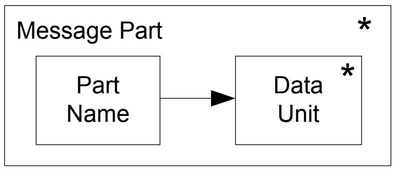


Figure 6-5: Fields of Message Part

The message payload is segmented into zero or more Message Parts. As shown in Figure 6-5 each Message Part has a *Part Name* and groups all *Data Units* that have to be processed together. The first part always contains all Data Units related to session maintenance. The remaining parts reflect the scopes in the shared state. The segmentation of messages into parts is a necessary preparation of an effective operation pipelining. Messages must not contain more than one Message Part for the same scope.

The format of a Data Unit is depicted in Figure 6-6. The Data Unit is a universal and partially customizable container for the communication of session parameters, state subsets and operations. The *Data Unit Type* determines the semantic of the Data Unit. The basic set of session related values for this field and their meaning is derived from the communication scheme and summarized in Table 6-10. Types that refer to operations have to be defined with regard to the application and are not part of the interface definition. The field *Data Unit Parameter* may appear zero or more times and realizes a list of parameters, which can be used to specify the meaning the Data Unit implied by its type in detail.

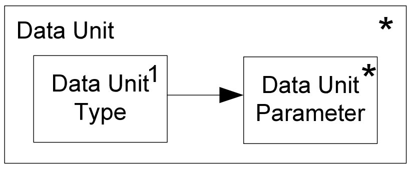


Figure 6-6: Fields in message Data Unit

The implementation of messages may vary depending on the underlying communication channel. A direct implementation on a transport protocol could favor a binary message representation. An implementation based on an upper-layer text-based protocol could require a message representation with markup languages. In either case the parallel transmission of messages must be supported. Otherwise large messages could block the communication channel and thus harm the timing of the communication scheme.

6.3.1 Session Participation

A session is initiated and terminated by the backend of a WHA through the server socket. A session is persistent for its lifetime. The presence or absence of clients does not affect the state of a session. Modifying the shared state requires clients to join a session and request changes by issuing operations. Figure 6-7 depicts the message flows on a client participation request in case of an error and in case of success.

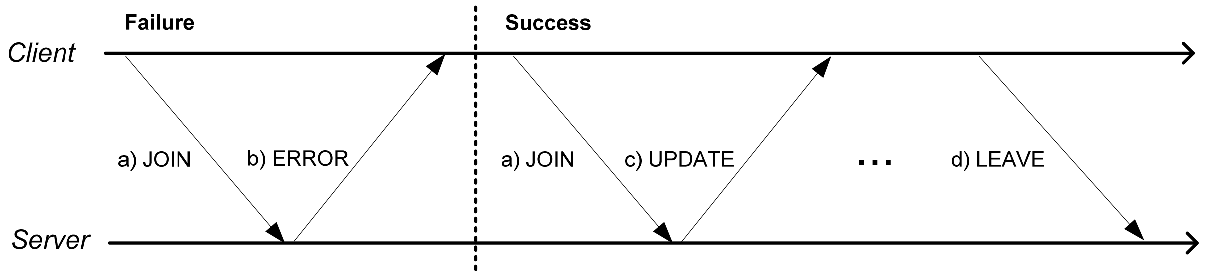


Figure 6-7: Message flow for session participation

The client requests the participation in a session by sending a message of type JOIN, referred to as message a) in Figure 6-7. A join message contains at least one Message Part with two Data Units as summarized in Table 6-1. The first Data Unit contains a reference to the session the clients intends to join. The second Data Unit names the scope the client subscribes for. Both types of Data Units are mandatory for a JOIN message. A JOIN message can also contain more than one scope subscription.

Constants are written in upper case in the table, variables in italics, custom constants in italics upper case. Fields which can be ignored are marked with a hyphen. The formatting remains the same for tables describing messages in the following sections.

Head	Protocol Identifier	-	
	Message Type	JOIN	
	Endpoint Identifier	*client-identifier*	
	Interval Number	-	
	Last Interval Number	-	
Message Part	Part Name	-	
	Data Unit	Data Unit Type	SESSION-REFERENCE
		Data Unit Parameter	*session-identifier*
	Data Unit	Data Unit Type	SCOPE-SUBSCRIBE
		Data Unit Parameter	*scope-identifier*

Table 6-1: Message a) - joining a session

The server can answer the client request to join a session either with a message of type ERROR or with a message of type UPDATE. An error can occur if a client message is not well formed or contains types of Data Units not allowed at this point of interaction. Another prevalent reason is that the server does not know session or scope the client refers to as illustrated for example in Table 6-2. If the server replies to a message of type JOIN with a message of type ERROR, the server considers the client attempt to join the session as failed. The client has thus to issue another message of type JOIN to reinitiate the procedure. Errors are categorized as critical or non-critical. Critical errors implicitly disconnect the client from the session, non-critical errors are supposed to be correctable without disconnection or only have an informatory character. Critical errors are always sent with a message of type ERROR. Non-critical errors can also appear in update messages.

Head	Protocol Identifier	-	
	Message Type	ERROR	
	Endpoint Identifier	*client-identifier*	
	Interval Number	-	
	Last Interval Number	-	
Message Part	Part Name	-	
	Data Unit	Data Unit Type	CRITICAL-ERROR
		Data Unit Parameter	SESSION-UNKNOWN
		Data Unit Parameter	*session-identifier*

Table 6-2: Message b) - error on joining a session

If the server accepts the client request to join a session, the server sends an UPDATE message to the client. The first update after a client has joined a session must contain at least two Message Parts. The first part has to contain session information, with at least a Data Unit containing the number of parallel pipelines. The Interval Number in the message head and the pipeline number allow the client to calculate the interval for its first request message. The second Message Part has the name of the subscribed scope and contains a copy of the state for the scope the client has subscribed for.

Head	Protocol Identifier	-	
	Message Type	UPDATE	
	Endpoint Identifier	*client-identifier*	
	Interval Number	*current-interval-number*	
	Last Interval Number	-	
Message Part	Part Name	-	
	Data Unit	Data Unit Type	PIPELINE-COUNT
		Data Unit Parameter	*number-of-pipelines*
	Data Unit	Data Unit Type	INSTANT-CHECK-FUNCTION
		Data Unit Parameter	*function-representation*
	Data Unit	Data Unit Type	REGISTER-OPERATION
		Data Unit Parameter	*function-name*
		Data Unit Parameter	*function-representation*
	Data Unit	Data Unit Type	INCLUSION-FUNCTION
		Data Unit Parameter	*function-representation*
Message Part	Part Name	*scope-identifier*	
	Data Unit	Data Unit Type	SCOPE-INIT
		Data Unit Parameter	*state-representation*

Table 6-3: Message c) - update on joining a session

An example is shown in Table 6-3. This additionally contains data units for the transport of a schema function, one operation function and the inclusion function to the client. Sessions are set up at the server, but if the client asks to execute operations in operation mode 1), the client has to be aware of the implementations for operation functions and the inclusion function. However, in an implementation of IS3 schema function, inclusion function and operation functions could also be initialized in a different way. The message can contain more than one Message Part to initialize a scoped state at the client, but it must not contain any operations.

To leave a session, a client sends a LEAVE message to the server. Since the Endpoint Identifier is tied to a session, the server knows from which session the client has to be disconnected. No Message Parts are required. Messages of type LEAVE must not contain any operation. An example is given by Table 6-4. If the server wishes to disconnect a client without a client request, the server sends a message of type ERROR that contains a Data Unit with the description for a critical error.

Head	Protocol Identifier	-
	Message Type	LEAVE
	Endpoint Identifier	*client-identifier*
	Interval Number	*current-interval-number*
	Last Interval Number	*last-interval-number*

Table 6-4: Message d) - leaving a session

6.3.2 State Copies Maintenance

When a client joined a session, the client sends messages of type REQUEST containing operations to modify the lead state, while the server replies with messages of type UPDATE containing the effects of remote operations the client has to apply to its local copy of the state. As explained in the communication scheme, an update message is not necessarily a direct reply to the last request messages. Messages and thus operation are pipelined as depicted in Figure 6-8.

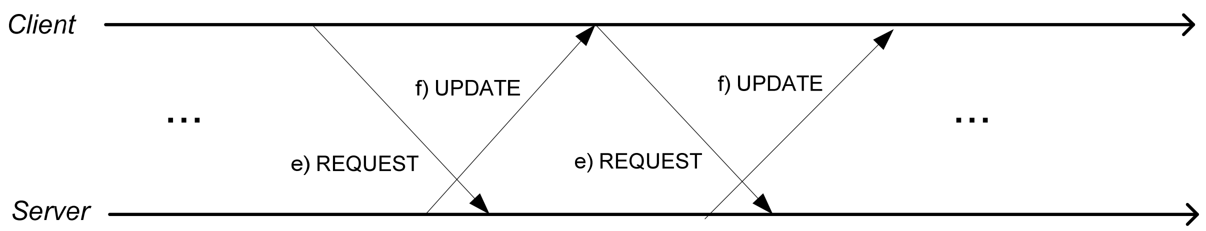


Figure 6-8: Operation exchange

A correct request message does not require any Message Parts. Such empty messages maintain the heartbeat the server uses to determine client presence and latency. The example in Table 6-5 contains one Message Part with operations concerning the scope referred to by *scope-identifier.* The Message Part contains one custom operation with a custom parameter. Operations and their semantic depend on the application, its state representation and use cases. Operations may also have multiple parameters.

Head	Protocol Identifier	-	
	Message Type	REQUEST	
	Endpoint Identifier	*client-identifier*	
	Interval Number	*current-interval-number + number-of-pipelines*	
	Last Interval Number	*current-interval-number*	
Message Part	Part Name	*scope-identifier*	
	Data Unit	Data Unit Type	*CUSTOM-OPERATION*
		Data Unit Parameter	*custom-operation-parameter*

Table 6-5: Message e) - requesting the application of operations to the lead state

Like the request message the update message does not have to contain any Message Parts and Data Units. Nonetheless, empty update messages are required to maintain a proper connection between client and server. For example, they indicate the beginning of the next interval and tell the client about the last received request message. The update message shown in Table 6-6 contains one custom operation with two parameters.

Head	Protocol Identifier	-	
	Message Type	UPDATE	
	Endpoint Identifier	*client-identifier*	
	Interval Number	*current-interval-number +1*	
	Last Interval Number	*current-interval-number + number-of-pipelines*	
Message Part	Part Name	*scope-identifier*	
	Data Unit	Data Unit Type	*CUSTOM-OPERATION*
		Data Unit Parameter	*custom-operation-parameter-1*
		Data Unit Parameter	*custom-operation-parameter-2*

Table 6-6: Message f) - updating a state copy

The association of Message Parts and scopes of the state can have a positive impact on the overall system performance, not only with regard to the message size but also with regard to message processing. Figure 6-9 depicts an example of the positive effects scopes have in message processing. Usually operations contained in a message for interval *i+1* are causally dependent on operations of a message for interval *i*. Thus, they must not be executed before all causally preceding operations have been applied.

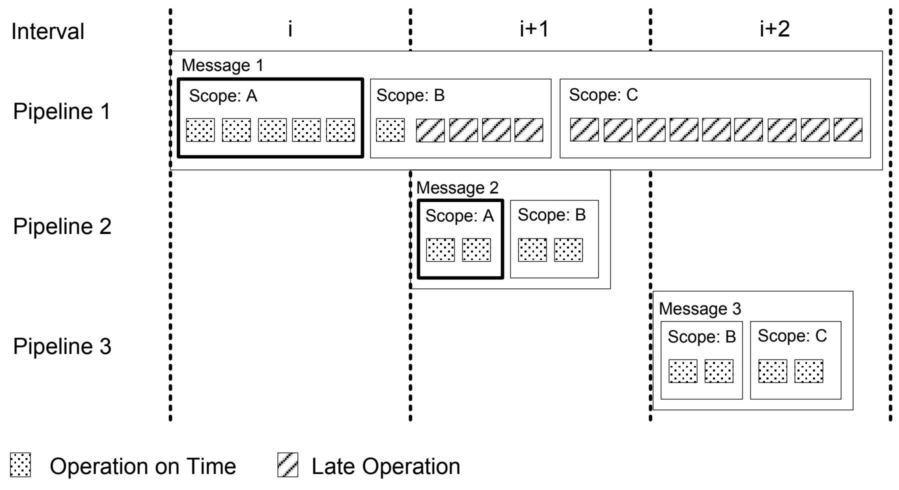


Figure 6-9: Processing of late messages

If the message size delays the processing of operations for interval *i* until interval *i+1* the operations for the later interval are delayed and the application blocks. Since scopes are disjoint, the beginning of a new part in a message implies that all remaining operations in the message do not causally depend on the operations of this message already processed. Accordingly, operations of the next message that share the same scope as those operations already applied from the first one can be applied too without delay. For transmission, the ordering of Message Parts according to the data size of payload while starting with the shortest is reasonable.

6.3.3 Scope Subscription and Unsubscription

A client can subscribe for multiple scopes. As long as a client is not subscribed for a scope, messages for this client must not contain a Message Part for that scope. Also client messages must not include a Message Part for scopes they have not subscribed for. The subscription for a scope is piggybacked on the regular request and update messages like depicted in Figure 6-10 and integrated in the session related Message Part.

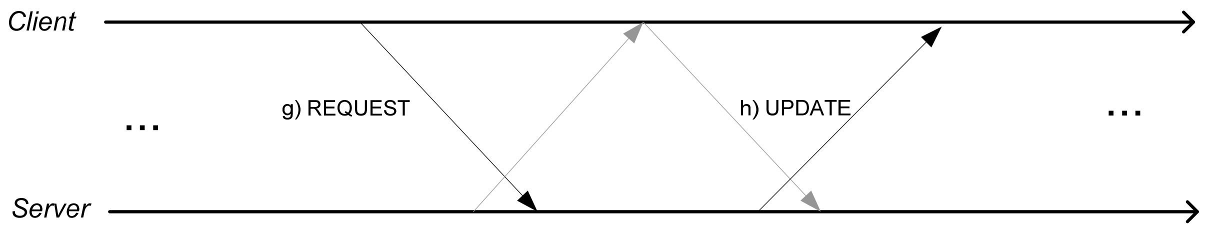


Figure 6-10: Scope subscription

Analogue to the initial join messages, a request message has to contain a session related Message Part with a Data Unit containing the name of the scope to subscribe for. A client can subscribe for multiple scopes at once. The example in Table 6-7 contains only one subscription. Additionally the example includes an unsubscription, which is parameterized with the name of the scope the client wishes to unsubscribe for. If the session related Message Part of a request contains an unsubscription for a scope, the request must not contain any operations for this scope.

Head	Protocol Identifier	-	
	Message Type	REQUEST	
	Endpoint Identifier	*client-identifier*	
	Interval Number	*current-interval-number + number-of-pipelines*	
	Last Interval Number	*current-interval-number*	
Message Part	Part Name	-	
	Data Unit	Data Unit Type	SCOPE-SUBSCRIBE
		Data Unit Parameter	*scope-identifier-2*
	Data Unit	Data Unit Type	SCOPE-UNSUBSCRIBE
		Data Unit Parameter	*scope-identifier*

Table 6-7: Message g) – subscribing and unsubscribing a scope

An unsubscription is not acknowledged explicitly by the server. At the earliest with the interval the request addresses, the server stops sending Message Parts for the unsubscribed scope. At the latest when the client receives the acknowledgement for the request from the server (indicated by the interval number), the client can be sure to receive no further updates for the unsubscribed scope. A subscription is acknowledged with a Message Part for the newly subscribed scope and a copy of the current lead state. A Message Part with a Data Unit for scope initialization must not contain any operations.

Head	Protocol Identifier	-	
	Message Type	UPDATE	
	Endpoint Identifier	*client-identifier*	
	Interval Number	*current-interval-number+1*	
	Last Interval Number	*current-interval-number+ number-of-pipelines*	
Message Part	Part Name	*scope-identifier-2*	
	Data Unit	Data Unit Type	SCOPE-INIT
		Data Unit Parameter	*state-representation*

Table 6-8: Message f) - initializing a scope

6.3.4 Summary of Basic Profile

The preceding sections introduced the network interface along with a couple of examples for basic use cases. The types for Message and Data Unit are derived from the basic profile of the network interface, which is the minimal set of definitions required for an implementation of the communication scheme. Table 6-9and Table 6-10 summarize these types again.

Message Type	Utilization	Description
JOIN	Client to Server	Connect to a session, begin request/update cycle.
LEAVE	Client to Server	Leave a session, i.e. request the termination of the connection to the session by the server.
REQUEST	Client to Server	Request operations and scope changes.
UPDATE	Server to Client	Deliver operation to the client in order to update the state copy.
ERROR	Server to Client	Notify the occurrence of a critical error and the termination of the session connection by the server.

Table 6-9: Message types of basic network interface profile

Data Unit Type	Message Types	Ses. Rel.	Description
SESSION-REFERENCE	JOIN	yes	Name the session to join.
SCOPE-SUBSCRIBE	JOIN, REQUEST	yes	Subscribe to a scope.
SCOPE-UNSUBSCRIBE	REQUEST	yes	Unsubscribe from a scope.
SCOPE-INIT	UPDATE	no	Initially set the state copy.
PIPELINE-COUNT	UPDATE	yes	Number of intervals within one roundtrip.
INSTANT-CHECK-FUNCTION	UPDATE	yes	Representation of a schema function.
REGISTER-OPERATION	UPDATE	yes	Representation of operation function.
UNREGISTER-OPERATION	UPDATE	yes	The operation not supported any longer.
INCLUSION-FUNCTION	UPDATE	yes	Representation of inclusion function.
CRITICAL-ERROR	ERROR	yes	Code and message of a critical error.
ERROR	UPDATE, ERROR	no	Code and message of an uncritical error.

Table 6-10: Data Unit types of basic network interface profile

6.4 Client Interface

The Client Interface exposes the capabilities of the Network Interface to the application while hiding special features of algorithm and network protocol. For example the pipelining is not modeled on this interface. Instead an asynchronous notification mechanism is intended to substitute the effects of pipelining for the application programmer. None of the functions in this interface is designed to be called synchronously. Whenever the processing time of a function call cannot be neglected, a callback function is used. Like the Network Interface the Client Interface specified below only contains the minimal set of required functions to enable the realization of an interactive application. Operations and their parameters have to be defined by the application developer.

$\boldsymbol{connect}: Literal \times Literal \times (Integer \rightarrow \emptyset) \rightarrow \emptyset$ joins a running session on the server. In *connect(name, scope, callback) name* is the identifier of the session, *scope* the name of the initial scope and *callback* a function expecting a session handle as parameter. The session handle is a client-only reference to a session and has to be used in all following function calls on the same session. The *name* can also include a reference to the network host of the server and the corresponding process.

$\boldsymbol{disconnect}: Integer \rightarrow \emptyset$ closes the connection of a client to a session without stopping the session. In *disconnect(handle) handle* is a reference of the session to terminate.

$\boldsymbol{read}: (S^* \rightarrow S^*) \times Integer \rightarrow S^*$ is a generic function to obtain the subset of the state representation at the client. In *result:=read(extract, handle), extract* is a custom function to get a subset of the current state, *handle* is the session reference and *result* is the result of executing *extract* in the current state. The function *extract* could also be substituted by a name for an addressable subset in the state, but for now a function provides the more flexible solution.

$\boldsymbol{apply}: Literal \times Parameter \times Integer \rightarrow Boolean$ executes an operation on the state. In *instant:=apply(operation, parameter, handle)* operation is the name of the function to call with *parameter* on the state. The result *instant* determines if the next call of *read* on the modified state would already return the changes (true) or if the operation takes effect later (false). The return value depends on the operation mode and the schema defined in the system configuration.

$\boldsymbol{listen}: \left((S^* \rightarrow S^*) \times Integer \rightarrow \emptyset\right) \times Integer \rightarrow \emptyset$ allows subscription for changes in the state. In *listen(callback, handle)* the function *callback* takes an extract function and a *handle* of the session the state change occurred in. The extract function serves as filter for a call of *read* and thus determines where in the state the change appeared. Again the extract function could also be replaced by an address to a subset of the state.

$\boldsymbol{subscribe}: Literal \times (Literal \times Boolean \times Integer \rightarrow \emptyset) \times Integer \rightarrow \emptyset$ subscribes the client to a scope. In *subscribe(scope, callback, handle) scope* determines the name of the scope to subscribe for, the function *callback* is invoked when the state copy for the scope is finally initialized at the client or if an error occurs. The *callback* therefore gets the scope name, a flag that indicates success or error and the session handle again. The parameter *handle* is the session handle.

$\boldsymbol{unsubscribe}: Literal\ \times Integer\ \rightarrow\ \emptyset$ removes the subscription of a client for a scope. In *unsubscribe(scope, handle)* the literal *scope* represents the scope name, while *handle* is the session handle.

$\boldsymbol{error}: (Literal\ \times Integer\ \rightarrow \emptyset) \rightarrow \emptyset$ supports the subscription for error notifications. In *error(callback, handle)* the function *callback* is invoked with error message and session handle whenever the client experiences an error. The parameter *handle* is the session handle.

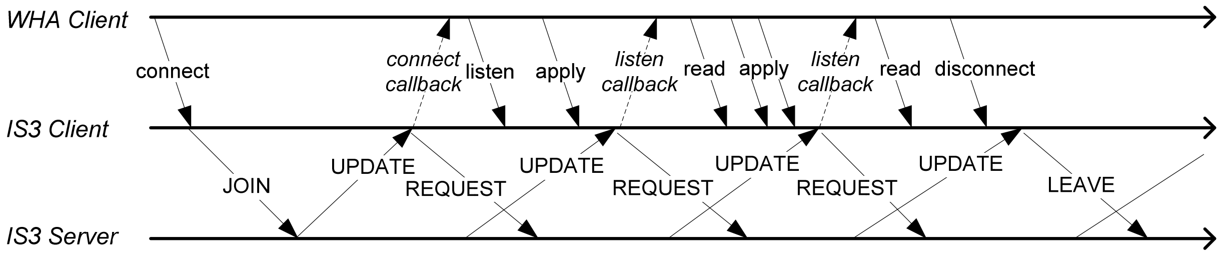


Figure 6-11: Example of the interplay between network and client interface

Figure 6-11 shows an example for the interplay of function calls on the client interface and messages on the network interface. The interval of updates also determines when changes to the state are cascaded to the WHA Client.

6.5 Server Interface

The Server Interface is the main control point for the IS3 systems. Since sessions are not initiated by the clients, the Server Interface is the only tool to setup and maintain sessions. Additionally, functions for the content-wise control of sessions are included. These functions enable application logic in the WHA Server to influence the development of the state. The Server Interface is designed to connect WHA Server and IS3 Server directly, i.e. not through a link suffering from communication latency. For that reason the processing time of all functions is supposed to be negligible and the interface has to be designed for synchronous function calls. If neither bandwidth nor latency can be guaranteed for the link between WHA Server and IS3 Server, a WHA Surrogate has to be used. The WHA Surrogate accesses the IS3 Server via the same interface.

$\boldsymbol{create}: Literal\ \rightarrow Integer$ creates a new session, but does not open it for clients to join. In *handle:=create(name), name* is the session reference clients will use to address a session when joining and *handle* is an internal session identifier the application has to use for all subsequent function calls to the same session.

$\boldsymbol{scope}: Literal \times (S^{*} \rightarrow S^{*}) \times Integer \rightarrow \emptyset$ adds a scope for the state. In *scope(name, filter, handle)* the parameter *name* represents the name of the scope and the function *filter* a mapping from the entire state to a subset of the state. Multiple scopes can be configured for each session. The parameter *handle* contains the session handle.

$\boldsymbol{load}: S^{*} \times Integer \rightarrow \emptyset$ sets the initial state for a session. In *load(state, handle)* the value of *state* is the representation of a state. The parameter *handle* contains the session handle.

$\boldsymbol{operation}: Literal\ \times Function\ \times Integer\ \rightarrow Boolean$ registers an operation function to a session. In a call *success:=register(operation, function, handle) operation* is the name of the operation to be used when referring to the function, *function* is any function in terms of Definition 2 and *handle* is the session handle. The return value *success* determines if the operation was registerd successfully (true). The registration fails of no inclusions have been installed for the operation to register. This function has to be called separately for all operations to be installed.

$\boldsymbol{nooperation}: Literal\ \times (Literal \times Integer \rightarrow \emptyset) \times Integer\ \rightarrow \emptyset$ uninstalls an operation function from a session. In *unregister(operation, callback, handle)* the parameter *operation* contains the name of the operation not to be supported longer, the function *callback* is invoked when all connected clients received the unregister. The parameter *handle* contains the session handle. A call of nooperation also removes the corresponding inclusions and schema check for instant application.

$\boldsymbol{instant}: Literal\ \times\ (Parameter\ \rightarrow Boolean) \times Integer \rightarrow \emptyset$ sets the schema function instant, which determines whether an operation can be applied to a state copy immediately or has to be sent to the server first. In *instant(operation, check, handle), operation* is the name of the operation to perform checks on. The function *check* finally expects the operation parameters and detects if the operation can be applied instantly (true) or not (false). For the check function the operation name is constant. The parameter *handle* contains the session handle. The function has to be called for any operation in the session to check.

$\boldsymbol{inclusion}: Literal\ \times Literal\ \times (Parameter \times Parameter \rightarrow Literal\ \times Parameter\ ) \times$
$Integer \rightarrow \emptyset$ sets the inclusion function for a session. In *inclusion(operation1,operation2, include, handle)* the literals *operation1* and *operation2* contain the names of operations the inclusion applies to. The function *include* represents the operation inclusion function, while the operation name are supposed to be constant in this function. The parameter *handle* contains the session handle. The inclusion function has to be called as many times as there are combinations of operations. Inclusions have to be set up before operations are registered.

$\boldsymbol{competing}: ((Literal\ \times Parameter) \times (Literal\ \times Parameter) \rightarrow Boolean) \times Integer \rightarrow \emptyset$
sets the function to check whether two operation invocations are competing or not. In *competing(compete, handle)* the function *compete* takes two operation invocations and checks them for a conflict. The parameter *handle* contains the session handle.

$\boldsymbol{open}: Integer \rightarrow Boolean$ has to be called when the configuration of a session is setup properly. In *success:=open(handle) handle* names the session to be opened. The result *success* determines whether the session has been opened or not. If a session could not be opened no operation is registered.

$\boldsymbol{terminate}: Integer \rightarrow \emptyset$ terminates a session. Calling *terminate(handle)* results in the termination of the session referred to with *handle*.

In addition to the above session control functions ***apply***, ***read*** and ***listen***, as defined on the Client Interface are also available on the Server Interface, with the little difference that calling apply at the server always returns true. This is because the IS3 Server does not maintain a state copy for the WHA Server, instead operation are directly applied to the lead state. Operations on this interface are thus intentionally out of synchronization with the clients. If the WHA sever has to act as state controlling instance (e.g. referee) between the clients, the fast link is mandatory. If the Server Interface cannot be implemented as local interface for any reason, remote evaluation concepts such as robots are recommended to ensure fast responses to client operations on behalf of the application operator.

6.6 Surrogate Management Interface

The Surrogate Management Interface enables a WHA Server to install and manage surrogates in the IS3 SRE and is thus directly linked to the IS3 Server. A WHA Surrogate can handle multiple IS3 sessions in parallel. A limitation of resource usage on the host system is recommended as well as a clear separation of the execution scopes (sandboxing). At least the following functions should be supported by the management interface.

$\boldsymbol{deploy}: Surrogate \times (Integer \rightarrow \emptyset) \rightarrow \emptyset$ deploys a WHA Surrogate to the IS3 SRE. In *deploy(surrogate, callback)* the parameter surrogate represents the executable code base of the WHA Surrogate. When a surrogate has been deployed successfully, the function *callback* is invoked with the handle of the WHA Surrogate instance as parameter.

$\boldsymbol{undeploy}: Integer \times (Literal \rightarrow \emptyset) \rightarrow \emptyset$ removes a WHA Surrogate from the IS3 SRE, after stopping the instance if it is running. In *undeploy*(*handle*, *callback*), handle refers to the instance to be stopped and removed. The function *callback* is invoked when the surrogate was finally undeployed. The function takes as parameter a literal with a status report.

$\boldsymbol{start}: Integer \times (Literal \rightarrow \emptyset) \rightarrow \emptyset$ starts a WHA surrogate on the IS3 SRE. In *start*(*handle, callback*) *handle* names the instance to start. The function callback is invoked when the start procedure was completed and takes as parameter the status of the instance or information about errors while starting.

$\boldsymbol{stop}: Integer \times (Literal \rightarrow \emptyset) \rightarrow \emptyset$ stops a WHA Surrogate on the IS3 SRE. In *stop*(*handle, callback*) *handle* names the instance to be stopped. The function callback is invoked when the stop procedure was completed and takes as parameter the status of the instance or information about errors while stopping.

$\boldsymbol{report}: Integer \rightarrow Literal$ can be used to obtain the status of any deployed instance by its handle.

7 Web Object State Model

The IS3 system is defined to support all representations chosen by application developers for the shared state. The web object state model is a flexible state representation, tailored to the standard tree structure used in web hypertext applications. The following sections introduce the state representation, the basic operations to modify the state and the inclusion function for operational transformation required for the state objects.

7.1 State Representation

Document Object Model and non-cyclic ECMA object notation both utilize the leaf tree as central structure for non-primitive data. While leaves in the DOM are text nodes, nodes in ECMA object model are primitive data types. The web object state model generalizes leaf nodes as literals.

Labeled edges connect a node with its child nodes. Each node is either a leaf or has at least one child node. Each leaf is addressable by a path which is comprised of the sequence of labels from the root node to itself. Leaf$_2$ in Figure 7-1a) is addressable by the sequence *label$_1$/label$_4$/label$_6$*.

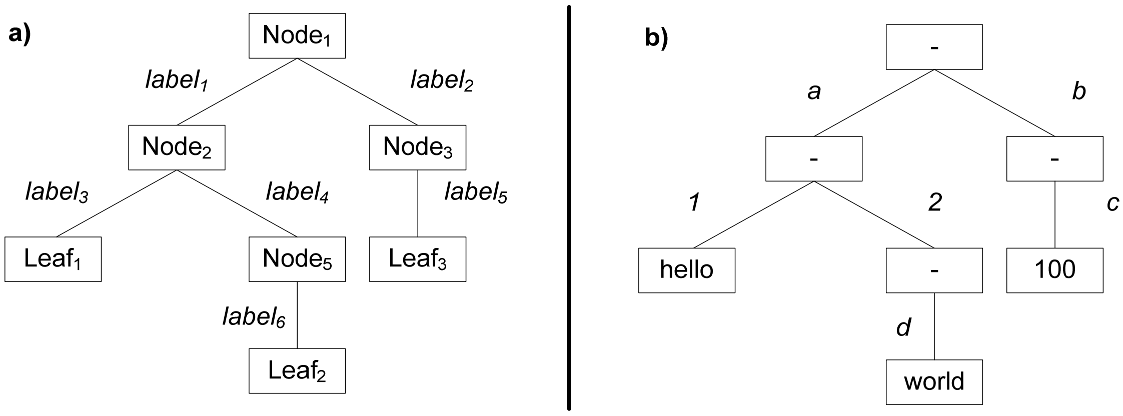


Figure 7-1: Partially ordered tree

In contrast to the DOM the ECMA Script object model is only partially ordered. In the DOM the child nodes of the same parent are addressed by a zero-initialized numeric offset and thus have an explicit order. If the tree in Figure 7-1a) were a DOM, *Leaf$_2$* would need to be addressed by *0/1/0*. The ECMA script object model allows string literals as alternative to numeric offsets. An example is given in Figure 7-1a). *Leaf$_2$* is addressed with *a/2/d*. Nonetheless the usage of numeric names allows an ordering of child nodes if required. For example, the node *a/2* is said to be positioned behind *a/1*.

Since the DOM model is a subset of the ECMA Script object model, the web object state model follows the latter and implicitly assumes an order for child nodes connected with numerically labeled edges to the same parent node.

The leaf nodes are the relevant ones in the tree. The state representation of the web object state model is accordingly tailored to the leaf nodes. Instead of mapping the whole tree, the state representation is comprised of paths, values and their relations as defined in the following.

Definition 7-1: Web Object State

The Web Object State $WOS = < Path, Value, lookup >$ is a data structure containing the set of paths, the set of values and a function mapping paths to values. The set of paths is defined as the set of vectors wherein the components of each vector are literal labels. A literal is any numeric or alphanumeric representation. $\mathcal{P}(S)$ is the power set of any set S.

$$Path \in \mathcal{P}(Path^*)$$

$$Path^* = \left\{ \begin{pmatrix} label_1 \\ \dots \\ label_i \\ \dots \\ label_n \end{pmatrix} \middle| label_1, \dots, label_i, \dots, label_n \in Literal \right\}$$

The set $Value \subseteq Literal$ is the set of values reachable by paths. The subjective function $lookup: Path \rightarrow Value$ is a mapping of paths to values, i.e. all paths in the tree pointing to a leaf node.

For instance the WOS of the tree depicted in Figure 7-1b) would be represented as follows:

$$example_1 = \langle \left\{ \begin{pmatrix} a \\ 1 \end{pmatrix}, \begin{pmatrix} a \\ 2 \\ d \end{pmatrix}, \begin{pmatrix} b \\ c \end{pmatrix} \right\}, \{hello, world, 100\}, lookup(p) \stackrel{\text{def}}{=} \begin{cases} hello, if\ p = \begin{pmatrix} a \\ 1 \end{pmatrix} \\ world, if\ p = \begin{pmatrix} a \\ 2 \\ d \end{pmatrix} \\ 100, if\ p = \begin{pmatrix} b \\ c \end{pmatrix} \end{cases} \rangle$$

In addition to the state representation, the following tool functions $path: WOS \rightarrow Path$ and $value: WOS \rightarrow Value$ are introduced, which return the sets of paths and values for a WOS.

7.2 Operations to Modify the State

As addition to the state representation, a minimal set of functions is defined on the WOS representation for later utilization within state modifying operations. The functions are focused on the concept of the partially ordered tree. The core functions support the setting, clearing and rewriting of path-value relations. The combined utilization of core functions results in a set of functions more strongly tailored to the modification of trees.

Definition 7-2: Set Path-Value Relation

The function $set: WOS \times Path^* \times Literal \rightarrow WOS$ is the basic function to establish a relation between path and value in a WOS instance. The function is defined for four cases. i) If neither the new path nor the new value is element in the respective set of the WOS, the elements are added to the sets and the function *lookup* is extended. ii) If the value already exists in the set of values, the new path is added to the set of paths and the *lookup* function is extended. iii) If the path already exists, the mapping in the lookup is changed, the new value is added to the set of values and the previously related value removed if no other path is pointing to this value. iv) If path and value to be set already exist, the function *lookup* is redefined to map path to value and the value formerly mapped by the path is removed if no other path maps to it.

$$set(S, p, v) \overset{\text{def}}{=} \begin{cases} \langle Paths_S \cup \{p\}, Values_S \cup \{v\}, lookup_{new}\rangle, if\ p \notin Paths_S \ \wedge v \ \notin Values_S \\ \langle Paths_S \cup \{p\}, Values_S, lookup_{new}\rangle, if\ p \notin Paths_S \ \wedge v \ \in Values_S \\ \langle Paths_S, Value_{new}, lookup_{new}\rangle, if\ p \in Paths_S \wedge \nexists_{r\ \in Paths_S\ r\neq p} lookup_S(p) = lookup_S(r) \\ \langle Paths_S, Value_{short}, lookup_{new}\rangle, if\ p \in Paths_S \ \wedge v \ \in Values_S \wedge lookup_S(p) \neq v \end{cases}$$

with $Value_{new} = (Values_S \cup \{v\})/\{lookup_S(p)\}$ and $lookup_{new}(q) \overset{\text{def}}{=} \begin{cases} v, if\ q = p \\ lookup_S(q), else \end{cases}$

and $Value_{short} = \begin{cases} Values_S/\{lookup_S(p)\}, if\ \nexists_{r\ \in Paths_S} lookup_S(r) = v \\ Values_S, else \end{cases}$

The tree represented by the set of paths is restricted in the sense that leaf nodes must not have child nodes. For that reason an additional precondition is introduced for the *set* function.

Precondition: $\nexists_{q\in Paths_S}\ head(p, |q|) = q$

An example for the utilization of the third case of the *set* function is depicted in Figure 7-2 for the above example. The right-hand part of the figure would result from the left-hand part on applying $set\left(example_1, \binom{b}{c}, 200\right)$.

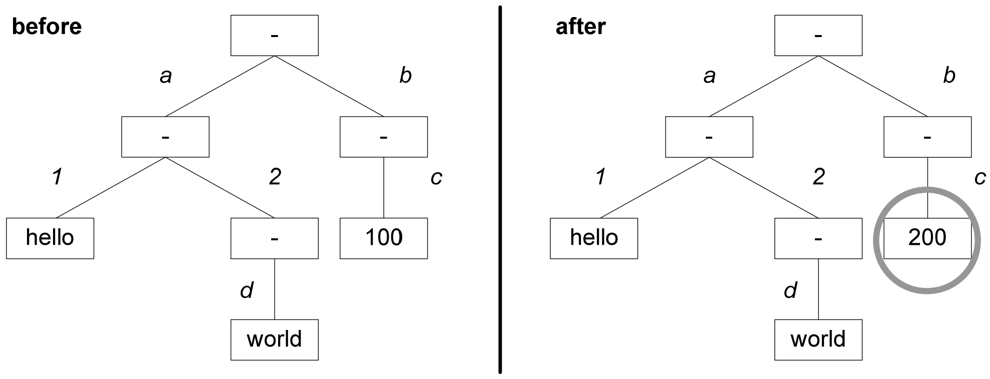


Figure 7-2: Example for (re-)setting a path-value relation

Definition 7-3: Unset Path-Value Relation

The function $unset: WOS \times Path^* \rightarrow WOS$ is the basic function to remove a relation between path and value from a WOS instance. For this purpose the function removes the mapping from the lookup function, the path from the set of paths and the value from the set of values, if no other paths refers to the same value.

$unset(S, p) \overset{\text{def}}{=}$

$$\begin{cases} \langle Paths_S/\{p\}, Value_{new}, lookup_{new}\rangle, if\ p \in Paths_S\ \wedge \nexists_{r \in Paths_S\ r \neq p} lookup_S(p) = lookup_S(r) \\ \langle Paths_S/\{p\}, Values_S, lookup_{new}\rangle, if\ p \in Paths_S \wedge \exists_{r \in Paths_S\ r\neq p} lookup_S(p) = lookup_S(r) \end{cases}$$

with $Value_{new} = Values_S/\{lookup_S(p)\}$ and $lookup_{new}(q) \overset{\text{def}}{=} \begin{cases} undef, if\ q = p \\ lookup_S(q), else \end{cases}$

The unset function can be used to prune branches. An example of the utilization of the *unset* function is depicted in Figure 7-3. The right-hand part of the figure would result in the left-hand part after applying:

$$unset\left(example_1, \binom{b}{c}\right)$$

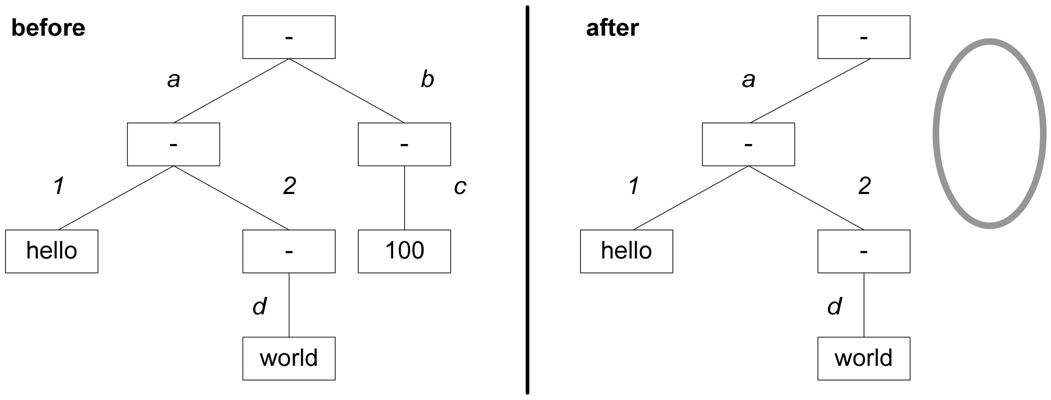


Figure 7-3: Unset path-value relation

Definition 7-4: Rewrite Path-Value Relation

The function $rewrite: WOS \times Path^* \times Path^* \rightarrow WOS$ is the third and last essential core function that consequently combines *set* and *unset* function to rewrite a path and its mapping to a value.

$$rewrite(S, p, q) \overset{\text{def}}{=} unset\big(set\big(S, q, lookup_S(p)\big), p\big)$$

The core functions represent the transformations essential to create any complex function. However, they are not suited for transformation on the WOS that follow the semantic of partially ordered trees. In fact the core functions do not respect any order in the paths at all. The *shift* function as summarized in Algorithm 7-1, is used to support not only the assignment of names and values, but also insertions in a sequence of ordered nodes with the same parent node. The shift function is defined with the help of the vector functions $head: Path^* \times Number \rightarrow Path^*$, $tail: Number \times Path^* \rightarrow Path^*$, and $\circ: Path^* \times Path^* \rightarrow Path^*$, where *head(p, x)* returns the *x* first components of *p* as vector, *tail(x, p)* returns the last x components of *p*, and $p \circ q$ returns a vector of length */p/ + /q/* with the components of both, beginning with those of *p*. Additionally the support functions $filter: \mathcal{P}(Path^*) \times Path^* \rightarrow \mathcal{P}(Path^*)$ and $sort: Path^* \times Natural \rightarrow List_{Path^*}$ are utilized. The function *filter(P, p)* returns the set of all elements in set *P* which begin with *p*. The function *sort(P, c)* returns the a list of all elements in *P* ordered by component *c* starting with the lowest value and ending with the highest.

```
1   shift: WOS × Path × Integer → WOS
2   shift(S, p, k) := $S_{new}$ where
3   $S_{new}$ = S
4   if |p|>0 then
5   paths = sort(filter($Path_S$, head(p,|p|-1)), |p|-1)
6   if k > 0 then
7    for i = |paths|-1 to 0 step -1
8     if numeric(paths[i][ |p|-1]) and paths[i][ |p|-1] ≥ p[|p|-1] then
9      $S_{new}$= rewrite($S_{new}$, paths[i], head(paths[i],|p|-1)∘(paths[i] [|p|-1] + k)∘tail(|p|, paths[i]))
10    endif
11   endfor
12  else
13   for i = 0 to |paths|-1 step 1
14    if numeric(paths[i][ |p|-1]) and paths[i][ |p|-1] ≥ p[|p|-1] then
15     $S_{new}$= rewrite($S_{new}$, paths[i], head(paths[i],|p|-1)∘(paths[i] [|p|-1] + k)∘tail(|p|, paths[i]))
16    endif
17   endfor
18  endif
19  endif
```

Algorithm 7-1: Shift function

The *shift* function implicitly assumes the label *p[|p|-1]* to be a numeric literal, so that it can be changed in its value by adding *k.* The path denoted by *p* is partial in the sense that it does not point to a leaf node. The precondition *|p| > 0* has to be fulfilled, otherwise *shift.* Figure 7-4 depicts how the *shift* function changes the state. The right-hand side of the figure represents the state after applying $shift\left(example_2, \binom{a}{2}, 1\right)$, while *example₂* is the representation of the left-hand side of the figure.

The *shift* function utilizes the rewrite function to increase or decrease depending on *k,* the label values of numerically labeled edges to nodes of the same parent. To avoid overwriting of values and the loss of data, Algorithm 7-1 switches the end from which to start the shifting in line 6. The lines 7-11 and 13-17 are almost identical. Only the initialization of the loop differs according to the shifting direction. The function $numeric\colon Literal \rightarrow Boolean$ checks if a literal is numeric (true) or alphanumeric (false).

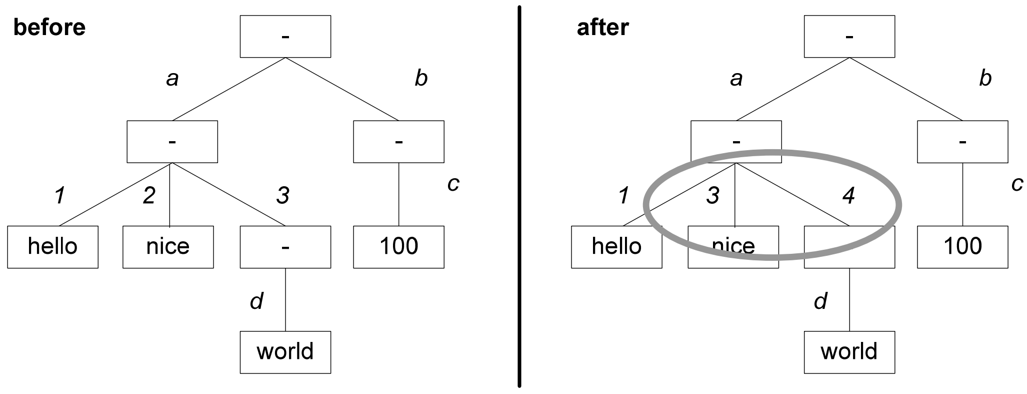


Figure 7-4: Shifting nodes

The function *set* and *unset* are appropriate and create branches in the tree structure of the WOS. With the help of the *shift* functions, the two functions can be extended to respect an ordering among child nodes of the same parent node. The resulting functions are accordingly named *insert* and *truncate*.

Definition 7-5: Insert and Truncate

The functions $insert: WOS \times Path^{*} \times Path^{*} \times Literal \rightarrow WOS$ and the function $truncate: WOS \times Path \times Path \rightarrow WOS$ combine shift function, set and unset function to enable the basic insertion of nodes into ordered sequences or the removal of nodes while restoring the continuity of the sequence.

$$insert(S, p_{head}, p_{tail}, v) = set(shift(S, p_{head}, 1), p_{head} \circ p_{tail}, v)$$

$$truncate(S, p_{head}, p_{tail}) = shift\big(unset_{multipe}(S, p_{head}), p_{head}, -1\big)$$

A path can contain several numeric labels. To indicate on which level to shift, the complete path to a leaf node is separated into two parts. The first part p_{head} points to the level to shift to, the second part p_{tail} names the rest of the path. The composition of both parts point to the value.

Precondition: $numeric(p_{head}[|p_{head}| - 1])$

The function $unset_{multiple}$ is a helper function and defined as follows:

Definition 7-6: Unset Multiple Path-Value Relations

The function $unset_{multiple}: WOS \times Path^{*} \rightarrow WOS$ removes path value relations from S for all paths which begin with p.

$$unset_{multiple}(S, p) = \begin{cases} unset_{multiple}(unset(S, r), p), if\ filter(Path_S, p) \neq \{\} \\ \qquad\qquad S, else \end{cases}$$

$$where\ r \in filter(Path_S, p)$$

While *rewrite*, *shift* and $unset_{multiple}$ are introduced for the seamless definition of all functions as well as their parameters and semantics, the set of finally relevant functions for the communication of state changes contains only the functions *set*, *unset*, *insert* and *truncate*.

7.3 Inclusion Function

With state representation and the functions to modify the state, the inclusion function for operation transformation can be defined, which also depends on the application context. The inclusion function has the signature $include: (f^* \times p^*) \times (f^* \times p^*) \to (f^* \times p^*)$. For better readability Table 7-1 summarizes all the cases required for the realization of the function on the WOS model. The actual state to which an operation shall later be applied to is determined implicitly by the session and for this reason is not listed among the function parameters. If an operation becomes obsolete, for example because the path it refers to does not longer exist, the operation is refused and its place in the operation queue overwritten with an empty operation denoted by the symbol *ε*.

Reference operation	Operation to transform	Resulting operation
set, (p_1, v_1)	set, (p_2,v_2)	set, (p_2,v_2); if $p_2 \neq p_1$ set, (p_2, v_2); if $p_2 = p_1$ and $v_2 > v_1$ set, (p_2, v_1); if $p_2 = p_1$ and $v_1 \geq v_2$
set, (p_1,v_1)	unset, (p_2)	ε; if $p_1 = p_2$ unset, (p_2); else
set, (p_1,v_1)	insert, (h_2, t_2, v_2)	insert, (h_2, t_2, v_2)
set, (p_1,v_1)	truncate, (h_2, t_2)	truncate, (h_2, t_2)
unset, (p_1)	unset, (p_2)	unset, (p_2); if $p_1 \neq p_2$ ε; else
unset, (p_1)	set, (p_2,v_2)	set, (p_2,v_2)
unset, (p_1)	insert, (h_2, t_2, v_2)	insert, (h_2, t_2, v_2)
unset, (p_1)	truncate, (h_2, t_2)	truncate, (h_2, t_2)
insert, (h_1, t_1, v_1)	set, (p_2,v_2)	set,$(head(p_2,\|h_1\|) \circ (p_2[\|h_1\|-1]+1) \circ tail(\|h_1\|,p_2),v_2)$; if $head(h_1,\|h_1\|-1) = head(p_2,\|h_1\|-1)$ $\wedge$ $numeric(p_2[\|h_1\|-1])$ $\wedge$ $p_2[\|h_1\|-1] \geq h_1[\|h_1\|-1]$ set, (p_2,v_2); else
insert, (h_1, t_1, v_1)	unset, (p_2)	unset,$(head(p_2,\|h_1\|) \circ (p_2[\|h_1\|-1]+1) \circ tail(\|h_1\|,p_2))$; if $head(h_1,\|h_1\|-1) = head(p_2,\|h_1\|-1)$ $\wedge$ $numeric(p_2[\|h_1\|-1])$ $\wedge$ $p_2[\|h_1\|-1] \geq h_1[\|h_1\|-1]$ unset, (p_2); else
insert, (h_1, t_1, v_1)	insert, (h_2, t_2, v_2)	insert,$(head(p_2,\|h_1\|) \circ (p_2[\|h_1\|-1]+1) \circ tail(\|h_1\|,h_2), t_2, v_2)$; if $head(h_1,\|h_1\|-1) = head(h_2,\|h_1\|-1)$ $\wedge$ $numeric(h_2[\|h_1\|-1])$ $\wedge$ $h_2[\|h_1\|-1] \geq h_1[\|h_1\|-1]$ insert, (h_2, t_2, v_2); else

insert, (h_1, t_1, v_1)	truncate, (h_2, t_2)	truncate,$(head(p_2,\vert h_1\vert) \circ (p_2[\vert h_1\vert-1]+1) \circ tail(\vert h_1\vert,h_2), t_2)$; if $head(h_1,\vert h_1\vert-1) = head(h_2,\vert h_1\vert-1)$ $\land$ numeric$(h_2[\vert h_1\vert-1])$ $\land\ h_2[\vert h_1\vert-1] \geq h_1[\vert h_1\vert-1]$ truncate, (h_2, t_2); else
truncate, (h_1, t_1)	set, (p_2,v_2)	ε; if $h_1 \circ t_1 = p_2$ set,$(head(p_2,\vert h_1\vert) \circ (p_2[\vert h_1\vert-1]-1) \circ tail(\vert h_1\vert,p_2),v_2)$; if $head(h_1,\vert h_1\vert-1) = head(p_2,\vert h_1\vert-1)$ $\land$ numeric$(p_2[\vert h_1\vert-1])$ $\land\ p_2[\vert h_1\vert-1] \geq h_1[\vert h_1\vert-1]$ set, (p_2,v_2);else
truncate, (h_1, t_1)	unset, (p_2)	ε; if $h_1 \circ t_1 = p_2$ unset,$(head(p_2,\vert h_1\vert) \circ (p_2[\vert h_1\vert-1]-1) \circ tail(\vert h_1\vert,p_2))$; if $head(h_1,\vert h_1\vert-1) = head(p_2,\vert h_1\vert-1)$ $\land$ numeric$(p_2[\vert h_1\vert-1])$ $\land\ p_2[\vert h_1\vert-1] \geq h_1[\vert h_1\vert-1]$ unset, (p_2); else
truncate, (h_1, t_1)	insert, (h_2, t_2, v_2)	insert,$(head(p_2,\vert h_1\vert) \circ (p_2[\vert h_1\vert-1]-1) \circ tail(\vert h_1\vert,h_2), t_2, v_2)$; if $head(h_1,\vert h_1\vert-1) = head(h_2,\vert h_1\vert-1)$ $\land$ numeric$(h_2[\vert h_1\vert-1])$ $\land\ h_2[\vert h_1\vert-1] \geq h_1[\vert h_1\vert-1]$ insert, (h_2, t_2, v_2); else
truncate, (h_1, t_1)	truncate, (h_2, t_2)	ε; if $h_1 \circ t_1 = h_2 \circ t_2$ truncate,$(head(p_2,\vert h_1\vert) \circ (p_2[\vert h_1\vert-1]-1) \circ tail(\vert h_1\vert,h_2), t_2)$; if $head(h_1,\vert h_1\vert-1) = head(h_2,\vert h_1\vert-1)$ $\land$ numeric$(h_2[\vert h_1\vert-1])$ $\land\ h_2[\vert h_1\vert-1] \geq h_1[\vert h_1\vert-1]$ truncate, (h_2, t_2); else

Table 7-1: Case switch for inclusion function on WOS

The inclusion function is designed keep data in case of conflicting delete and write operations, as well as to avoid partial rollbacks. For example, if a *set* and an *unset* operation refer to the same path, the *unset* most likely happened without awareness of the new value to be set. Accordingly the inclusion function makes sure the converged state contains the new value for the path instead of none. If an *unset* and a *truncate* refer to the same path, the truncate is always chosen, because the *unset* is part of the definition of the *truncate* function.

7.4 Pseudo Random Tie-Breaking Method for the Inclusion Function

Section 5.4.2 concludes with a discussion of fair tie-breaking methods. If a single decision is considered in isolation, either a rejection of all competing operations or the random selection of a winner are fair alternatives. In case a decision is needed recurrently between operations from the same origin, a random distribution of success between the origins has to be granted.

The WOS tie-breaking rule bases on the operation value, i.e. the value to be set or inserted. The operation with the greater value is always preferred. Unless such a behavior is explicitly required by an application (e.g. auction platform) the tie-breaking rule cannot be considered fair. For that reason, the inclusion function is extended with an optional pseudo-random tie-breaking method. Random decision making raises two problems: i) clients and server nonetheless have to be able to come to the same decision, because both perform inclusions and ii) an even distribution of success among all origins has to be guaranteed, but the inclusion function only sees two operations from different origins on each execution.

Problem i) can be solved with a pseudo-random number generator (e.g. Linear Congruential Generator), which recursively creates pseudo-random numbers based on a seed value. Each transformation function is instantiated from the current bucket interval index *i*. The index can serve as common seed for all sites to ensure that equal sequences of pseudo-random numbers are generated. The consistency of the generated sequences is the basis for deterministic decision making.

A solution for problem ii) requires the inclusion function to be aware of the operation origins. Therefore, each operation originator (i.e. endpoint) requires a unique name. The origin names are selected from a finite name space Q and are sortable. Any sorted list of origin names is totally ordered. The origin name is added as parameter to each operation. The inclusion function derives the priority of an operation from its origin name by the use of the associative array *priority_i*. Thus, if the function $name: Operation \rightarrow Q$ returns the parameter of an operation with the origin name and $priority_i\left[name(O_{Client_1})\right] > priority_i\left[name(O_{Client_2})\right]$ the operation of *Client₁* is preferred over the operation of *Client₂*. The associative array *priorityᵢ* is prepared on the instantiation of the inclusion function for bucket interval *i*. One possible approach for the instantiation is presented in Algorithm 7-2:

```
1    $R_n$ = randomlist(i, |Q|)
2    for k = 0 to $|R_n|$
3      $priority_i[L_n[k]] = R_n[k]$
4    endfor
```

Algorithm 7-2: Initialization of the priority hash array

The function $randomlist: \mathbb{N} \times \mathbb{N} \rightarrow List$ uses the random number generator to create a list of $|Q|$ pseudo-random numbers starting from *i* as seed. This list has to contain each number *x* with $0 \leq x < |Q|$ exactly once. L_n is an ordered list of all elements in *Q.*

A weak point of this approach is the obvious predictability of the priorities for each interval number if the origin names are known. Two preventive options to ensure fairness are to run only in operation mode (1) and to obscure origin names by interval–related aliases with equal priorities in the version of the inclusion function used at the server.

8 The Pulsar System

IS3 describes the recommended system architecture, but is not a software blueprint itself. This chapter describes the design of the Pulsar System, an IS3 implementation, with specifications for client, server and network interfaces.

8.1 Overview

The Pulsar system is a full implementation of the IS3 reference architecture, designed for experimental evaluation of IS3 and the realization of a case study to be introduced in chapter 10. As intended by IS3, Pulsar supports all state models that can be implemented in accordance with the requirements and conditions defined in chapter 5. The Pulsar architecture is not designed to support one state model only. However, the Web Object State Model (WOS) is implemented as one possible configuration for the Pulsar system and is additionally reflected in the Application Programmer Interface (API).

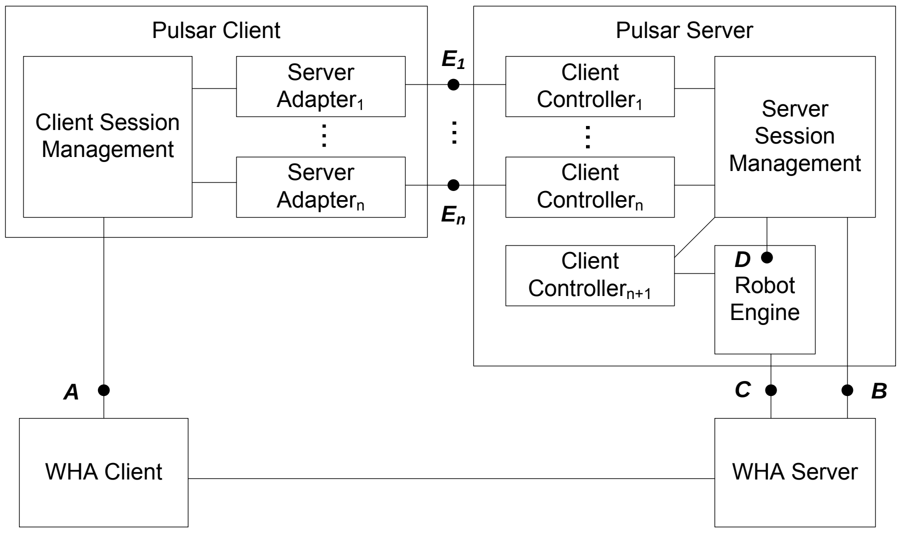


Figure 8-1: Pulsar system overview

Figure 8-1 depicts all components of the Pulsar implementation and their public interfaces. The main components of the Pulsar implementation are Pulsar client and server. The Pulsar client extends the WHA client with means to access a shared state that reflects modifications instantly. The WHA client accesses Pulsar through the client interface *A*. The Pulsar server implements the shared memory by maintaining multiple sessions, processing operations of different client and reflecting these operations to the client state copies. The Pulsar server additionally includes a Robot Engine, which provides application framework and runtime environment for small, autonomous components provided by the WHA operator, so called robots. Robots interface the Pulsar server via a low delay link and have privileged access to session state and operation stream. For example, the WHA operator can provide robots to add value to the interaction of clients or enforce rules on the operation stream and the state. The Pulsar System is designed with the assumption that Pulsar server and WHA server are physically separated and communicate via a non-negligible geographic distance. Robots are thus a suitable instrument to customize state and session without inducing extra delays. The implementation of the Robot Environment [90] is provided by Stanley Schragl and is therefore not documented here. The Pulsar server is accessed through the interfaces B, C and D, where B corresponds to the IS3 server interface, C enables deployment and life-cycle control of robots, and D the access of robots to the session management in the server core. Pulsar client and server are connected via network interfaces E_1-E_n.

8.2 Network Interfaces

8.2.1 Text Interface

The text interface is the default network interface of the Pulsar implementation. The name is chosen for the text-based representation of messages as enabled by the utilized application protocols. While this work is conducted, first standard drafts and implementations for the WebSocket appear. Originally the text interface is based on long-polling, but in the final iteration of the Pulsar system documented here a hybrid version based on XHR and the most recent WebSocket draft is utilized. For user agents that do not yet support the WebSocket API, a work around with Flash is utilized [91].

The types of message exchanged by Pulsar client and server are taken from the IS3 network interface recommendation introduced in section 6.3. Except for the message parts, the message structure is mostly adopted from IS3 and mapped to a text representation in JSON [39]. Fields that could be mapped to the underlying protocols more efficiently where extracted. In the hybrid approach session control messages and state control messages are separated. Messages of the type JOIN, LEAVE, SUBSCRIBE and UNSUBSCRIBE are transported via HTTP requests of the type POST. In contrast to the IS3 suggestions, endpoint identifier, session identifier and message type are therefore encoded in the path of the server URL and not as fields of the message in the request body. Scope names for subscriptions and unsubscriptions are also moved to the URL and appended as query string. This way, the request body does not have to contain any data at all. The responses contain the fields which, according to the IS3 network interface, should be included with an UPDATE message.

When a client joins a session and client and server do not yet maintain a WebSocket connection, the client has to open such a connection by issuing an HTTP request with a protocol upgrade. As soon as the connection is provided, the server initiates the pipe-lining scheme and starts sending messages of the type UPDATE to the client as depicted in Figure 8-2. The client has to reply immediately with a REQUEST message to each UPDATE message. As long as a client has joined a session, but no WebSocket connection is available, the server buffers UPDATE messages. Buffered messages are released in the order they were buffered when the connection was established. Before the last buffered UPDATE is delivered no current UPDATE is sent. Since buffered UPDATE message do not refer to the current bucket interval, the client must not reply to them with a REQUEST message. For this reason previously buffered messages are marked with a “patch” flag to indicate that they only help the client to update the state representation. The same procedure is applied if the WebSocket connection is unintentionally intermitted. To leave a session, the client sends the respective message via XHR. If the session was the last session the client maintained with the server the client is supposed to close the WebSocket connection too. Analogue to the LEAVE message in Figure 8-2, a client can subscribe and unsubcribe scopes and join further sessions while the WebSocket connection is open. Messages on HTTP and WebSocket are synchronized by the bucket interval of a session.

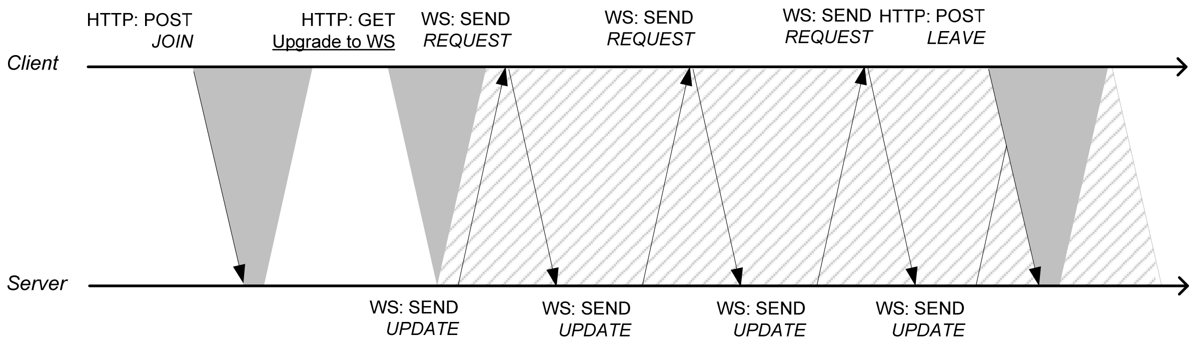


Figure 8-2: Connection handling in the text-based network interface

The hybrid approach was chosen with respect to the size of initialization messages. The response to an initialization message contains state representations and JS code for the configuration functions. If the state representation belongs to a text resource, for instance a shared document with 200 pages, the WebSocket connection would be busy as long the document is transferred to the client. Controlling the session via HTTP has the advantage that the control mechanisms keep response even when the WebSocket connection is blocked or busy and vice versa. The schemes of messages are summarized in Annex A.

8.2.2 Binary Interface

For the effective utilization of network capabilities in experimental evaluation the IS3 network interface is also implemented in a binary version for Pulsar. This version, called RTSYNC, is designed for direct usage with the Universal Datagram Protocol (UDP), but still uses the Transport Control Protocol (TCP) as fallback. Basically, all operations for or from a client for one interval are put into one UDP packet for delivery. To avoid packet dropping or fragmentation the packet size must not exceed 4096 byte, while 1024 byte is the recommended maximum size. The binary interface assumes compact representations of state and operations, but therefore operates easily at intervals below 50ms. However, RTSYNC may not be suited for applications that require the server to send updates with an average data rate greater than the preferred datagram size.

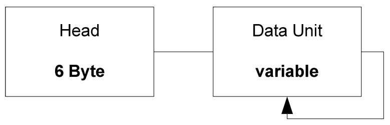


Figure 8-3: RTSYNC message for UDP PDUs

An RTSYNC message for a UDP packet comprises a *Head* and multiple *Data Units* as depicted in Figure 8-3. The head of a pulsar message, as depicted in Figure 8-4, is defined in accordance with the IS3 message head definition. Only the meaning of the *Last Message Interval* field differs slightly. Since a UDP datagram can get lost, the field does not contain the interval of the last received message, but rather the interval of the last message received before loss occurred. The *Reserved* field is a placeholder to obtain a multiple of 8 bits.

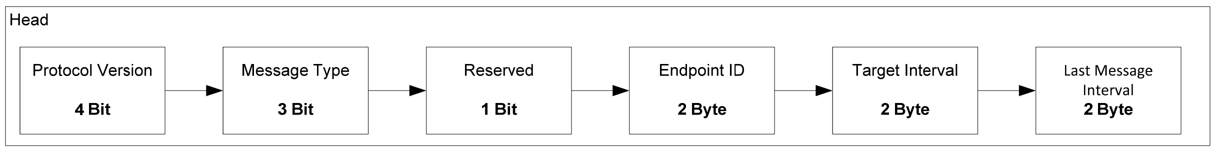


Figure 8-4: RTSYNC message head

In contrast to the IS3 interface definition, RTSYNC does not subdivide message explicitly in parts. The session related part is rather represented by specially flagged Data Units, called Meta Data Units. If the *Type Flag* depicted in Figure 8-5 is not set, the Data Unit contains meta information about the connection. The length of the *Value* field is fixed and defined by the *Unit Code*. A complete list of predefined Unit Codes is appended in Annex B. The RTSYNC message begins with Meta Data Units and is followed by Action Data Units. The latter are characterized by a number of variable-length fields, which are for example used to map operation parameters.

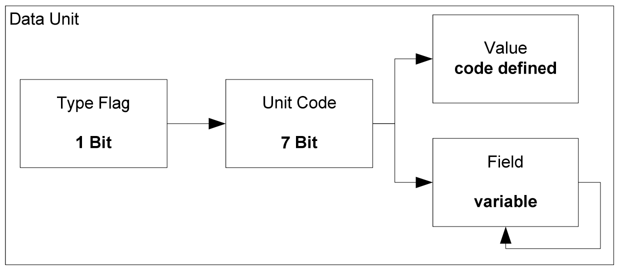


Figure 8-5: RTSYNC message data units

Each field value begins with a *Field Length*. The Field Length has a minimum length of one byte. If the first bit is set, the next byte excluding its first bit is also used to represent the field length. If the first bit of the second byte is also set, additionally the next byte is included and so on. The seven remaining bits of each byte are interpreted as one continuous sequence to obtain the actual length of the *Field Data*. The encoding is big-endian.

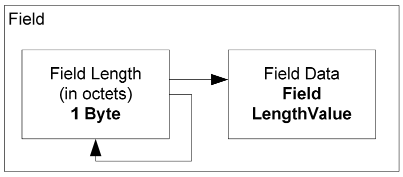


Figure 8-6: RTSYNC message variable-length field coding

All Action Data Unit are grouped by scope and sorted ascending by group size in an RTSYNC message. Although sorting does not have a serious impact within an UDP datagram, it has an impact on the performance in TCP breakout transmission. A serious problem in UDP transmission is loss. Figure 8-7 shows the behavior of client and server if an update is lost. The client recognizes loss when receiving the first successfully transmitted update after one or several losses by the discontinuity of *Target Interval* numbers in the message header. The client notifies the server implicitly by not increasing the value of the *Last Message Interval* header of its next request. The server has to retransmit the lost update immediately. For this purpose the server should store a message provided it was acknowledged by the client.

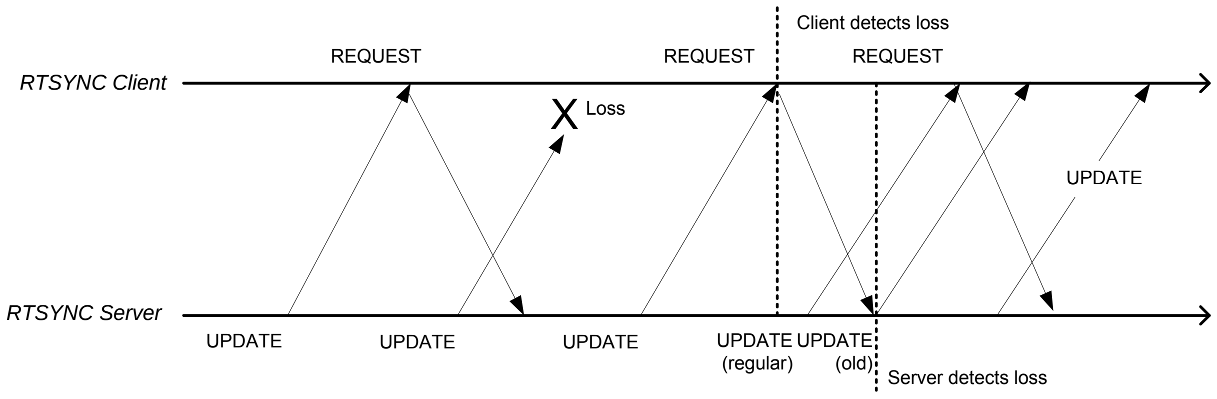


Figure 8-7: RTSYNC lost update

The client must not apply any out of order operations to maintain causal order. Thus, basically the client has to wait with the application of operation from the server until all lost messages have been retransmitted and received successfully. To minimize the need for strict blocking, the scopes are utilized. The scopes of the state are disjoint. Accordingly, two operations of different scopes cannot have a causal relation. If the client knows for which scopes operations were lost, operations of the message following a lost one could be applied. For this purpose, an RTSYNC updates can include several Meta Data Units with the hash keys of the scopes covered by the directly preceding update.

A lost request is first anticipated by the server and signalized with the next update to the client. Like the server, the client retransmits lost requests immediately as depicted in Figure 8-8. The server does not block when a request is lost, instead the server treats the loss as if the client had sent a message without operations. When the late operations are received they are still respected by the server, but their effect is delayed.

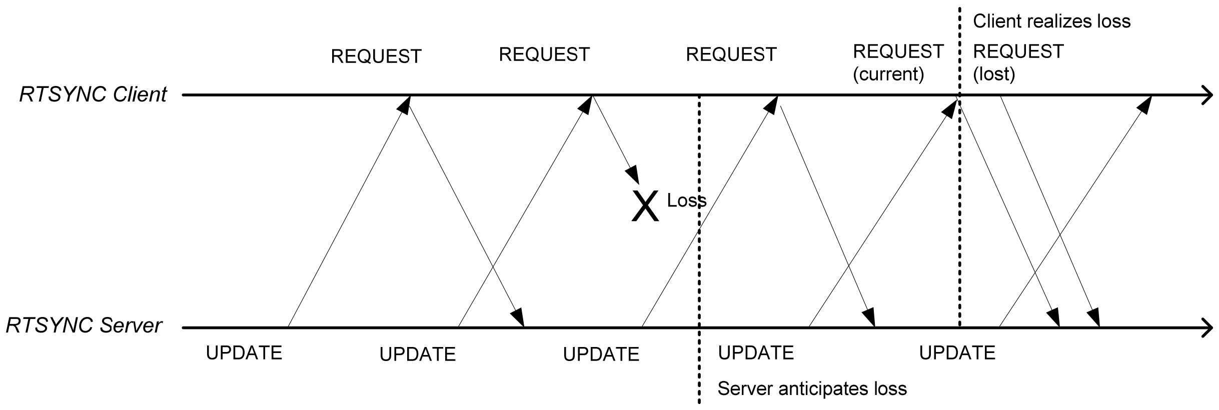


Figure 8-8: RTSYNC lost request

For the server, late and lost requests are difficult to distinguish. The server has to assume a message is lost if it has not yet been received at $\frac{\Delta}{2}$. To minimize the confusion of loss and lateness, the duration T of the interval is adapted dynamically. The initial value, the so called maximum interval length, is successively shortened to a safe interval that still respects maximum jitter values. For the calculation, the roundtrip time of request and update is continuously measured for all clients. If no safe interval length less than the maximum interval length is found for a client, the client is disconnected. The runtime adaptation of play-out delay Δ optimizes the user experienced system behavior. A slow adaptation, as depicted in Figure 8-9, ensures that potential dead reckoning procedures applied on top of Pulsar have sufficient time for adaptation too.

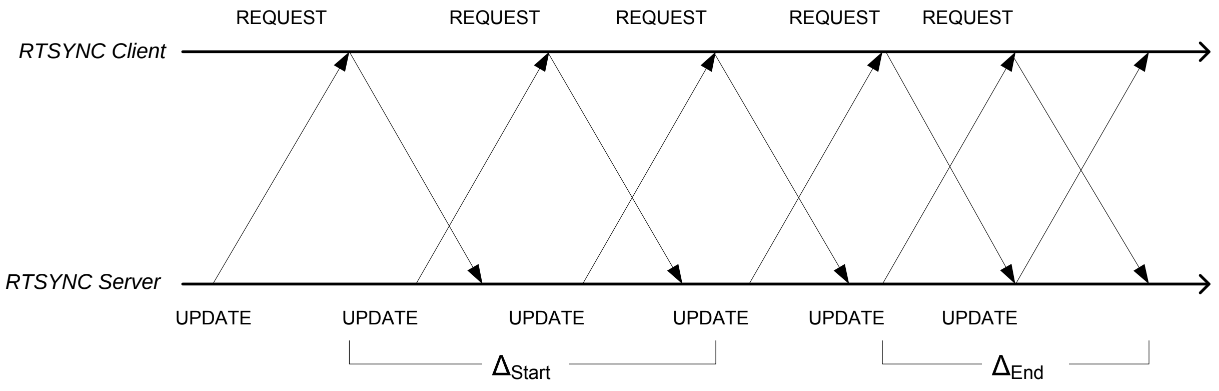


Figure 8-9: RTSYNC interval length adaptation

The limitation of the message size is a problem if a client subscribes for a new scope, such as during session initialization. In this case a serialized subset of the state has to be sent to the client. For this purpose RTSYNC supports a TCP breakout. If the server determines that the serialization of the state exceeds the maximum length of a message, an Action Data Unit is added to the message instead with a breakout instruction, host name, port and the number of Data Units. On receiving the breakout instruction, the client has to temporarily open a TCP connection to the given host and port and resend the header of the message the server answered with the breakout. The server has to reply with an RTSYNC update and the number of previously announced Data Units. Afterwards the client terminates the TCP connection. The message flow is depicted in Figure 8-10.

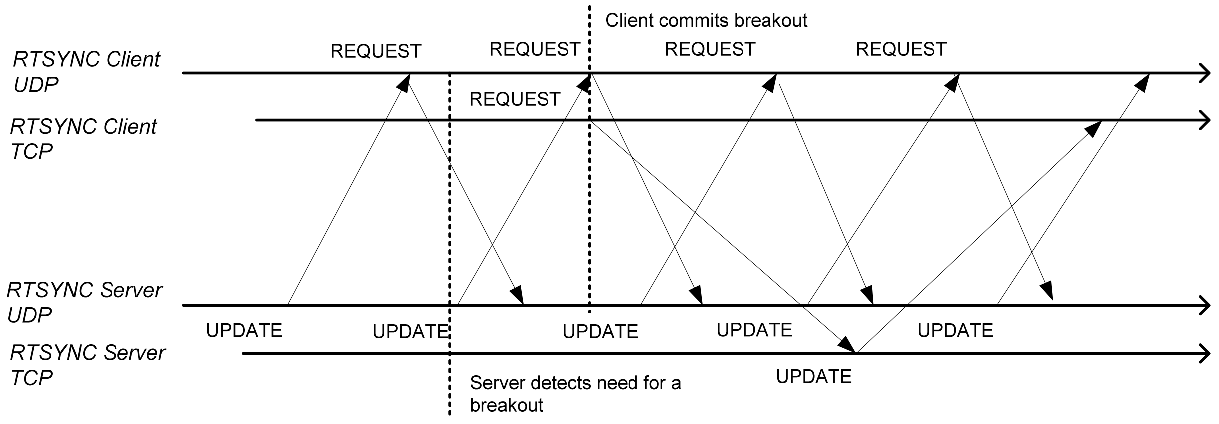


Figure 8-10: RTSYNC TCP breakout

8.3 Server

The Pulsar sever is implemented in Java and can maintain multiple sessions in parallel and is structured accordingly. Each session is controlled by a corresponding session controller that processes the operations connected clients, i.e. transforms them and applies them to the lead state assigned with a session. Additionally the controller manages all users participating in a session as well as their subscriptions for scopes.

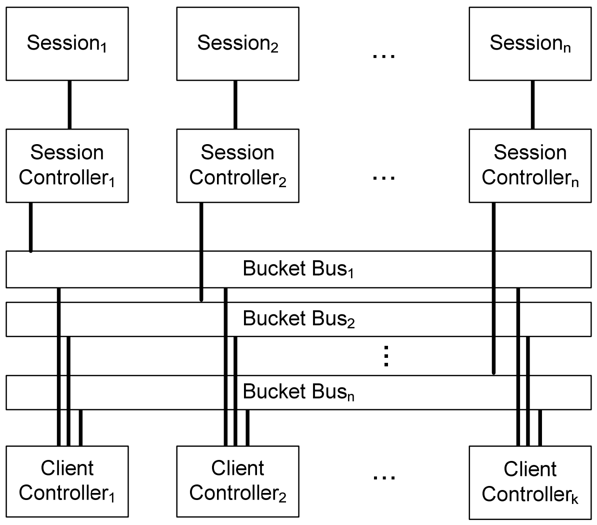


Figure 8-11: Pulsar server overview

Since Pulsar supports multiple client interfaces, client operations for the same session can be received on any of these interfaces. For this reason each session controller is associated with a bucket bus. The bucket bus supports bidirectional communication of operations between a session controller and multiple client controllers. Following the interval-based communication scheme, all operations of the same interval are collected in the same bucket. The bucket bus serves buckets of incoming operations in a queue, while the head element of the queue is the bucket for the next interval. The session controller manages the timing of dequeuing and releasing processed buckets to the client controllers. The client controllers receive and parse messages via the network interfaces from the clients and pass them on to the message bus. They also obtain the ready processed bucket from the busses and serialize and send the transformed operations to the clients. The organization of the basic server element session, session controller, bucket bus and client controller is depicted in Figure 8-11. The following sections introduce the core data structures and system components as well as their interworking.

8.3.1 Data Structures

All components of the Pulsar server are designed for the parsing, serialization, forwarding and processing of state modifying operations. For the realization these operations are organized in two container structures, the *Load* and the *Bucket*. The Load container represents the list of operations per interval and client. Thus, all operations sent within the same interval by the same client are collected in the same Load container in causal order. All operations that have to be sent for one interval to a client are also collected in one Load container in causal order. The causal order is derived from the state *Reference* if it was changed during processing. All Loads provided by clients for one interval and all Loads to be sent to clients in one interval are collected in *Bucket* containers. The Loads of the different clients within one bucket are ordered. This order may vary from interval to interval, i.e. the order with respect to the sender. However, for one interval the order, once determined at the server, must not change within server or at any client.

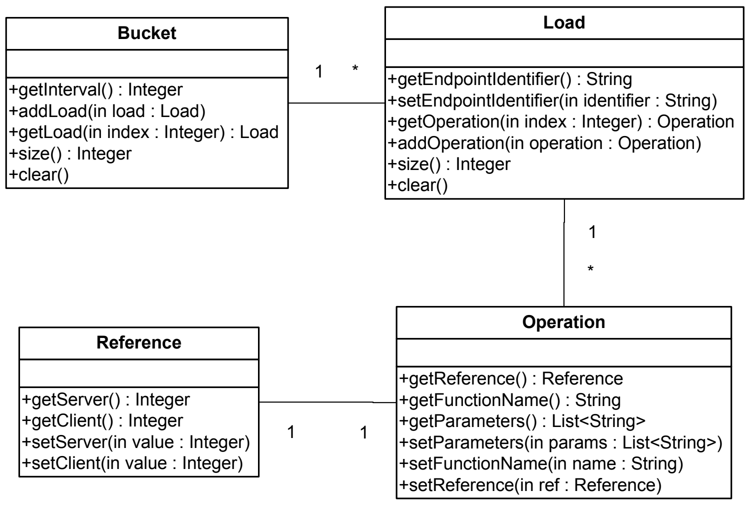

Bucket
+getInterval() : Integer
+addLoad(in load : Load)
+getLoad(in index : Integer) : Load
+size() : Integer
+clear()
1
*
Load
+getEndpointIdentifier() : String
+setEndpointIdentifier(in identifier : String)
+getOperation(in index : Integer) : Operation
+addOperation(in operation : Operation)
+size() : Integer
+clear()
1
*
Reference
+getServer() : Integer
+getClient() : Integer
+setServer(in value : Integer)
+setClient(in value : Integer)
1
1
Operation
+getReference() : Reference
+getFunctionName() : String
+getParameters() : List<String>
+setParameters(in params : List<String>)
+setFunctionName(in name : String)
+setReference(in ref : Reference)

Figure 8-12: UML class diagram for sata structures in Pulsar server realization

8.3.2 Session Management

The core of the Pulsar server implementation, as depicted in Figure 8-13, is represented by the above introduced *SessionController*, the *SessionBucketBus*, the *ClientController,* and an additional *SystemController*. All these components follow a bipartite interface concept of basic and extended interfaces. The system view granted by basic interfaces is intended for clients and their delegate, the ClientController. The system view granted by the extended interfaces addresses management access, such as required by the WHA Server. The SystemController is the central entry point to create, control and shutdown sessions. This controller is configured by the *SystemConfiguration*, which basically contains a list of ClientControllers. The actual access to session state and management data is provided by the SessionController.

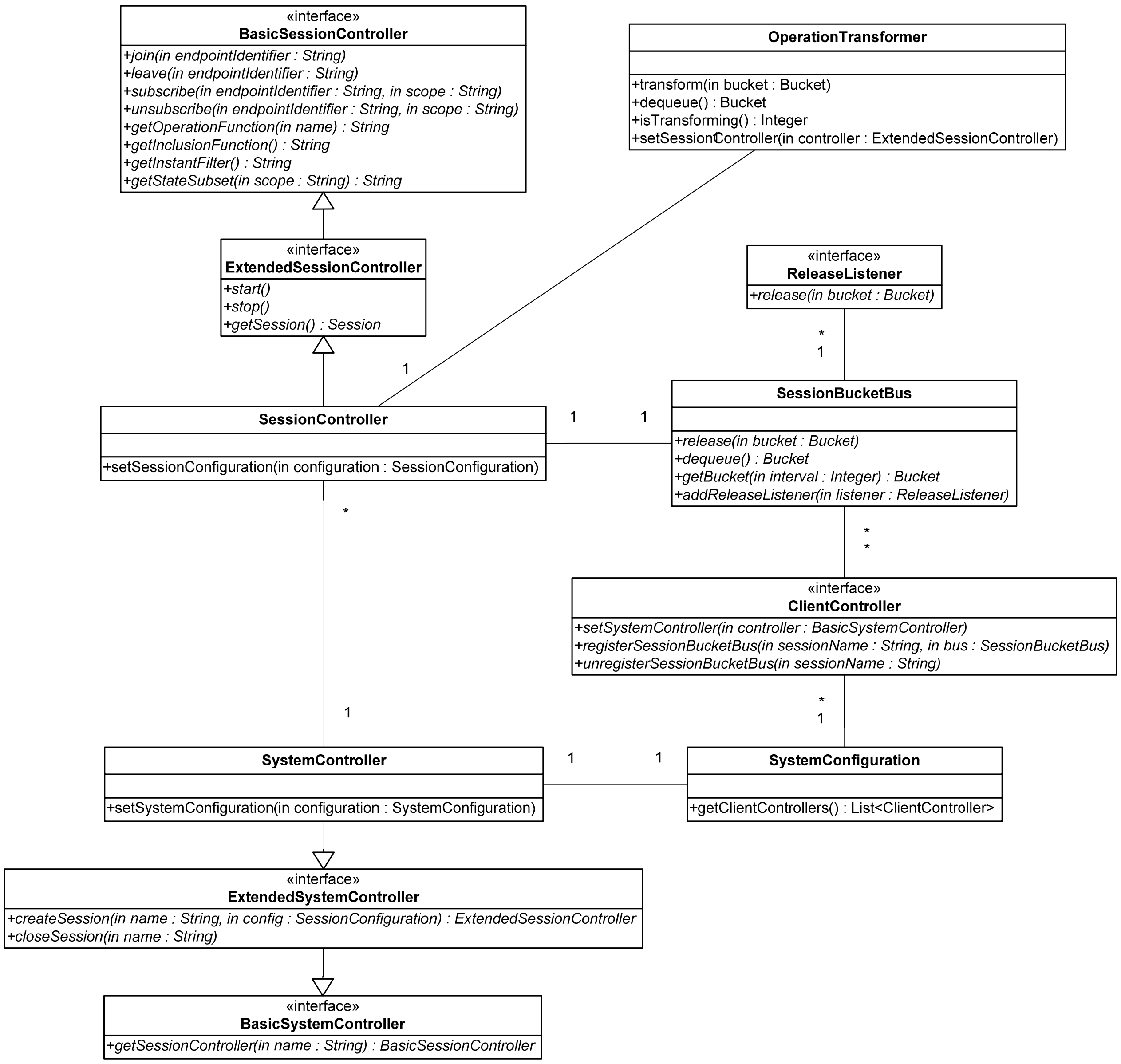

«interface»
BasicSessionController
+join(in endpointIdentifier : String)
+leave(in endpointIdentifier : String)
+subscribe(in endpointIdentifier : String, in scope : String)
+unsubscribe(in endpointIdentifier : String, in scope : String)
+getOperationFunction(in name) : String
+getInclusionFunction() : String
+getInstantFilter() : String
+getStateSubset(in scope : String) : String
OperationTransformer
+transform(in bucket : Bucket)
+dequeue() : Bucket
+isTransforming() : Integer
+setSessionController(in controller : ExtendedSessionController)
«interface»
ExtendedSessionController
+start()
+stop()
+getSession() : Session
«interface»
ReleaseListener
+release(in bucket : Bucket)
SessionBucketBus
+release(in bucket : Bucket)
+dequeue() : Bucket
+getBucket(in interval : Integer) : Bucket
+addReleaseListener(in listener : ReleaseListener)
SessionController
+setSessionConfiguration(in configuration : SessionConfiguration)
«interface»
ClientController
+setSystemController(in controller : BasicSystemController)
+registerSessionBucketBus(in sessionName : String, in bus : SessionBucketBus)
+unregisterSessionBucketBus(in sessionName : String)
SystemController
+setSystemConfiguration(in configuration : SystemConfiguration)
SystemConfiguration
+getClientControllers() : List<ClientController>
«interface»
ExtendedSystemController
+createSession(in name : String, in config : SessionConfiguration) : ExtendedSessionController
+closeSession(in name : String)
«interface»
BasicSystemController
+getSessionController(in name : String) : BasicSessionController
1
*
1
1
1
*
*
*
*
1
1
1
1
*
1

Figure 8-13: UML class diagram for Pulsar server core components

Each SessionController is also associated with an *OperationTransformer*. The OperationTransformer processes the inclusion of client operations and temporarily stores the transformation results. The result of each transformation has to be stored in case a client load is received that is still concurrent with the results of the previous transformation. This load would require transformation into the current state before it could be transformed against concurrent loads. Actually, if the pipeline depth of the communication scheme is greater than zero this is the default case for all loads. Like the SessionBucketBus the OperationTansformer organizes processed buckets of outgoing operations in a queue. The SessionController reads from this queue whenever the time is due to provide the clients with the operations for the next interval.

8.3.3 Custom Configuration

When a new session is created, an appropriate configuration has to be provided that defines the parameters for system operation. The most important elements of the system configuration are the various functions for modifying states and transforming operations. These functions are provided during the creation of a session with the *SessionConfiguration.* Each SessionConfiguration is associated with a list of *Scope* instances. The Scope component manages client subscriptions to scopes in the state and provides a function to check if an operation affects a scope of the state or not. This check has to be implemented as *OperationFilter* The *Session* serves at runtime as door to SessionConfiguration and *Resource* for the SessionController. Additionally it manages the current list of clients connected to a session.

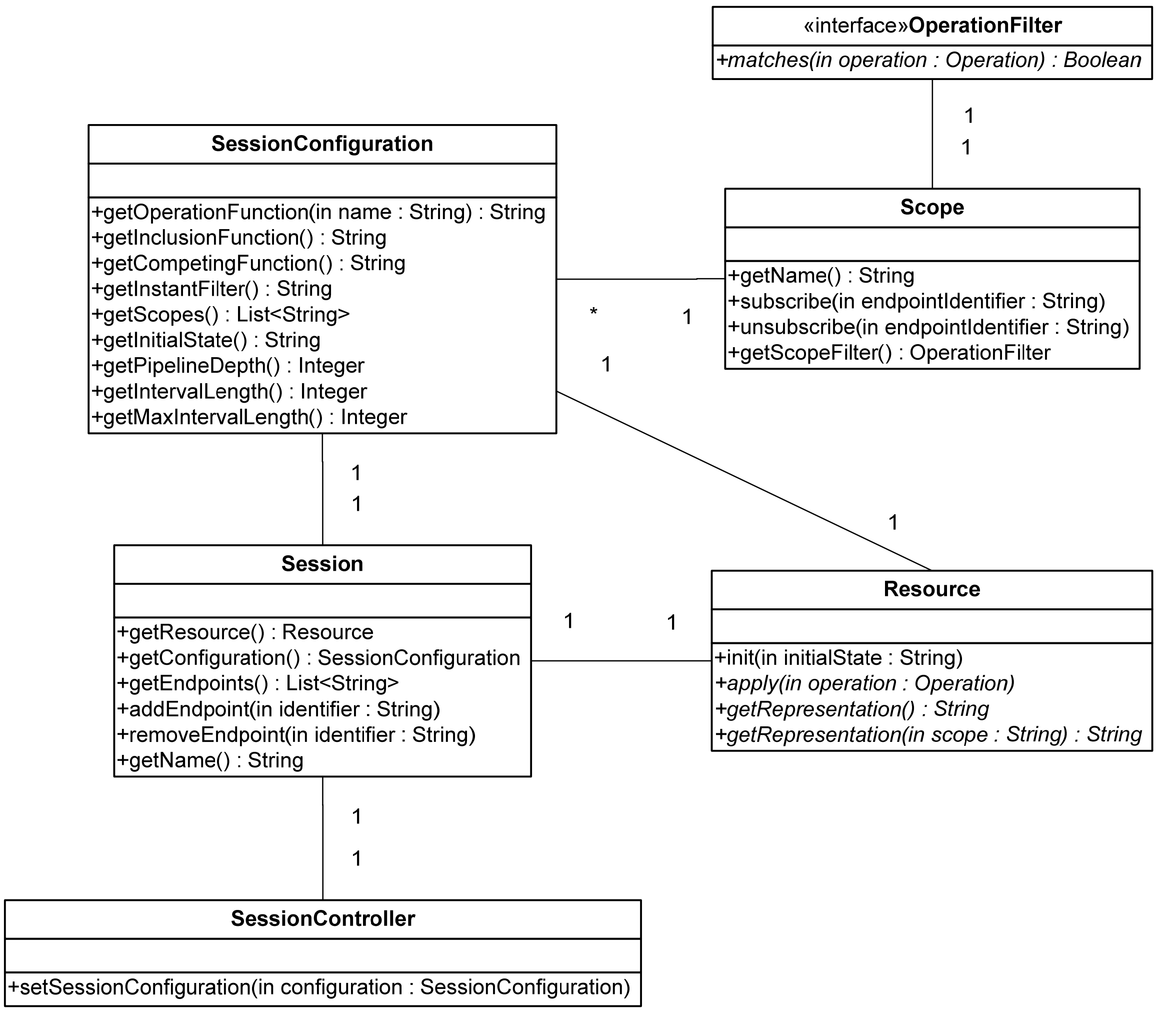

«interface»OperationFilter
+matches(in operation : Operation) : Boolean
1
1
SessionConfiguration
+getOperationFunction(in name : String) : String
+getInclusionFunction() : String
+getCompetingFunction() : String
+getInstantFilter() : String
+getScopes() : List<String>
+getInitialState() : String
+getPipelineDepth() : Integer
+getIntervalLength() : Integer
+getMaxIntervalLength() : Integer
*
1
1
Scope
+getName() : String
+subscribe(in endpointIdentifier : String)
+unsubscribe(in endpointIdentifier : String)
+getScopeFilter() : OperationFilter
1
1
1
Session
+getResource() : Resource
+getConfiguration() : SessionConfiguration
+getEndpoints() : List<String>
+addEndpoint(in identifier : String)
+removeEndpoint(in identifier : String)
+getName() : String
1
1
Resource
+init(in initialState : String)
+apply(in operation : Operation)
+getRepresentation() : String
+getRepresentation(in scope : String) : String
1
1
SessionController
+setSessionConfiguration(in configuration : SessionConfiguration)

Figure 8-14: UML class diagram of custom session configuration

Pulsar is designed independently of state representation and operation functions. Appropriate implementations have to be provided with the session configuration. For the utilization in a WHA setup, these implementations have to be represented in ECMA 262 [92] compliant syntax. During session initialization, the functions can thus be marshaled to the clients and executed without further translation or interpretation. Within the Pulsar server, the functions are represented as strings. When a session is created these functions are interpreted and checked for correctness. If their correctness is proven they are compiled for later usage. Just-in-time compilation prevents the server from interpreting the function each time they have to be called. The server expects the strings to contain JS functions implementing the interfaces listed below. The listings are given in WebIDL syntax [93]. If the functions and entities in the strings returned by the corresponding configuration methods do not match the interfaces, a session will not be created. The interface of an operation described in Interface 8-1 is not utilized by the server implementation directly, but by the server to make sure that any operation will match the signature.

```
 [NoInterfaceObject]
interface Operation {
    attribute DOMString type;
    attribute DOMString[] parameters
    attribute DOMString origin
}
```

Interface 8-1: Operation

A state representation is provided as the Resource depicted in Figure 8-14. The code of the returned string has to be a function call (e.g. a closure), which returns an implementation for Interface 8-2. At the client, this entity is used to optimistically update a representation of the resource state (i.e. a replication) and read locally from this state.

```
interface Respresentation {
    DOMString read(DOMString name);
    void apply(in Operation op);
    void addSubset(DOMString scope, DOMString representation);
};
```

Interface 8-2: Handle to a copy of the resource state

The three parts of the session configuration provided by the methods getInclusionFunction, getCompetingFunction and getInstantFilter documented in Figure 8-14 have to match the signatures summarized in Interface 8-3, Interface 8-4 and Interface 8-5. Again the code returned by the methods has to implement a function call statement which returns, once executed, a function with the required interface.

```
interface Inclusion{
    Operation transform(Operation op1, Operation op2);
}
```

Interface 8-3: Inclusion function

```
interface Competition{
    boolean competing(Operation op1, Operation op2);
}
```

Interface 8-4: Function to analyze relation of two operations

```
interface Instant{
    boolean instant(Operation op);
}
```

Interface 8-5: Schema function to check of an operation can be applied instantly

8.3.4 Component Interworking

A Pulsar session is created with the help of the SystemController. The SystemController creates a SessionController, a SessionBucketBus and assigns the bus to all ClientControllers. Also via the SystemController a ClientController can obtain any SessionController by citing the name of the session. The session name can be chosen free, but must be unique on the server. A ClientController requires access to the SessionController to add new clients to a Session and forward their subscriptions to Scopes as depicted in Figure 8-15.

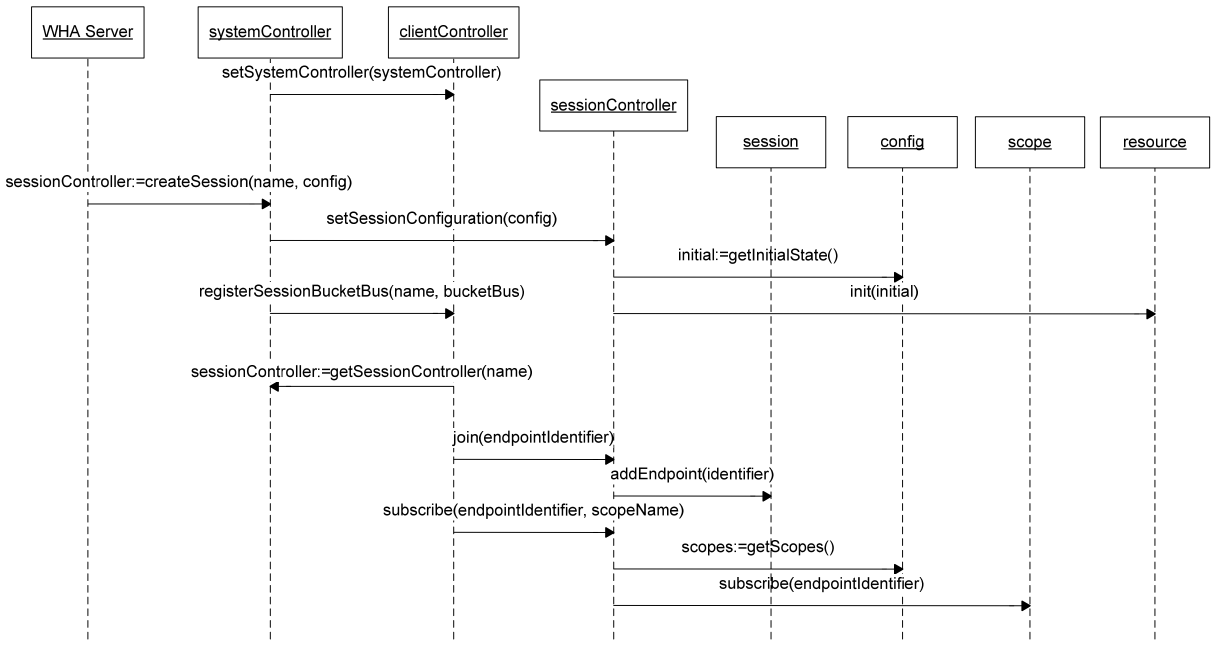

WHA Server
systemController
clientController
sessionController
session
config
scope
resource
setSystemController(systemController)
sessionController:=createSession(name, config)
setSessionConfiguration(config)
initial:=getInitialState()
registerSessionBucketBus(name, bucketBus)
init(initial)
sessionController:=getSessionController(name)
join(endpointIdentifier)
addEndpoint(identifier)
subscribe(endpointIdentifier, scopeName)
scopes:=getScopes()
subscribe(endpointIdentifier)

Figure 8-15: Simplified UML sequence diagram for session initialization

The regular interworking of the core components is depicted in Figure 8-16. When a ClientController receives a Load of operations from a client, the controller looks up the SessionBucketBus for the corresponding Session and the Bucket for the corresponding interval. The Load is added to the Bucket. This process may be repeated for one Bucket until each client on a session has delivered its Load. The Bucket remains in the queue of the bus until its interval is due for processing. At this time the SessionController dequeues the Bucket and forwards it for transformation to the OperationTransformer. The OperationTransformer retrieves all Loads, transforms them and enqueues the Bucket with the results again until the SessionController decides to forward the Bucket to the clients. Usually a Bucket is immediately released to the ClientControllers without further queuing.

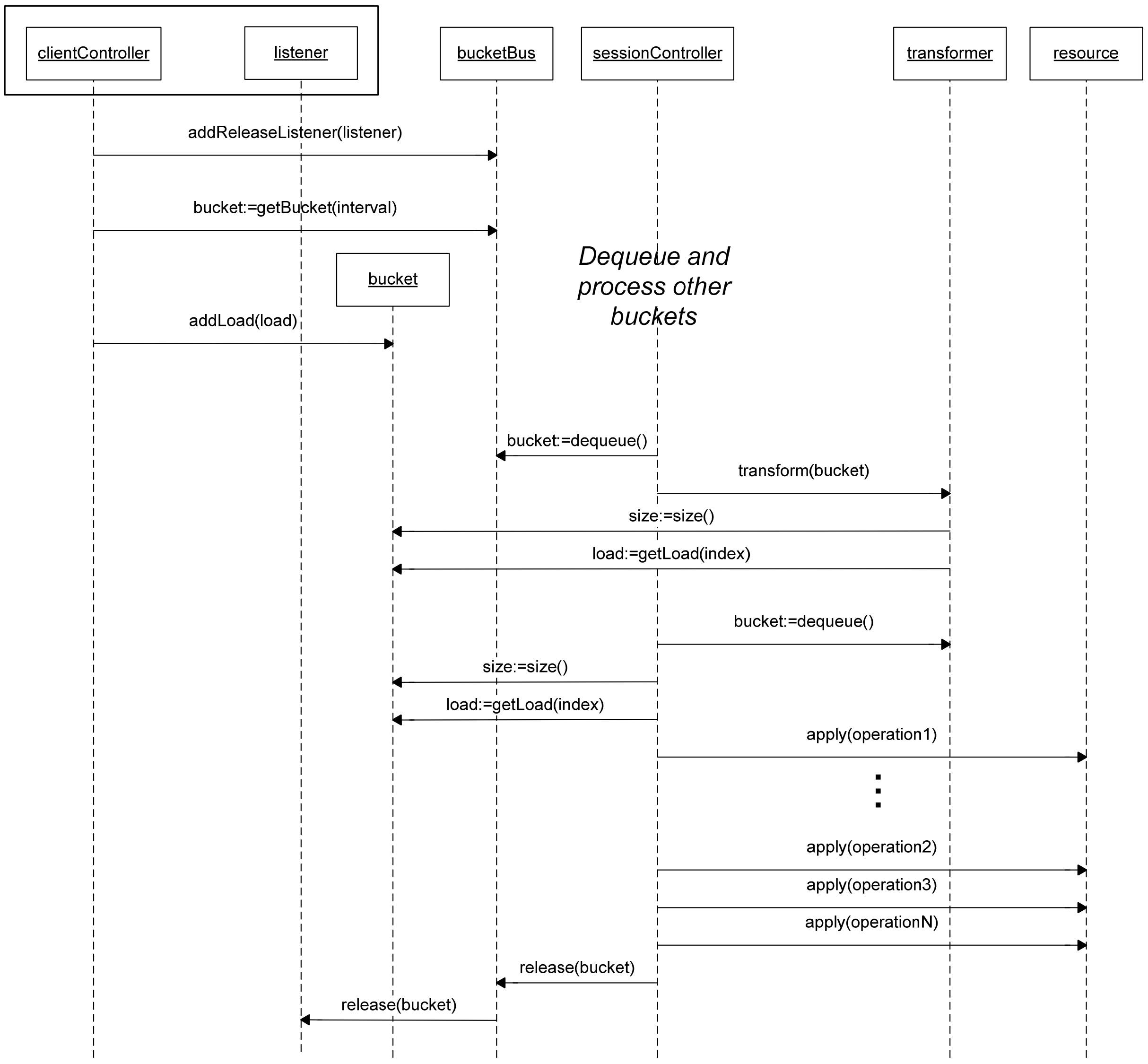


Figure 8-16: Simplified UML sequence diagram for regular component interworking in Pulsar server

8.4 Client

The Pulsar client realizes the minimal stub of the Pulsar system required in the application model of the WHA client. This stub comprises three major components, *Controller*, *Session* and *Server Adapter* as depicted in Figure 8-17. The Server Adapter is the gatekeeper to the Pulsar Server. Corresponding to the network interfaces, two Server Adapters are implemented for Pulsar, one for text-based messages and one for binary messages. The adapter used has to be determined when a WHA client joins a session. One client must not be connected to the same session via two or more Server Adapters, but each Server Adapter can serve several session connections. The Session component maintains the state and processes operations, i.e. transforms them against local, concurrent operations. In contrast to the server algorithm, the client-side transformation is a simple, sequential inclusion as explained in section 5.6. The Pulsar client can maintain multiple connections to sessions in parallel. These connections may share the same Server Adapters. The Controller manages the sessions per endpoint. Each WHA client may define multiple endpoints, e.g. for different widgets. Controller, Sessions and state representations are accessed through the Pulsar Application Programmer Interface (API). In the following sections the design of session management is described in detail, as well as the API to access a session in Pulsar. For this version of Pulsar, two Server Adapters are implemented, one for the text-based network interface and one for the binary network interface. While the first Server Adapter simply bases on a single JS prototype, the realization of the binary interface with browser extensions comprises a number of classes for different execution contexts. Therefore, the implementation of the Server Adapter for the binary network interface, also called RTSYNC Server Adapter, is explicitly described in section 8.4.2.

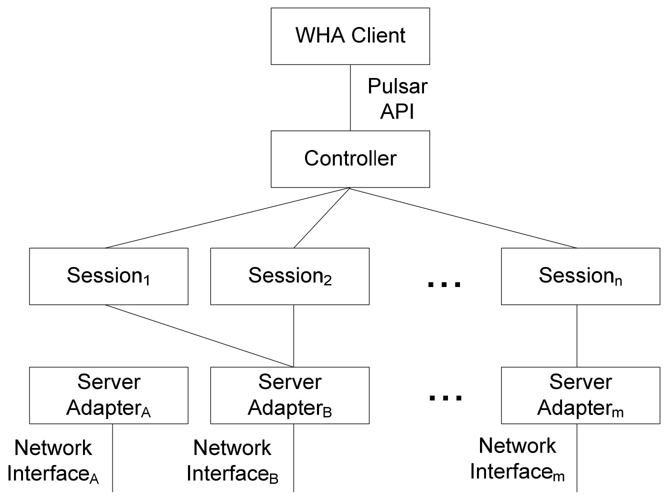


Figure 8-17: Overview of Pulsar client architecture

8.4.1 Session Management

The implementation of the session management comprises a number of JS prototypes, for simplicity summarized as classes on the UML diagram in Figure 8-18. The entrance to the session management is the *SystemController*. The *EndpointController* provides functionality to join sessions and manage all open sessions. One client can represent multiple endpoints. This peculiarity of the design addresses mashed web pages and portals that integrate third party content. Each EndpointController is associated with a number of *ServerAdapter*s. The *ServerAdapter* has to implement the communication scheme, i.e. generation, parsing and processing of messages exchanged between client and server. From the viewpoint of the *Session*, the *ServerAdapter* hides the bucket-based transmission of operations. In contrast, a publish/subscribe methodology is provided to the *Session*. The *Session* represents the delegate between *ServerAdapter*, *State* and WHA client. It makes sure operations are not only applied to the local copy of the resource state reflected in the *Replica*, but also sent to the server. Moreover, operations received from the server are transformed against local, concurrent operations by the *Session* if required.

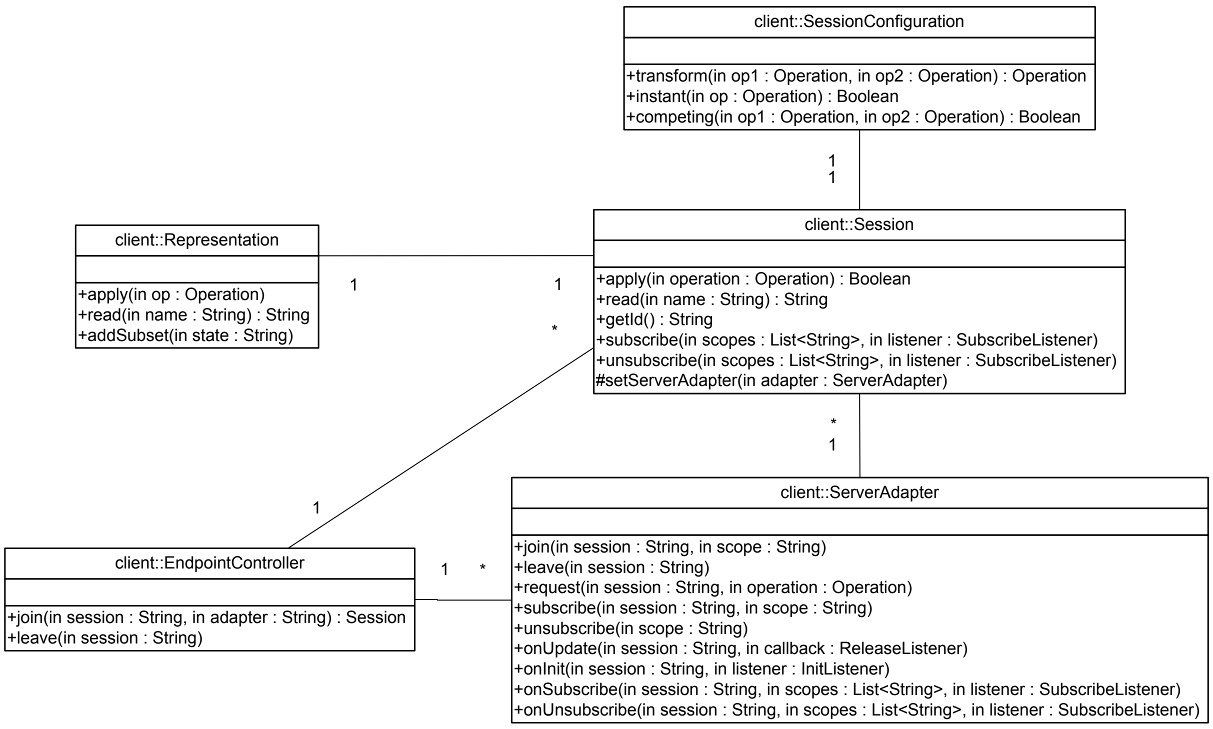

client::SessionConfiguration
+transform(in op1 : Operation, in op2 : Operation) : Operation
+instant(in op : Operation) : Boolean
+competing(in op1 : Operation, in op2 : Operation) : Boolean
1
1
client::Representation
+apply(in op : Operation)
+read(in name : String) : String
+addSubset(in state : String)
1
1
*
client::Session
+apply(in operation : Operation) : Boolean
+read(in name : String) : String
+getId() : String
+subscribe(in scopes : List<String>, in listener : SubscribeListener)
+unsubscribe(in scopes : List<String>, in listener : SubscribeListener)
#setServerAdapter(in adapter : ServerAdapter)
*
1
1
client::EndpointController
+join(in session : String, in adapter : String) : Session
+leave(in session : String)
1
*
client::ServerAdapter
+join(in session : String, in scope : String)
+leave(in session : String)
+request(in session : String, in operation : Operation)
+subscribe(in session : String, in scope : String)
+unsubscribe(in scope : String)
+onUpdate(in session : String, in callback : ReleaseListener)
+onInit(in session : String, in listener : InitListener)
+onSubscribe(in session : String, in scopes : List<String>, in listener : SubscribeListener)
+onUnsubscribe(in session : String, in scopes : List<String>, in listener : SubscribeListener)

Figure 8-18: UML class diagram of client session management

The *SessionConfiguration* is a container for all functions provided during session initialization. The functions are assigned at runtime to the *SessionConfiguration* instance associated with a *Session*. Also the Replica is instantiated from the state representation obtained from the server during initialization. The initialization process is depicted in Figure 8-19. When a WHA client joins a session, not only the session name has to be provided but also the *ServerAdapter* to be used for the connection. The *SystemController* creates a new session and registers the *ServerAdapter* to the session. On creation, the session subscribes at the *ServerAdapter* for various events, such as the provision of the session configuration by the server, the reception of operations or the availability of the current state for a recently subscribed scope. If callbacks for all possibly relevant events are subscribed, the *Session* instance initiates the join. Joining a session already requires naming at least one scope. In the Pulsar implementation a fixed initial scope called *global* is introduced.

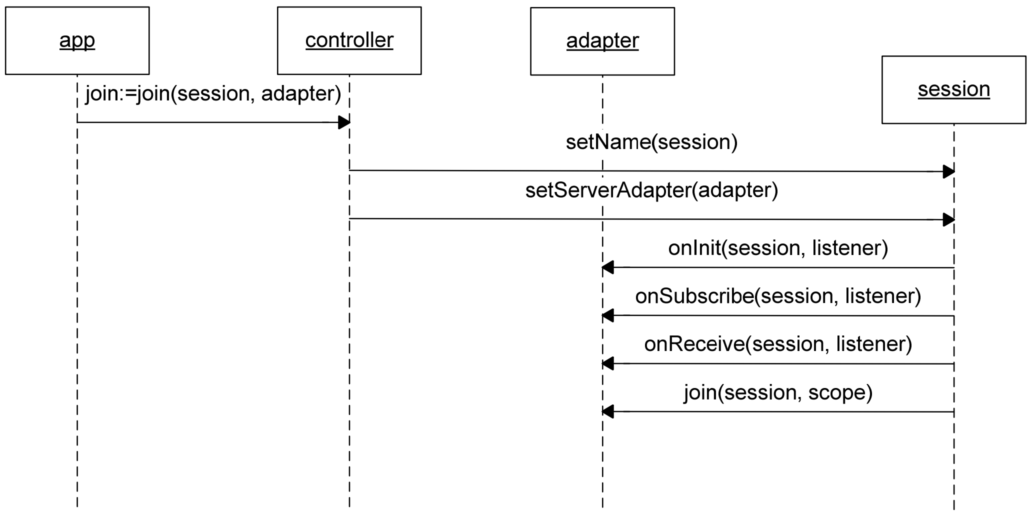

app
controller
adapter
session
join:=join(session, adapter)
setName(session)
setServerAdapter(adapter)
onInit(session, listener)
onSubscribe(session, listener)
onReceive(session, listener)
join(session, scope)

Figure 8-19: UML sequence diagram of client session setup

8.4.2 RTSYNC Server Adapter

The RTSYNC Server Adapter is implemented for the WHA client application model. Processing of binary messages and access to the network interface of the user agent host are not supported in the standardized JavaScript Runtime Environment (JSRE). For this reason parts of the RTSYNC Server Adapter must be realized in the form of a Web Browser extension or plug-in. The Server Adapter interacts with the Pulsar server and maintains the state of all sessions. Additionally it realizes the interface to the client session management.

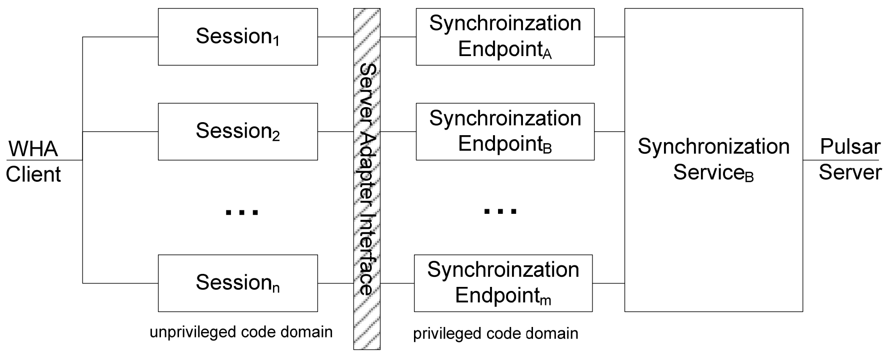


Figure 8-20: RTSYNC Server Adapter Overview

Figure 8-20 shows the reference architecture of the client component, including the two major subsystems *Synchronization Service* and *Synchronization Endpoint*. The central Synchronization Service communicates with the server. It receives incoming messages, parses them partially and delivers them to the corresponding endpoints. For each synchronization connection, a separate Synchronization Endpoint object is created. Endpoints maintain the session state. Incoming update messages are parsed and forwarded to the session management. All modifications to the state during one interval are collected as load for the same bucket. At the end of an interval, requests are summarized and issued to the server.

The need for a Web Browser extension and the very different extension interfaces of the various products makes it necessary to either implement an extension for each product or to support one product only. Since the RTSYNC network interface was primarily realized for the experimental validation of an efficient IS3 realization, the second option was chosen here. The RTSYNC Server Adapter is implemented for the Mozilla Firefox Web Browser (Firefox). In addition to the legacy Netscape Plugin Application Programming Interface (NPAPI) Firefox supports proprietary browser extensions. In contrast to common plug-ins, an extension can access a multitude of browser subsystems and is also the primary means for extending the browser core functionality.

For the interaction between the diverse subsystems the Cross Platform Component Object Model (XPCOM) is utilized. XPCOM is a cross platform component model from Mozilla, similar to the Common Object Request Broker Architecture (CORBA) or Microsoft Component Object Model (COM). The functionality of the Mozilla platform is organized in a number of individual components, which are easily reusable. Components may be developed in multiple programming languages and are accessed through properly defined interfaces. This interoperability of components with code bases in different languages also enables extensions to access each component of the Browser from JS code if the required permissions are granted.

Since many internal components of Firefox are implemented with JS, the JSRE distinguishes privileged code from unprivileged code. Depending on the context of the code, it is executed with different rights. Privileged code and unprivileged codes are summarized in two code contexts, Chrome and Content. Browser internal code is executed in the Chrome context. Execution of this code is considered safe without limiting privileges. Web site embedded scripts, or separate script content, loaded from external sources is considered unsafe and potentially dangerous. This kind of code is executed with limited privileges, i.e. in the code context Content.

The support of Firefox for cross-language realization of extensions allows implementation of the RTSYNC Server Adapter in C++ and JS. The Synchronization Service is implemented in C++ for fast message parsing and the support for network communication. The service is a singleton responsible for the multiplexing of messages, allowing multiple synchronization connections to share a single datagram socket. Particular tasks of the service are organized in runnable classes to enable straightforward dispatching to other threads. These tasks include listening on a UDP socket for incoming messages, processing update messages (also called sync messages in the implementation) by parsing their header, and dispatching them by means of the Endpoint Identifier in the header to their destination Synchronization Endpoint. Additionally, request messages are constructed from client synchronization requests and sent to the server. If an update contains a breakout instruction, the Synchronization service also establishes the required TCP connection.

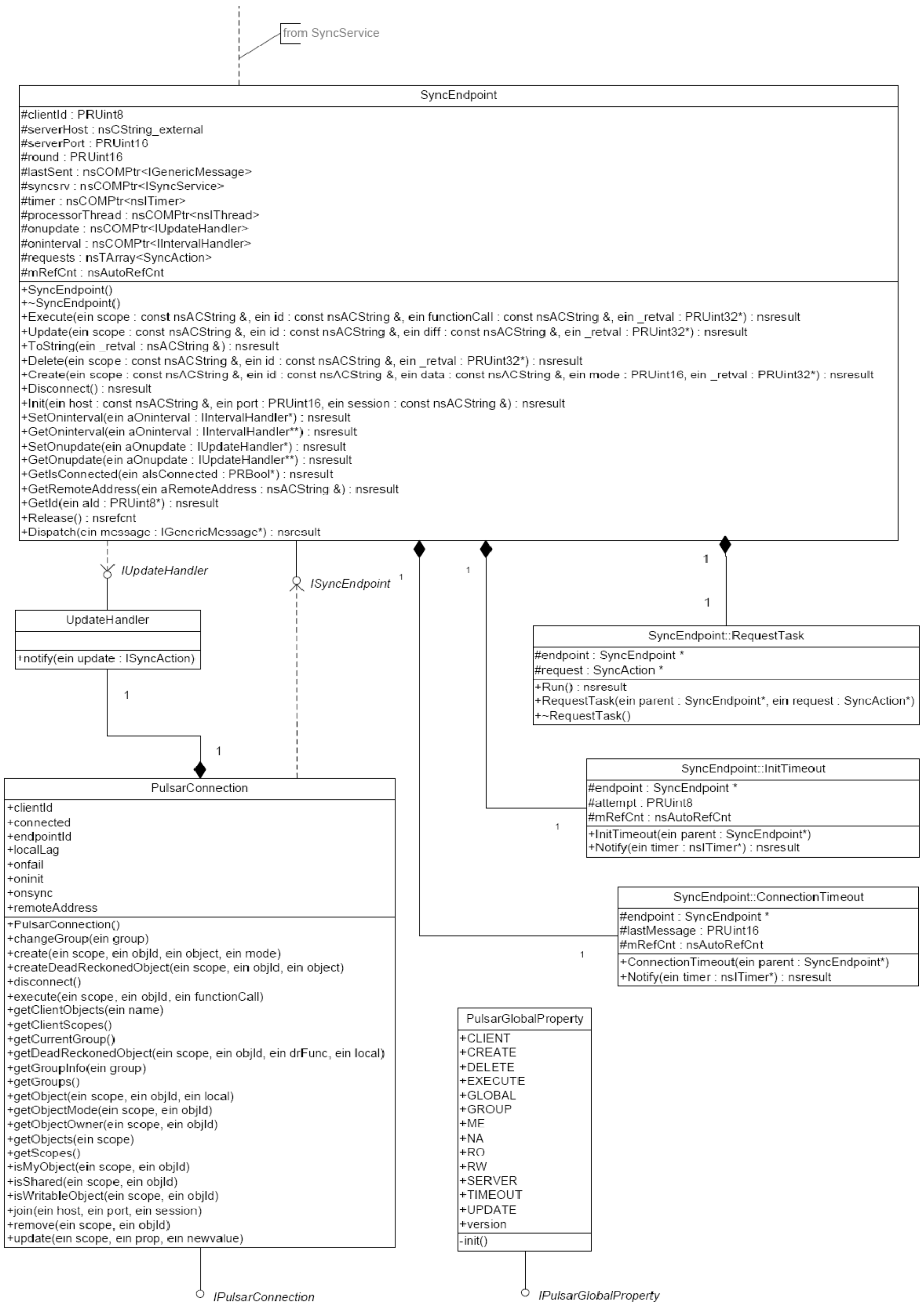

from SyncService
SyncEndpoint
#clientId : PRUint8
#serverHost : nsCString_external
#serverPort : PRUint16
#round : PRUint16
#lastSent : nsCOMPtr<IGenericMessage>
#syncsrv : nsCOMPtr<ISyncService>
#timer : nsCOMPtr<nsITimer>
#processorThread : nsCOMPtr<nsIThread>
#onupdate : nsCOMPtr<IUpdateHandler>
#oninterval : nsCOMPtr<IIntervalHandler>
#requests : nsTArray<SyncAction>
#mRefCnt : nsAutoRefCnt
+SyncEndpoint()
+~SyncEndpoint()
+Execute(ein scope : const nsACString &, ein id : const nsACString &, ein functionCall : const nsACString &, ein _retval : PRUint32*) : nsresult
+Update(ein scope : const nsACString &, ein id : const nsACString &, ein diff : const nsACString &, ein _retval : PRUint32*) : nsresult
+ToString(ein _retval : nsACString &) : nsresult
+Delete(ein scope : const nsACString &, ein id : const nsACString &, ein _retval : PRUint32*) : nsresult
+Create(ein scope : const nsACString &, ein id : const nsACString &, ein data : const nsACString &, ein mode : PRUint16, ein _retval : PRUint32*) : nsresult
+Disconnect() : nsresult
+Init(ein host : const nsACString &, ein port : PRUint16, ein session : const nsACString &) : nsresult
+SetOninterval(ein aOninterval : IIntervalHandler*) : nsresult
+GetOninterval(ein aOninterval : IIntervalHandler**) : nsresult
+SetOnupdate(ein aOnupdate : IUpdateHandler*) : nsresult
+GetOnupdate(ein aOnupdate : IUpdateHandler**) : nsresult
+GetIsConnected(ein aIsConnected : PRBool*) : nsresult
+GetRemoteAddress(ein aRemoteAddress : nsACString &) : nsresult
+GetId(ein aId : PRUint8*) : nsresult
+Release() : nsrefcnt
+Dispatch(ein message : IGenericMessage*) : nsresult
IUpdateHandler
ISyncEndpoint
1
1
1
1
UpdateHandler
+notify(ein update : ISyncAction)
1
1
SyncEndpoint::RequestTask
#endpoint : SyncEndpoint *
#request : SyncAction *
+Run() : nsresult
+RequestTask(ein parent : SyncEndpoint*, ein request : SyncAction*)
+~RequestTask()
SyncEndpoint::InitTimeout
#endpoint : SyncEndpoint *
#attempt : PRUint8
#mRefCnt : nsAutoRefCnt
+InitTimeout(ein parent : SyncEndpoint*)
+Notify(ein timer : nsITimer*) : nsresult
1
SyncEndpoint::ConnectionTimeout
#endpoint : SyncEndpoint *
#lastMessage : PRUint16
#mRefCnt : nsAutoRefCnt
+ConnectionTimeout(ein parent : SyncEndpoint*)
+Notify(ein timer : nsITimer*) : nsresult
1
PulsarConnection
+clientId
+connected
+endpointId
+localLag
+onfail
+oninit
+onsync
+remoteAddress
+PulsarConnection()
+changeGroup(ein group)
+create(ein scope, ein objId, ein object, ein mode)
+createDeadReckonedObject(ein scope, ein objId, ein object)
+disconnect()
+execute(ein scope, ein objId, ein functionCall)
+getClientObjects(ein name)
+getClientScopes()
+getCurrentGroup()
+getDeadReckonedObject(ein scope, ein objId, ein drFunc, ein local)
+getGroupInfo(ein group)
+getGroups()
+getObject(ein scope, ein objId, ein local)
+getObjectMode(ein scope, ein objId)
+getObjectOwner(ein scope, ein objId)
+getObjects(ein scope)
+getScopes()
+isMyObject(ein scope, ein objId)
+isShared(ein scope, ein objId)
+isWritableObject(ein scope, ein objId)
+join(ein host, ein port, ein session)
+remove(ein scope, ein objId)
+update(ein scope, ein prop, ein newvalue)
PulsarGlobalProperty
+CLIENT
+CREATE
+DELETE
+EXECUTE
+GLOBAL
+GROUP
+ME
+NA
+RO
+RW
+SERVER
+TIMEOUT
+UPDATE
+version
-init()
IPulsarConnection
IPulsarGlobalProperty

Figure 8-21: UML diagram of synchronization endpoint in the RTSYNC server adapter

The implementation of the Synchronization Endpoint is separated in two distinct components connected through XPCOM, *SyncEndpoint* and *PulsarConnection* depicted in Figure 8-21. The separation helps to optimize networking and message processing on the one hand, and data exchange to the WHA client execution environment (i.e. JSRE for Content code context) on the other.

The SyncEndpoint connection to the Synchronization Service is implemented entirely in C++ to satisfy performance requirements. It benefits from the fast execution of compiled code. Thus, this component should manage as many tasks as possible and is responsible for all performance dependent tasks, such as processing and parsing of messages. Requests are stored in this part and request messages are constructed and dispatched when update messages are received. The incoming state updates are passed to the JavaScript component, which realizes the connection to the session management. Timers help to determine message loss and anticipate the moment of the next update from the server. The PulsarConnection implements the functionality required by the Session component in the session management, such as event subscription, scope subscription and session setup. Moreover, operations are encoded to and decoded from an appropriate JavaScript representation by this component.

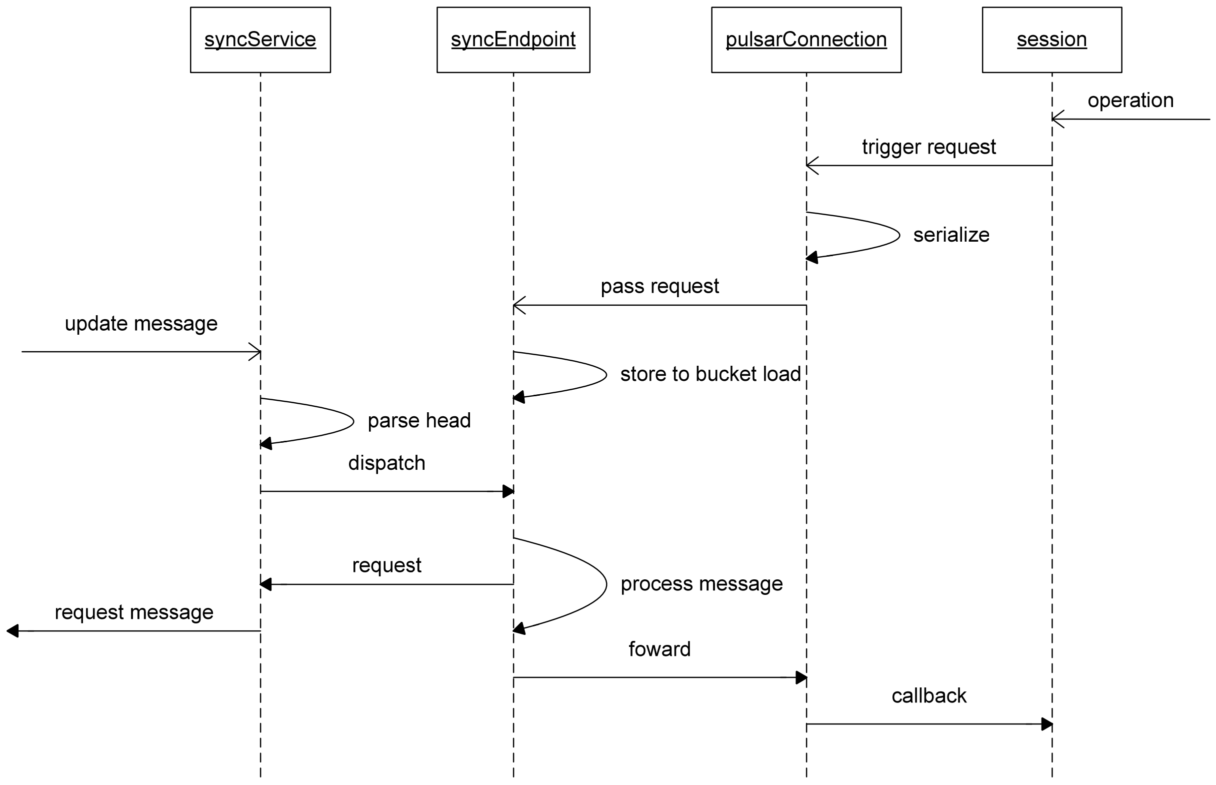


Figure 8-22: Rough UML sequence diagram of RTSYNC client component interaction

The interaction between components is described by means of typical processes. Figure 8-22 depicts a sequence diagram for the interaction of components. Two parallel processes are illustrated: i) an operation request resulting from a user input in the WHA client, ii) an incoming message from the server.

The user request is reported to *PulsarConnection* and passed to the C++ *SyncEndpoint*. The latter collects all operation requests and stores them until the next update is received from the server. The incoming update message from the server is partially parsed and forwarded to the targeted endpoint. In the SyncEndpoint, the message head is processed and the message replied with a request message, which is immediately passed back to the SyncService. Then the message body is parsed and forwarded to the PulsarConnection. Finally the Session is notified through a previously subscribed operation callback function.

The interface between C++ and JS parts of the Synchronization Endpoint is also the boundary between the two code contexts. The Mozilla development team recommends the utilization of DOM events for communication across code contexts. Chrome code subscribes for a specific DOM event and is notified on events raised by code in the context Content. To raise an event the DOM has to be changed, i.e. the actual data to be passed from one context to the other has to be added to the DOM.

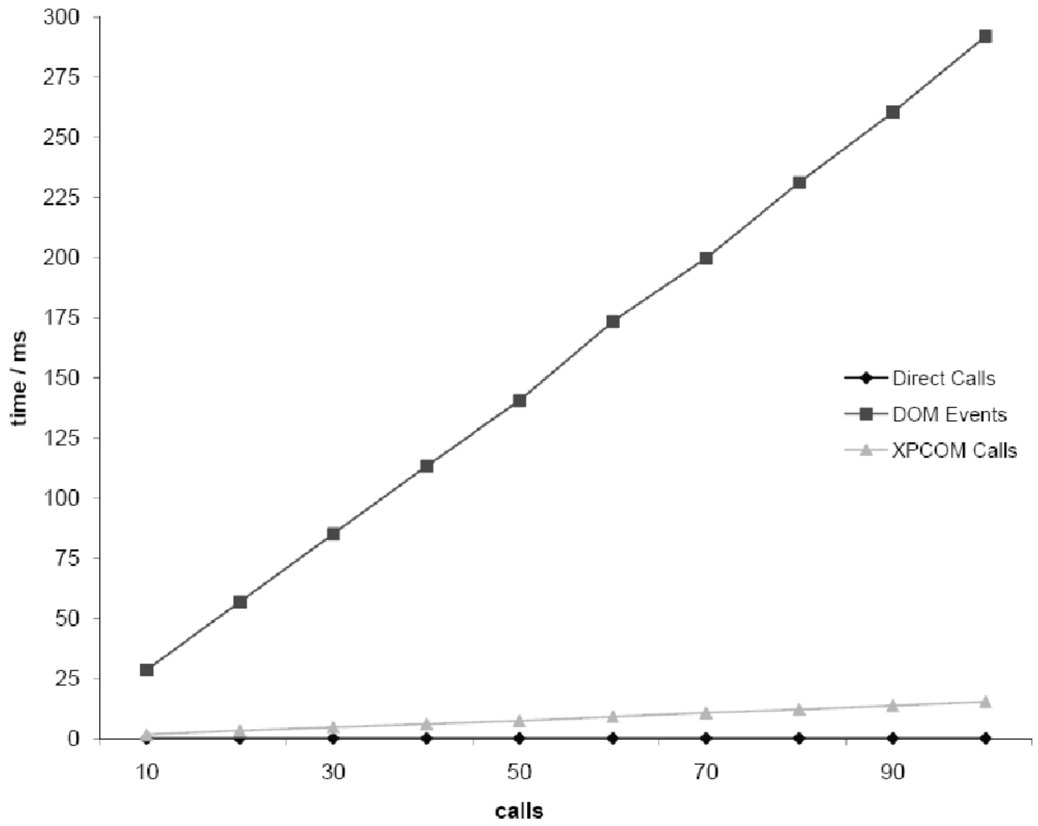


Figure 8-23: Communication across code contexts, DOM vs. XPCOM

In an experimental evaluation, the DOM based communication across code contexts was found to be significantly slower than direct XPCOM calls, as shown in Figure 8-23. The graph visualizes the delays for 10 to 100 subsequent function calls. While the delay for plain JS function calls (line with diamonds) always remains below 1ms, the two cross context calling techniques need substantially more time increasing linearly with the number of calls. However, the XPCOM approach (line with triangles) appears to be much faster than the DOM based approach (line with squares), which needs roughly 2.5ms for a single function call. For this reason the XPCOM approach is favored over the DOM approach and utilized for implementation.

8.4.3 Application Programmer Interface

At the client, application engineers access Pulsar through the Pulsar Application Programmer Interface (API). The API provides access to all Pulsar functions, but hides the complexity of the implementation from the engineer. The starting point of the API is a global object called *P4*. This object is made available by JS libraries which have to be included by the parent HTML document. The interface outline is described with an ECMAScript Binding for the Interface Definition Language in Interface 8-6.

```
[NoInterfaceObject]
interface ServerAdapter {
}

[ImplicitThis]
interface P4 {
    Controller                controller (DOMString endpointId);
    void        attachAdapter    (DOMString adapterName, ServerAdapter adapter);
}
```

Interface 8-6: API – global access object

The object supports the following functions:

controller	A factory function that returns an instance of a system controller from the memory or creates a new one if no controller with the passed endpoint identifier has been requested before.
attachAdapter	Registers a server adapter to Pulsar. Adapters are registered to a unique name. The adapter name is later required to assigned session to a server adapter.

The controller enables application engineers to control the connection of an endpoint to sessions. An outline of the controller interface is provided in Interface 8-7.

The functions are described in the following:

join	Tells an endpoint to join another session. In addition to the session identifier, the server adapter to use and the scopes to subscribe to have to be passed. The function is asynchronous. Accordingly, callback functions have to be provided for both, success of joining the session or failure.
leave	Tells an endpoint to leave a session. Also this function is asynchronous. Callback functions have to be provided to obtain the results of the procedure.
session	A delegate function to access a session once it is available, i.e. joining the session was successful.

```
[Callback]
interface SessionCallback {
    void done(Session session);
};

[Callback]
interface ErrorCallback {
    void notify([AllowAny] DOMString message);
};

interface Controller {
    void join    (DOMString sessionId, DOMString adapterName, DOMString [] scopes,
        SessionCallback success, ErrorCallback error)
    void leave          (DOMString sessionId,
        SessionCallback success, ErrorCallback error)

    Session session (sessionId);
}
```

Interface 8-7: API - session controller

Finally, the session interface and the interfaces of all required callback functions are outlined in Interface 8-8.

```
[Callback]
interface UpdateCallback {
    void update(Operation[] scopes);
};

[Callback]
interface SubscriptionCallback {
    void done(DOMString[] scopes);
};

interface Session {
    boolean  apply (DOMString type, DOMString[] params);
    object     read (DOMString name, any... option);

    attribute UpdateCallback    onupdate;
    attribute UpdateCallback    onbeforeupdate;
    attribute ErrorCallback  onerror;

    void subscribe    (DOMString[] scopes,
        SubscriptionCallback success, ErrorCallback error);
    void unsubscribe           (DOMString[] scopes,
        SubscriptionCallback success, ErrorCallback error);

    DOMString getId();
}
```

Interface 8-8: API - session interface

The functions of the session interface are summarized below:

apply	Is used to apply an operation to the shared state. The function takes the operation type and the parameters of the operation.
read	Returns a subset of the shared state, depending on the implementation provided in the configuration of the Pulsar session. Options are also passed on to the JS implementation of State.
subscribe	Is used to subscribe one or more new scopes. The subscription is performed asynchronously to obtain the subscription result callback functions which have to be provided. If a scope was already subscribed or could not be subscribed for some reason (except an error), this scope is not included in the list passed to the callback.
unsubscribe	Removes a subscription for one or more scopes. Again the list of scopes passed to the callback reflects only the successful unsubscriptions.
getId	Returns the identifier of the current session.
onupdate	Is a session-wide callback handler for state updates. The function can be provided by application engineers to process notifications when and which updates have been performed.
onbeforeupdate	Is similar to onupdate, but called before the updates take effect in the state.
onerror	Is a session-wide callback handler for errors. The function is called when an error occurs that does not result from calling one of the above functions, for example a connection loss.

9 Evaluation and Benchmark

IS3 combines known concepts for the realization of interactive real-time applications. While the related work analysis revealed the novelty of the combination, the constituting algorithms base on previous findings. Thus, a quantitative comparison of the solution as a whole to related solutions is not feasible. A quantitative comparison on the algorithmic level would not result in any new findings. This chapter instead documents the results of evaluation and benchmark tests with the Pulsar implementation. The tests demonstrate the appropriateness of the algorithms selected for IS3, confirm complexity expectations and provide basic performance figures for future comparisons.

9.1 System Evaluation

The aim of system evaluation test is to confirm the correct implementation of the pipelining algorithm introduced in section 5.3.1, and to demonstrate the appropriateness of the pipelining algorithm to achieve fairness in face of delay prone networks.

9.1.1 Setup of Test Environment

The evaluation tests are conducted for a small distributed testbed. The setup, shown in Figure 9-1, comprises five computers connected through an exclusive, local area network based on Gigabit Ethernet (the figure shows the logical connections).

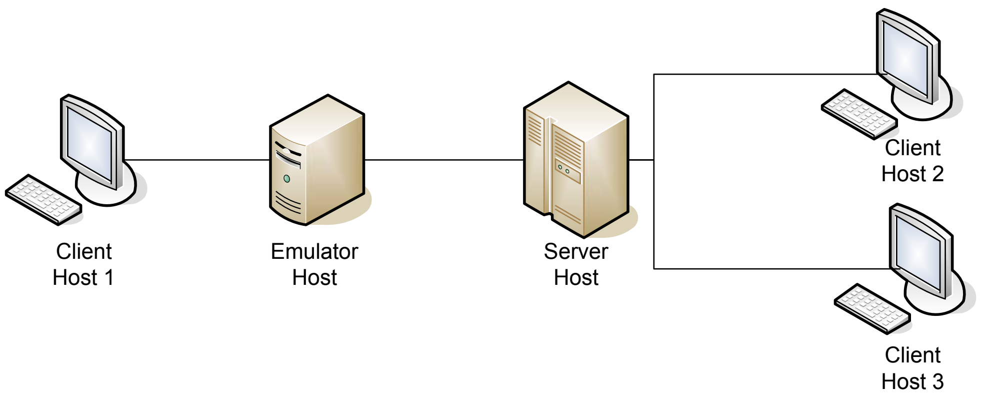


Figure 9-1: Setup of test environment

The application run by the Emulator Host works basically like a repeater on the link between *Client Host 1* and *Server Host*. In addition to forwarding packets, this application can induce network characteristics common for wide-area network connections, such as packet delay, packet delay jitter, and packet loss. The characteristics are configured through a SOAP interface per flow and enforced by *netem* [94], an established network emulation software for Linux systems. For example the network characteristics emulator allows configuring the delay of each packet of a specific transport connection from an application on *Client Host 1* to an application on the *Server Host*. For the communicating applications, the network emulator is completely transparent. They only experience the induced packet characteristics. Table 9-1 summarizes the test bed components and their characteristics. The distributed setup is chosen to bypass scheduling mechanisms of operation systems and virtual machines, as well as to obtain results which present the performance of the logical components in isolation. The user agent is Mozilla Firefox 3.0.1.

Machine	*Configuration*
Server Host	Hosting the Pulsar server and a Web server; Running Windows Server 2008 64bit; 8GB of RAM, Intel Xeon 2GHz quad core CPU, and two network adapters.
Client Host 1	Hosting WHA Clients; Running Windows XP 32bit with 2GB of RAM, and AMD Athlon 64 2.2GHz CPU.
Client Host 2	Hosting WHA Clients; Running Windows Vista 32bit; 2GB of RAM, and Intel Core 2 Duo 2.4GHz dual core CPU.
Client Host 3	Hosting WHA Clients; Running Windows XP 32bit with 2GB of RAM, and Intel Pentium 4 2.8GHz CPU.
Emulator Host	Shaping network traffic. Running Ubuntu Linux 9.04 32bit; 1GB of RAM, Pentium 4 3,2GHz CPU, and two network adapters.

Table 9-1: Configuration of testbed machines.

9.1.2 Test Procedure

For the system evaluation, a simple web application based on Pulsar is realized. To investigate the impact of network characteristics as directly as possible, the binary network interface of Pulsar is chosen for the tests. The qualitative difference to comet techniques is shown in [95]. The test application shares one object between all clients. Object modifications are exchanged in operation mode (1) - continuous convergence and effect simultaneity. WOS is used as state model. For the WOS inclusion function, the pseudo-random tie-breaking method described in section 7.4 is activated.

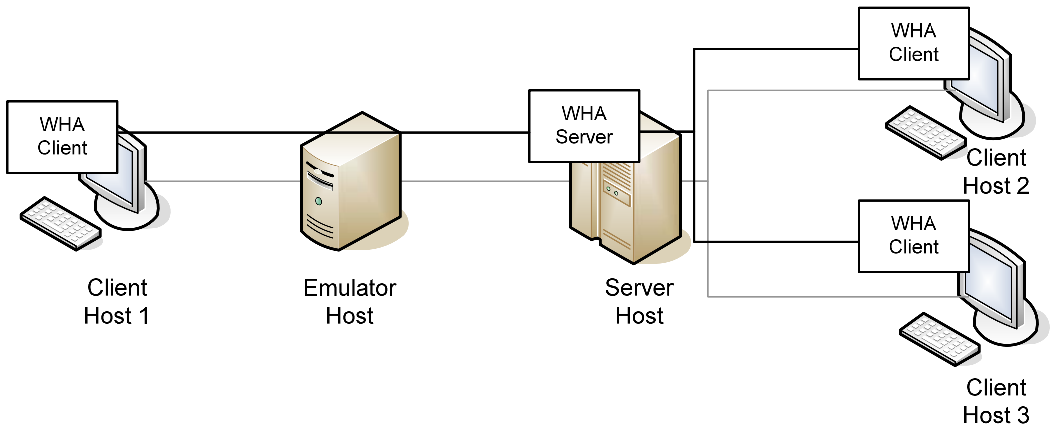


Figure 9-2: Evaluation web application deployed to the testbed (logical setup)

The test application implements two client types, an active client and a reactive client. An active client (also called *master)* issues operations. A reactive client (also called *slave)* only responds to operations by creating its own operation for each one received. Figure 9-2 shows the web application deployed to the testbed. Each machine runs exactly one client in this setup. The actual test procedure is summarized in the following:

1. The master starts the test procedure by issuing one operation on the shared object. The time T_{issue} is recorded.
2. All clients receive an operation from the server reflecting the operation of the master. The master itself notes the time T_{effect} and calculates the interval $t_{action} := T_{effect} - T_{issue}$.
3. As soon as a slave receives the operation, it updates the shared object by writing a unique identifier to the shared object with an operation of type set. Since all slaves receive the operation in the same interval, the resulting set operations are competing. Only one of the slave requests can succeed.
4. The server processes the operations and sends updates to all clients. The master logs which slave request has been successful. Additionally, the master records the time $T_{feedback}$. and calculates the interval $t_{reaction} := T_{feedback} – T_{effect}$

During the tests the time is taken in Pulsar client directly, as well as with the open packet sniffer Wireshark [96]. A sequence diagram visualizing the procedure is shown in Figure 9-3. For simplicity, only the master client and one slave are depicted. In practice two or more slave clients join the same session, as shown in Figure 9-2.

The binary Pulsar network interface supports an automatic adaptation of the bucket interval duration T introduced in section 5.3.1. The minimum interval duration is set to 20ms to avoid overload on the client machines distorting the results. The maximum interval duration is set to 120ms. The pipeline depth is set to a = 1, thus the play-out delay is $\Delta = aT = T.$ The test application is configured to repeat each test 1000 times. The average values of the results should be statistically meaningful.

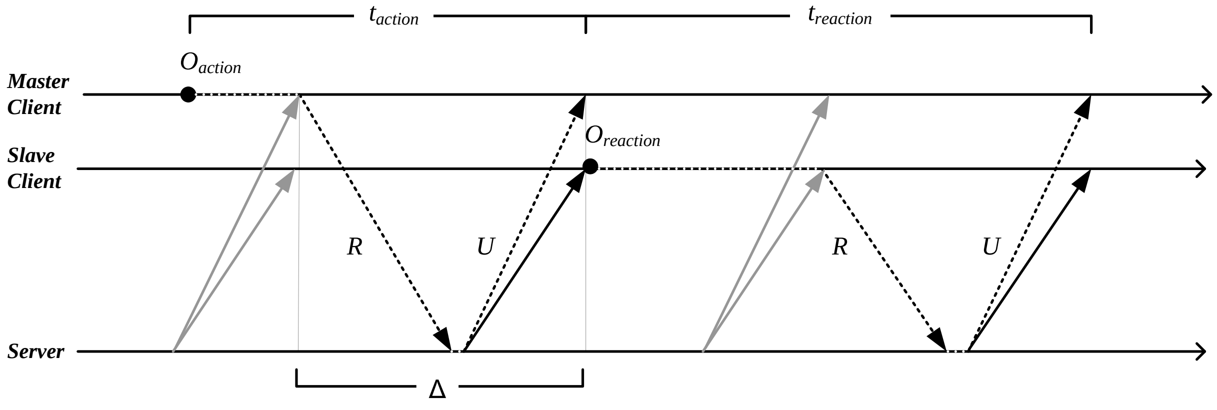


Figure 9-3: Sequence diagram of test procedure

9.1.3 Testing Interaction Delay

Interactions in IS3 base on mutual write operations to objects. The period which elapses between a write operation of one user and the play-out of a write operation reflecting a response from another user is called interaction delay. The interaction delay is defined as $t_{interaction} := t_{action} + t_{reaction}$. As suggested by Figure 9-3 the time span t_{action} is defined by $t_{action} := \Delta +$

t_x. The constant Δ is the play-out delay introduced in section 5.3.1. The variable t_x represents the interval between the moment a user issues an operation O_{action}, above also called T_{issue}, and the moment of transmission with a request message. In the test application operations are automatically issued at random times by the master client. The time span t_{action} is referred to as *synchronization delay* in the following.

Proposition

The results are average values based on a large number of records. It is assumed that t_x approximates to its probabilistically expected value $E(t_x) = \frac{\Delta}{2}$. The average value for synchronization delay is thus $t_{action} = 1.5\ \Delta$.

A user can react to O_{action} as soon as the operation effect is played out. Analogously the slave client responds to a received operation immediately with an operation $O_{reaction}$. The reception of an update message is also the beginning of a new interval at the client. Thus the client has to wait for the end of the interval and the next update message from the server to deliver $O_{reaction}$ with a request message. Since the pipeline depth is $a=1$, the idle time is Δ. The operation is processed by the server and reaches the originator of O_{action} at the beginning of the next interval. This means $t_{reaction} := 2\Delta$ and thus the average value of $t_{interaction}=1.5\ \Delta + 2\ \Delta = 3.5\Delta$.

It is assumed that packet loss, packet delay and packet delay jitter are negligible. All packets and thus messages and operations are on time. The time points are recorded for an interaction between two clients, one slave and one master. The second slave introduced in the description of test procedure is inactive.

9.1.4 Results of Interaction Delay Test

The measurement results for the synchronization delay are depicted in Figure 9-4. During the measurements the play-out delay Δ is increased in steps of 10ms. The values for the synchronization delay are mapped to the corresponding play-out delay. The figure contains four graphs, i) the highest individual value measured throughout the test runs (upper end of error bar), ii) the average value over all test runs, iii) the lowest individual value measured through the test runs (lower end of error bar) and iv) the expected value for the mean over all test runs.

The mean of the recorded values matches the expectations for the synchronization delays. The negative variation of individually measured values can be explained with the randomly assigned value for t_x, which causes t_{action} to vary in the range $2\Delta \geq t_{action} \geq \Delta$. Thus, the lowest individual value that can be found theoretically during the test runs is $t_{action} = \Delta$. The measurements confirm this expectation in practice. The positive variance of individually measured values significantly exceed the expected maximum value $t_{action} = 2\Delta$. The values greater than the expected maximum value occur with a frequency that has a negligible impact on the average value. These outliers can be explained with load peaks at the client machines, e.g. caused by garbage collection mechanisms of user agents.

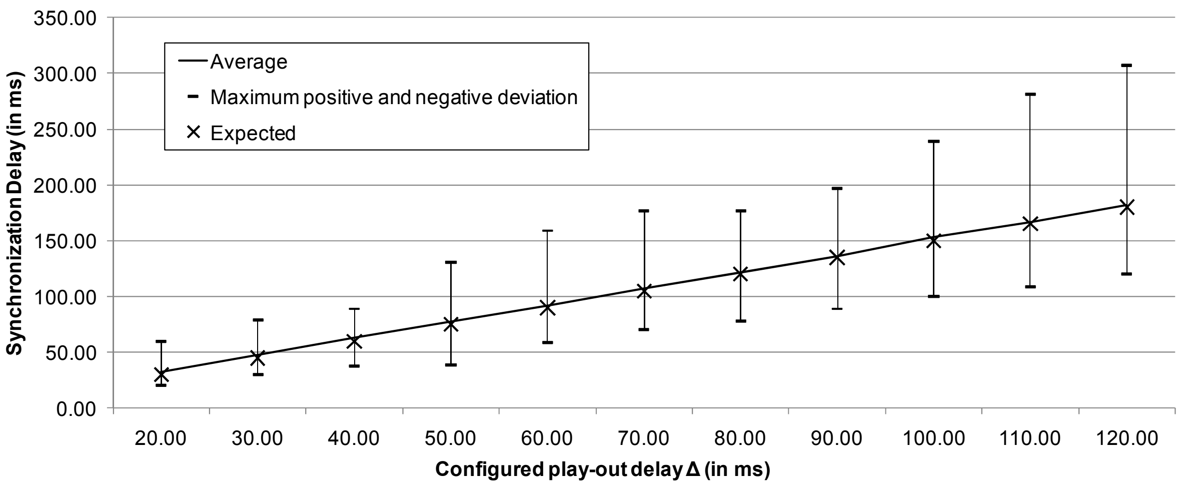


Figure 9-4: Measurement results for the synchronization delay with minimum and maximum deviation

The measurement results for the interaction delay are depicted in Figure 9-5, which again contains four graphs, i) the highest individual value the interaction delay throughout the test runs, ii) the average value over all test runs, iii) the lowest individual value of the test runs, and iv) the expected value of the interaction delay for the average over all test runs.

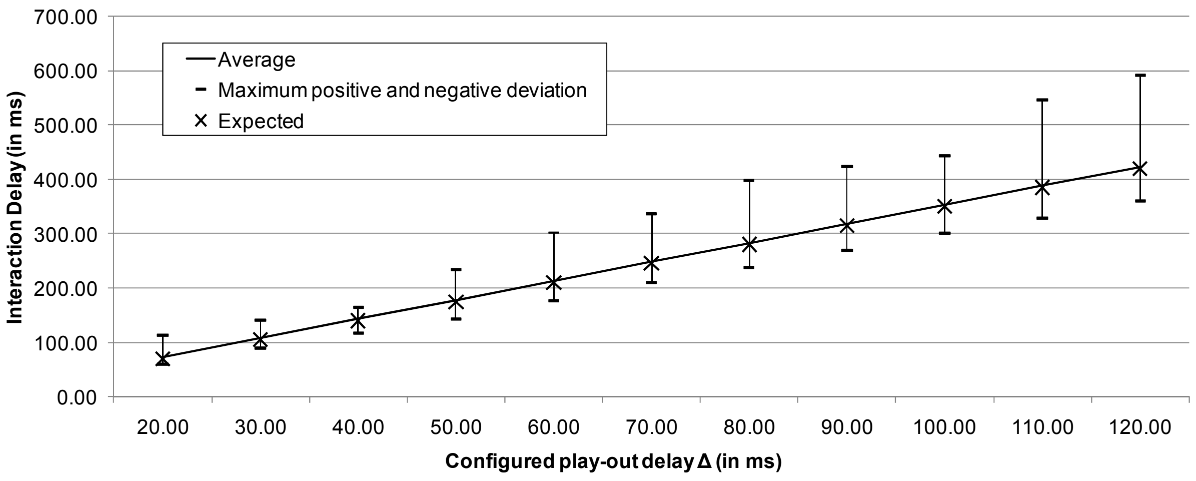


Figure 9-5: Measurement results for the interaction delay with minimum and maximum deviation

The average values match the expected values for the interaction delay. The negative variance of the measured values is again explainable with t_x.Individual values of the interaction delay can be found in the range $4\Delta \geq t_{interaction} \geq 3\Delta$. The positive variance again reflects a number of outliers that do not appear frequently enough to have an impact on the average values.

The measurements of synchronization and interaction delay indirectly confirm the reaction delay $t_{reaction}$ of IS3 with $t_{reaction} = 2\Delta$. As long as the play-out delay Δ is less than 150ms a reasonably natural presentation of human interactions is given.

9.1.5 Testing Fairness of Conflict Resolution

An important criterion for the design of IS3 is to support competing interactions, i.e. guarantee fairness. Fairness not only means to replay operation effects simultaneously, but also to process competing operations independently from network characteristics. As introduced in section 9.1.2, the test procedure is explicitly designed to provoke competing operations of slave clients, i.e. concurrent write operations of different values to the same object. A modified WOS inclusion function ensures a random distribution of success at the server as long as the concurrent operations are received on time. Success means the object finally has the value intended by a client operation. With two slave clients, a resulting success rate of 50% is expected per slave. A deviation of less than ±2.5% is assumed to be tolerable. The mean of the individual success rate *r* of each client (i.e. write success across all iterations) thus has to be within the range 52.5% ≥ r ≥ 47.5% to be considered fair.

Proposition

Packet delay t_{delay} does not affect fairness as long as the overall time for request message transport does not exceed $\frac{\Delta}{2}$. Since the network interface used for the tests sends each message with a single datagram, the time for message transport is mainly comprised of t_{delay} and a negligible message processing time. If a request message is on time, i.e. $t_{delay} < \frac{\Delta}{2}$, the enclosed operations are on time. Competing operations received on time are treated equally in conflict resolution. Consequently, an even distribution of operation success among the slave clients is expected. Also packet delay jitter does not affect fairness as long as $t_{delay} + t_{jitter} < \frac{\Delta}{2}$ for any variation t_{jitter} from the average delay. In the test procedure, late request messages and their operations are ignored by the server. In this case, the originating client is in an inferior position.

A different behavior is observable when packets are lost. Although the network interface used in the tests retransmits a lost message and thus the lost datagram, the retransmitted message is inevitably late. Consequently, the success rate of a client should decrease when message loss increases.

The binary network interface used for the tests supports an automatic adaptation of the bucket interval length *T* per session. With every late message, the play-out delay Δ is increased by 20ms and *T* is increased by $\frac{\Delta}{a}$. Since the pipeline depth for the tests is a=1, *T* is increased in steps of 20ms up to a predefined maximum interval length of 100ms. The initial interval length is 20ms.

9.1.6 Results of Fairness Test

For the measurements in the fairness tests, packet delay, packet delay jitter and packet loss were artificially generated by the network characteristics emulator. The datagrams and thus the messages sent by the slave client on Client Host 1 are exposed to the generated network characteristics.

Figure 9-6 depicts the success rate of the client with delay prone connection to the server. The test is repeated with different packet delays. For each tested packet delay, the success rate of the client is within the expected range. The measurements do not indicate any correlation of packet delay and success rate. The automatic adaptation of the interval duration *T* ensures that after at most three late messages, all artificially delayed messages are on time. Operations of these three late messages are rejected. An effect on fairness is not observable since the test was conducted with 1000 iterations on the same session.

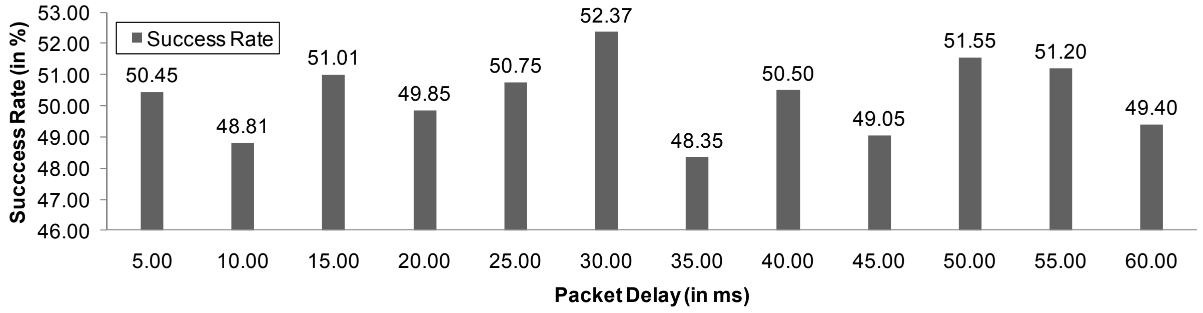


Figure 9-6: Success rate of a client with delay prone connection

Figure 9-7 depicts the success rate of a client with packet delay jitter prone connection to the server. For the measurements, a constant delay of 50ms was chosen for each message, while the delay jitter varied randomly between 0ms and 40ms. The measurement results show that packet delay jitter has not significantly impacted on the success rate in the scenario. The automatic adaptation of the interval duration *T* ensures a save interval duration is reached at the latest after four late messages. In contrast to the above discussed measurements for packet delay, the adaptation is not terminated after a number of heading test runs. Since messages are exposed to varying delays, late messages are not necessarily consecutive.

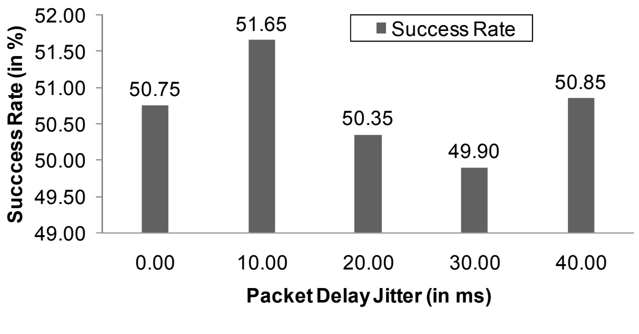


Figure 9-7: Success rate of a client with jitter prone connection

As depicted in Figure 9-8, packet loss clearly decreases the success rate of the affected client. The success rate is measured as the probability of losing individual packets increases. For example, with a loss probability of 50% in roughly half of the 1000 test runs operations of the affected client are late. The success rate for about 500 timely operations remains about 50%. The total success rate thus decreases to 25%. The client is obviously in an inferior position. The measured values beyond 30% loss rate appear to be slightly irregular. For those values, the actual number of test runs is partially less than planned. Due to the large number of lost messages the client implementation occasionally skipped test runs.

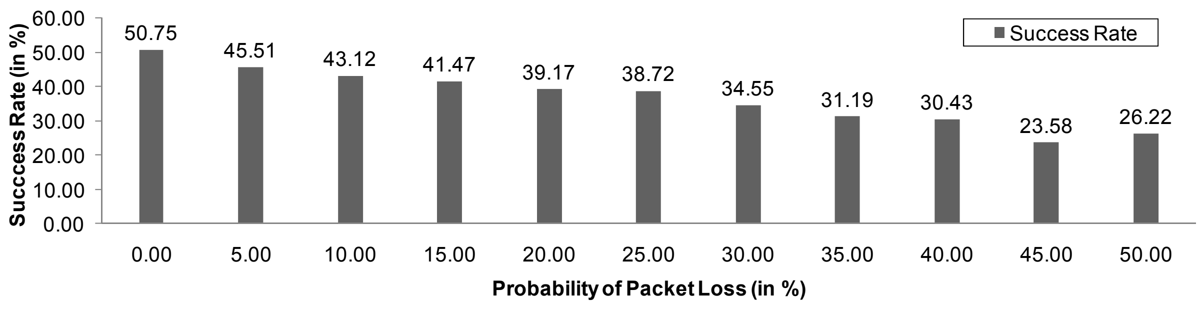


Figure 9-8: Success rate of a client with loss prone connection

Although the measurements are conducted with the binary implementation of the Pulsar network interface and thus a datagram based connection, the measurement results are also of significance for the text based network interface. Although the impact of packet delay and packet delay jitter is not as direct in TCP based connections (e.g. HTTP, WebSockets) as in UDP based connections, both phenomena have a comparable effect on the transmitted data stream and will influence the transport time of messages. The same is true for packet loss. A detectable side effect of TCP retransmission is a lag in the data flow. A countermeasure, for example an increase in the TCP input buffer size, has a negative impact on the timeliness of transported messages. More results are published in [97].

9.2 System Benchmark

The aim of the system benchmark tests is to confirm complexity estimations of the algorithms and to determine the performance of the Pulsar implementation for IS3. The following sections describe test environment, setup, procedure and results.

9.2.1 Setup of Test Environment

The test environment for the benchmark tests comprises two machines, a Server Host and a Client Host. Both are connected to an isolated Gigabit Ethernet network without other participants. The configuration of both machines is summarized in Table 9-2. The distributed setup is chosen to minimize side effects caused by sharing resources between multiple applications on one machine.

Machine	*Configuration*
Server Host	Hosting the Pulsar server; Running Ubuntu 10.10 32 Bit Desktop Version; 1GB of RAM, and Genuine Intel CPU T2400 Core Duo 1.83GHz.
Client Host	Hosting test client with Pulsar client stack; Running Windows 7 32bit; 2GB of RAM, and Genuine Intel CPU T2500 Core Duo 2.0GHz.

Table 9-2: Configuration of machines in benchmark testbed

9.2.2 Test Procedure

The architecture of IS3 is centralized. The performance of an IS3 implementation like Pulsar thus mainly depends on the server. To investigate the server behavior in face of scaling load, i.e. number of clients and concurrent operations, a test setup with real web user agents is inappropriate. 100 and more browser instances on a single client host would dramatically slow the host machine down. The actual server performance figures would be masked in the test results. In contrast to delay and fairness tests, the Pulsar system is therefore tested in isolation. A simple *dummy client* is used in place of the regular WHA client. The dummy client is a lightweight console application only utilizing the Pulsar client stack. For the benchmark tests, the text-based client stack implementation is used in combination with an implementation of the WebSocket protocol draft 76 [98]. For the server, the client implementation looks like a proper one integrated with the user agent. A single instance of the dummy client can open and maintain concurrently several hundred synchronization connections to the server, i.e. mime endpoints. The Client Host runs the dummy client, while the Server Host runs the Pulsar server. Object modifications are exchanged in operation mode (1). The operation mode, however, does not affect the performance. The computation load at the server is the same for both operation modes. WOS is used as state model.

Once started, the dummy client initiates a predefined number of endpoints and joins the same session with all endpoints. Depending on the test configuration, the number of operations per client is varied. Operations of different clients are defined to conflict. The time between the reception of the first request message and the sending of the last update message per interval is measured for each configuration. All tests are repeated 100 times for each configuration. The measurement procedure comprises the following four steps:

1. The server receives and parses messages of all clients. Since messages can arrive at any time during the first half of the play-out delay, measuring the interval between receiving the first message and receiving the last message is inappropriate. Instead, the times required to parse the single request messages on reception are recorded and finally summed up to $t_{receive}$. Since the reception of messages from different clients occurs in parallel at the server, $t_{receive}$ is only an indicator for the overall time complexity of message parsing. Play-out delay Δ is set to 2s, which is far beyond a reasonable value for an application. This setting, however, ensures the server will wait long enough to receive the message of all clients issued for an interval on time. The pipeline depth is set to $a=1$ for the tests.

2. The server processes the operations contained in the request messages. All operations are intentionally conflicting and thus subject to the transformation procedure. At the beginning and at the end of the transformation the time is recorded. The interval between the two timestamps is referred to as $t_{transform}$.

3. The resulting client operation sequences are serialized as update messages and sent to the server-site WebSocket implementation. Again the time is recorded at the beginning and the end of the process. The interval is referred to as t_{send}.

9.2.3 Testing Performance

The two variables which mainly affect the performance of the IS3 server and its implementation in Pulsar are the number of messages per interval, i.e. the number of clients on the same session, and the number of operations per message. Increasing numbers of messages and operations result in an increasing demand on processor time and memory of the machine hosting the Pulsar server. This increasing demand reflects in the overall time taken for processing per interval.

Section 5.6 closes with a discussion of the time complexity of the transformation algorithm. The derived estimates predict an exponential time complexity for the control algorithm with respect to the number of requests per interval. Also they predict a quadratic time complexity of the sequence inclusion algorithm with respect to the number of operations per client and interval. The time complexity for parsing messages is linear with respect to the number of clients and operations per client and interval. If x clients send n operations, x request messages and xn operations have to be parsed. The time complexity for serializing and sending messages is quadratic with respect to the number of operations per client and interval. Each received operation is forwarded to every client in the test setup. With n operations per client and x clients xn^2 operations have to be serialized and send.

The intention of the performance tests is to validate the time complexity estimates and to provide benchmarks for different test configurations. The benchmarks point out limits of Pulsar when applied in practice.

9.2.4 Results of Performance Tests

The first part of the benchmark test addresses the interval transformation algorithm documented in Algorithm 5-2. 10 to 100 endpoints are connected to the server. For each test run, the rate number of endpoints is increased by 10. Each endpoint requests exactly one operation per interval, all operations compete. Thus, the time complexity for all list transformations performed to calculate the transformation matrix explained in section 5.6 is constant during this test. The measured times directly reflect the time complexity of the interval transformation algorithm estimated in the same section.

Figure 9-9 shows the overall processing time $t_{processing} = t_{receive} + t_{transform} + t_{send}$ in milliseconds. The shaded areas in the columns represent the components of $t_{processing}$. The measurements were repeated 100 times and the mean taken for each configuration. The highest measured individual variation from the mean was 247.93%. However, the means best show the trend of the performance and confirm the estimated exponential time complexity.

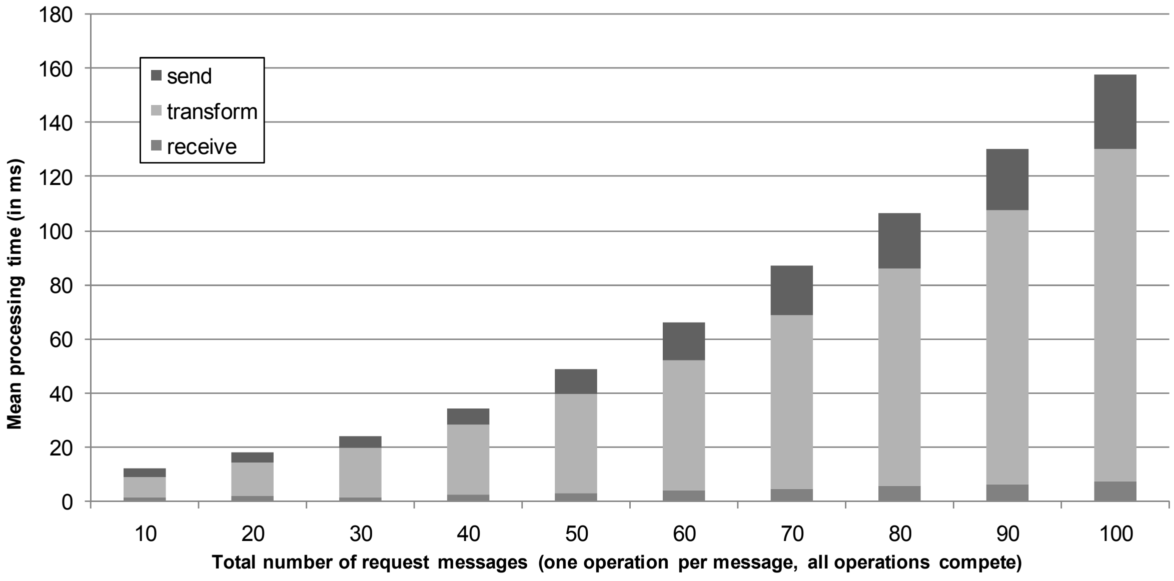


Figure 9-9: Mean processing time for increasing numbers of clients and a constant number of operations per client.

A configuration with 100 clients and one operation per request message causes a processing time above 150ms. This time has to be found between the reception of the last request message and the end of $\frac{\Delta}{2}$ in order not to extend the play-out delay unintentionally. The bucket interval duration T has to be chosen for an application. An alternative strategy is to implement the control algorithm to immediately fit a list of operations in the transformation matrix on reception. The effect is the same as for Algorithm 5-2, but the bulk processing of operations is potentially better balanced over the first half of the play-out delay.

Figure 9-10 again depicts $t_{receive}$ and t_{send}. As expected the overall time required for message parsing grows linearly, while the time for message serialization and sending grows quadratically. Although both curves are not shaped perfectly, they allow derivation of the above mentioned trends. Deviations of at most 4ms are probably caused by peculiarities in the implementations of included third party libraries (e.g. buffer flushing behavior), which come into effect with specific test configurations.

The second part of the benchmark test addresses the list inclusion described in Algorithm 5-1. For this part of the test, ten request messages arrive at the server per interval. All messages contain the same number of operations. Each operation competes with all operations from other origins. The number of operations per message is increased from 1 up to 15 throughout the test iterations. Each test is repeated 100 times.

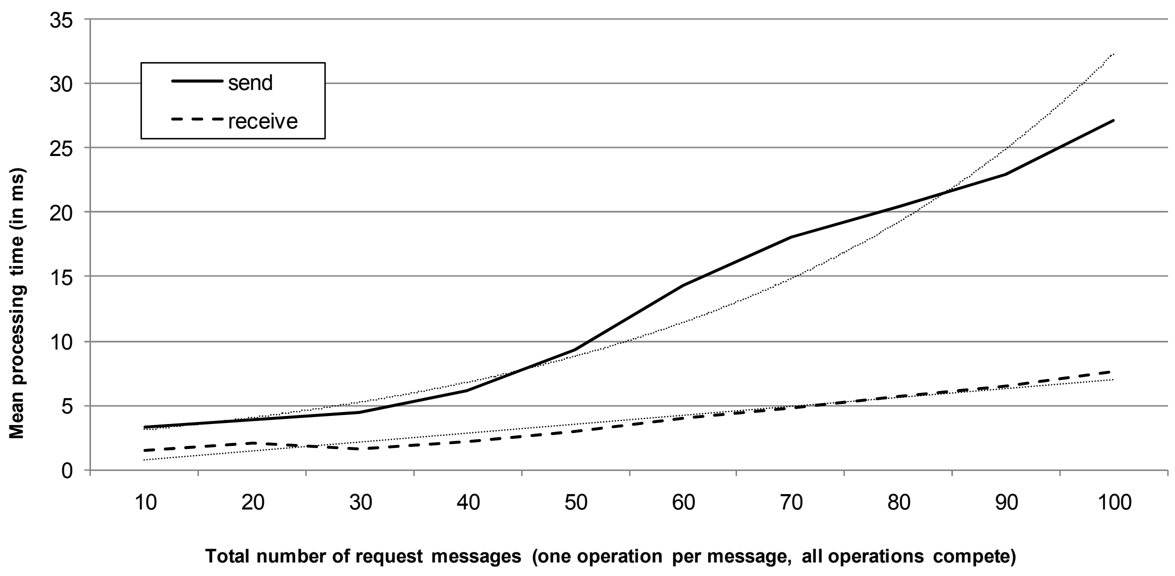


Figure 9-10: Mean receive and send time for increasing numbers of clients and a constant number of operations per client.

Figure 9-11 depicts the results for the components of $t_{processing}$ as columns. Again the mean of the measurement results is taken. The highest deviation of a maximum value from the corresponding mean throughout all iterations is 23.01ms. The absolute deviation of maximum values from mean develops linearly. As expected, $t_{transform}$ grows quadratically with respect to the number of operations per request message.

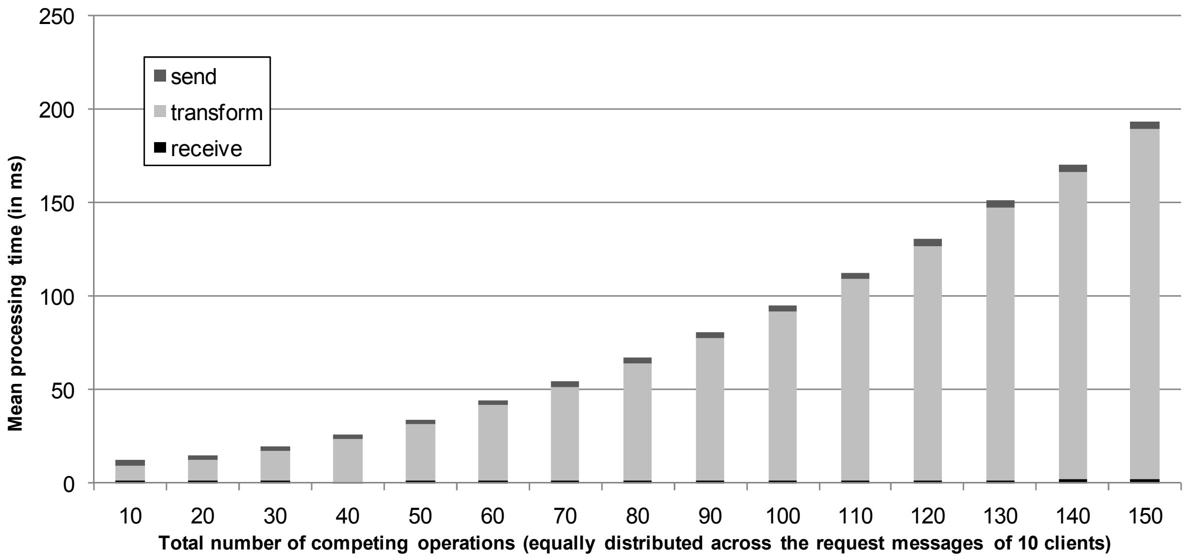


Figure 9-11: Mean processing time for increasing numbers of operations per client and constant a number of clients.

Ten competing clients, each originating 15 operations per bucket interval, cause an overall processing time of almost 200ms. Unless the bucket interval duration for an application is chosen greater than 1000ms, 15 operations per interval and client is a rather unrealistic assumption. However, one reason for the poor performance of the list inclusion algorithm here is probably the runtime interpretation of a JS inclusion function. Using the JS implementation of the WOS inclusion function is the default option for a session configuration which was also chosen for the benchmark test. The Pulsar implementation also supports inclusion functions implemented in Java. An implementation of the WOS inclusion function in Java can be expected to have had a positive impact on the performance of interval transformation algorithm too.

10 Case Study

This chapter documents the review of the IS3 concept and its reference implementation Pulsar with respect to the objectives of the thesis. Two software engineering processes are researched in a case study. The documentation starts with a summary of prospects to be verified and concludes with a summary and analysis of the findings.

10.1 Propositions and Methodology

The objective of this work is to provide a software framework that supports the engineering of real-time interactive applications. Software engineers are intended to experience the software framework as a tool and a ready-to-use building block that simplifies their task and saves them time for other aspects of the application to be produced. The benchmark and evaluation tests documented in chapter 9 showed the sufficient operation of Pulsar and thus the IS3 concepts, but they cannot confirm whether the objectives with regard to the engineering process have been achieved or not. For this purpose a Case Study is conducted which shall support the following propositions:

1) Pulsar can be used to realize applications that support real-time collaborative as well as competitive interactions.
2) Utilizing Pulsar to realize real-time interactions takes less time than then implementing a propriety solution.

A suitable case to investigate the impact of Pulsar in software engineering is an exemplary software engineering project. To cover usage spectrum of Pulsar, two cases are researched. The first case addresses collaborative interaction, the second case competitive interaction. The first case is the realization of a collaborative editor for rich text documents. The second case is the realization of an online adaption for the game Bomberman. Although the cases address different applications, the processes should result in similar findings with regard to the propositions.

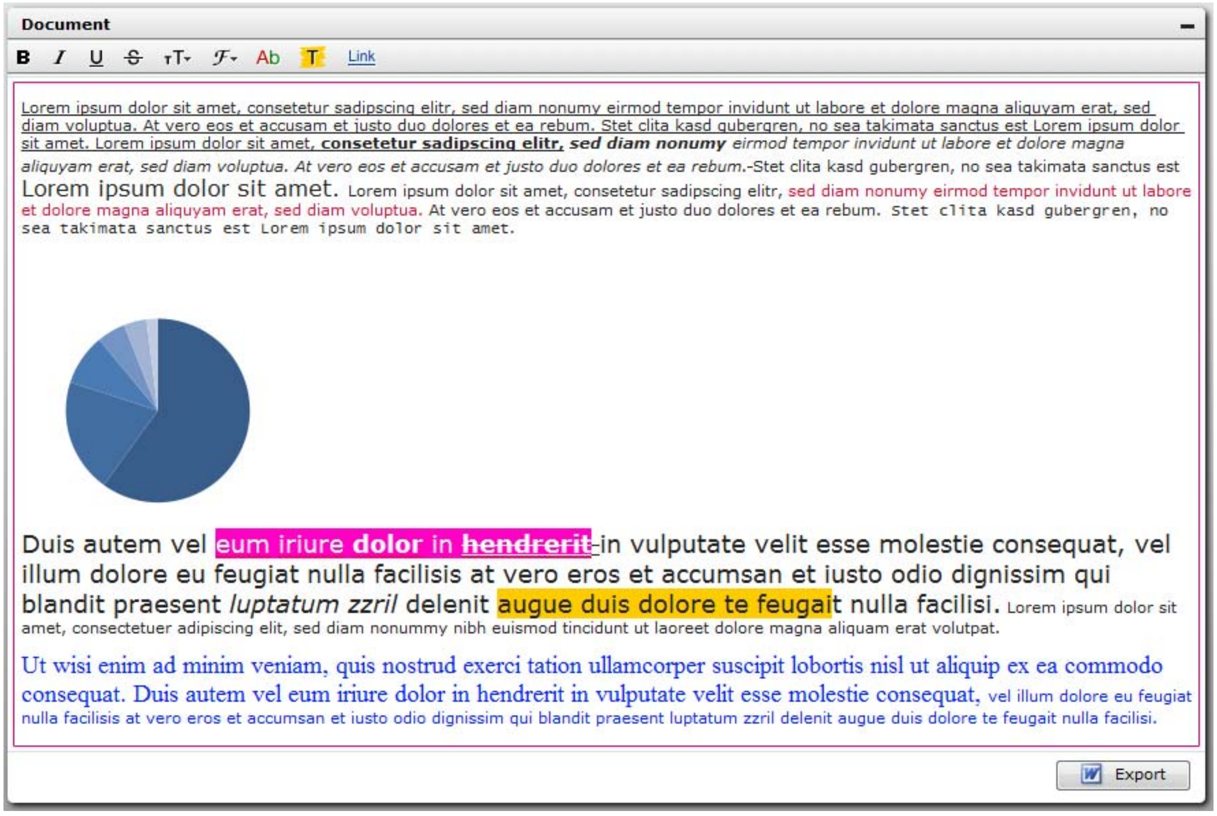
Proposition 1) is supposed to be verified when the teams produce applications that fulfill the application requirements summarized at the beginning of sections 10.2 and 10.3 respectively. As evidence for the correctness of proposition 2) the following aspects are investigated:

a) The learning effort, i.e. the time a team needs to learn and utilize Pulsar for the envisioned application.
b) The scale of program code (lines of code) needed to adapt Pulsar and its interfaces to the application context.
c) The familiarity of team members with the algorithms and concepts implemented in Pulsar before entering the case study.
d) The total time needed for the realization of applications.

For the collection of evidence both software engineering processes are observed over time, for Bomberman on a weekly basis, for the text editor on a daily basis. Additionally the members of the engineering teams are interviewed about their skills before and after the projects, as well as their experience with Pulsar. Insights into the program code produced by both teams help to understand the problems the team members are facing when integrating Pulsar with their applications. Thus, i) direct observations, ii) interviews, iii) code reviews, and iv) requests for technical support are the data sources forming the base of triangulation in this Case Study.

10.2 Case: Collaborative Text Editor

Application: The Collaborative Text Editor enables people to write and style text collaboratively on the web. The editor is identical to text editors realized as single-user desktop applications. A marker, the caret, shows where text is modified if the user starts writing or deleting, while the caret position can be changed by arrow keys and mouse actions.



Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, **consetetur sadipscing elitr,** ***sed diam nonumy*** *eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum.*-Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure **dolor** in ~~**hendrerit**~~ in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent *luptatum zzril* delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Figure 10-1: Screenshot of the application resulting from the case "collaborative text editor"

Text is styled by selecting a section and choosing the preferred style option for this section from a menu. The text editor in this case supports only simple style options such as bold, italic, change font-size, change font-style and change colors. Additionally, hyperlinks and images can be inserted into the text, and text from the clipboard can be pasted in. Figure 10-1 depicts a screenshot of the final editor. The difference of the collaborative text editor to a single-user editor is the immediate reflection of user actions in all open copies of the text. Typed characters appear and styles take effect at a copy without affecting the locally editing user. Instead of manually saving changes, the edited text is saved automatically every 30 seconds from the master state of the resource – user can only export the current state as a portable document if needed.

The implementation of the text editor here is a subproject of larger software project on the realization of an integrative frontend for various communication channels, such as phone, SMS, forums and collaboratively editable documents. The project structure nonetheless ensures an isolated realization of the text editor.

Team: The collaborative text editor is realized by a team of two software engineers, a computer science student and a computer science graduate. Before the latter two started with the text editor they realized other components in the project in which the editor is embedded. Accordingly, they are familiar with the modern application model of the web and the programming languages required for the creation and configuration of a Pulsar-enabled application, i.e. JS and Java, and they have worked with Pulsar and the WOS model before. They used it for the realization of smart message inboxes and contact lists with rich presence features and did not need to change the server configuration. Initial interviews with the team revealed that both engineers had not implemented a rich text editor with HTML and JS before. They have not realized a collaborative text editor before or studied relevant solutions. Nonetheless, they know the capabilities of OT and DS from version management systems, office solutions and online services like Etherpad [99].

Engineering Process: The engineering activities for the collaborative text editor began with a 2 hour face-to-face meeting with the manager of the parent project. The application and the planned features were first presented to the team, then discussed and finally collected in a feature backlog. 36 person days are granted to the team for the features in the backlog. The list reflects the capabilities of the editor application introduced above. The original specifications came from the customer of the parent project. Starting with the kickoff, the team members were sporadically observed throughout each working day. Additionally, their progress and problems they faced were recorded during daily SCRUM meetings. The resulting insights in the engineering process are documented in the following.

The team used a WHA and Pulsar server provided for the parent project. The Pulsar server and the WHA server were hosted on the same machine, so the team could use the plain server interface to connect both servers. Consequently, they saw no need to implement a robot in order to realize the automatic saving of documents and loading of documents. Since the postgraduate and the student were already familiar with client and server interface of Pulsar, they began the engineering with researching best practices for implementing a rich text editor with HTML and JS. As a first step they realized a very simple text editor without styling options, copy and paste or other convenience features. Code reviews show that the text input based on an editable HTML5 layout element. The text was mapped to an array of the WOS model, while each character represented a separate array field. Pulsar was used in its default configuration. Whenever a key press event occurred on the editable layout element, the event type was analyzed and an appropriate operation to the character array in Pulsar derived. For that purpose the caret position relative to the beginning of the text had to be obtained from the range API [100] of the DOM. The key-press event of the "A" key was for example mapped to an *insert* operation; a backspace or remove key was mapped to a *truncate* operation. Operations received from the server, i.e. insertions to or deletions from the array, were applied to the HTML layout element with the help of the HTML5 editing API [101].

This first iteration of realizing a collaborative editor took only four person days, after which the team was able to fully utilize Pulsar in their application context. But this first iteration also revealed challenges the team had to face when starting to support rich text. The editing API of browsers is not fully implemented in all user agents which the parent project addresses. For example, the insertion of a new paragraph does not work properly in Mozilla Firefox version 4. The API for accessing text selection and text ranges is implemented slightly differently in all user agents. However, the consistency of the states presented to users depends on the reliable detection of changes to the text and the reliable application of the effect to all currently viewed instances of the text. Solving this challenge cost the team more than 90% of the engineering time while less than 10 % was spent on configuring and integrating Pulsar as the observations and interviews document in triangulation.

As a first step towards a rich text editor, the team extended the character array representing the text in Pulsar to a kind of token array. For that purpose they adapted the idea of nesting markup with parentheses from XHTML. The example below shows formatted text and the corresponding array in Pulsar as JSON [39] representation.

Text: hello **wo*r*ld**

Array:
```
["h", "e", "l", "l", "o", " ",
{start: "bold"}, "w", "o", {start:"i"}, "r", {end:"i"}, "l", "d", {end:"bold"}]
```

To assign a formatting to a text section a *start* and an *end* object have to be added as tokens to the array. Object attributes define the token type. The tokens represent a direct mapping of the markup elements in the editable HTML container element for the editor (e.g. *<i>* and *</i>* for the above example). Adding the tokens for one style (e.g. italic) requires the application of at least two Pulsar operations to the text. Thus, the user action of applying a style to the text cannot be mapped to one atomic operation in Pulsar with this approach. As a consequence, style-related operations cannot be fully treated by the inclusion mechanism of Pulsar, which results in suboptimal state text representation when utilizing WOS. This problem is addressed in a second iteration of the editor that has been developed after the end of the case study. The findings are summarized below. Besides this drawback, the implementation of the WOS model delivered with Pulsar proves to be useful for the realization of rich text editors. Team observations and code reviews, the sources of the above data, make clear that the adaption of Pulsar in the context of the editor requires no implementation of a middle layer or complex data mappings. Thus, the team had to spend no extra days on making Pulsar useful in the application context.

In the first iteration of the collaborative editor, the reliable detection of the actual changes a user agent renders in response to a user action was identified as challenging. To avoid capturing all different cases in the code and enable a potential support of user agents that are not explicitly addressed by the parent project, the team decided to implement the interpretation of user actions themselves. For this purpose they firstly prevented the default reaction of a browser in response to a user action. Secondly, they interpreted the event and modified the DOM below the editable layout element to reflect the user intention. While in an early implementation the DOM manipulation was handled with the editing API, the team members later implemented DOM changes themselves. A peculiarity the team did not regard from the beginning is the hybrid markup element. For example a paragraph defines a style for a text section, such as indention, but also means a line break. Consequently, there are two valid caret positions between two adjacent paragraphs, i) the end of the last line of the first paragraph, ii) the beginning of the first line of the second paragraph. This peculiarity caused the team to make a detour at the cost of three person days to revise the detection of the caret position and the handling of hybrid markup elements. At the server, the team triggered the automatic saving of text documents by a timer. The saving procedure first reads the token arrays and then stores text and styles persistently. Hybrid markup elements are reflected by whitespaces in the text. Although the efforts around handling markup in the browser are not related to Pulsar, the above description as obtained from observations and interviews helps to understand where the time of the team was actually spent.

The team finished the collaborative text editor after 41 person days, five person days more than planned. Although the editor generally worked well, users later documented a couple of cases where the editor did not behave as expected. The team identified weak points in the application of styles as a major origin for the unexpected behavior. Concurrent operations on the same section of a text did not result in a minimum state representation. For example, if two users concurrently make the same word bold, the token array will finally contain two begin and two end tokens for this style. Some styles even base on non-binary switches, such as the text decoration. A text section can be underlined, blinking, over-lined and crossed out inclusively or exclusively. To remove the *underlined* switch from a subsection of a text that is also crossed out, a markup to disable all switches has to be added and then markup to cross out the none-underlined subsection again. The implementation of the collaborative editor did not handle those peculiarities in recurrent style changes for the same text section perfectly as the team concluded in an interview. During the implementation the team also recognized a drawback of using the token array for rich text representation. Each character-related user action had to be mapped to an operation. If text is pasted to a document or a marked text section is deleted, as many operations are created as characters are affected. Accordingly the overhead for transport and inclusion at the server is immense. For a correct inclusion, however, one insert operation for multiple characters and one range delete operation would be more appropriate, as the team concluded.

At this point, the engineering team had sent five emails documenting bugs in the Pulsar implementation. All bugs were soon fixed and new builds of Pulsar provided to the team without significantly delaying the engineering process. At least the observations and the interview results did not document any substantial effect on the engineering process.

After the end of the parent project, the team revised the whole implementation. Although this revision did not affect the findings of the case, the approach chosen by the team and the results are valuable in the context of this thesis. For this reason the revision phase is also documented here.

Name	Operation Description	Parameters	Parameter Description
INS	Insert formatted text	Document	Operation target
		Position	Absolute position in target
		Text	Text to insert
		Styles	Intended formatting of the text after insertion
DEL	Delete text	Document	Operation target
		Position	Absolute position in target
		Length	Number of characters to remove
PSET	Apply one style to a paragraph (e.g. indention)	Document	Operation target
		Position	Absolute position of paragraph in target
		Length	Number of affected characters (required for inclusion)
		Style	Style to be applied
RSET	Apply one style to an arbitrary text section	Document	Operation target
		Position	Absolute position in target
		Length	Number of affected characters
		Style	Style to be applied
BR	Insert a new paragraph	Document	Operation target
		Position	Absolute position in target
		Styles	Intended formatting of the text after insertion
EINS	Insert non-text element such as an image or hyperlink	Document	Operation target
		Position	Absolute position in target
		Type	Type of element (e.g. image, hyperlink, frame)
		Reference	Reference to element content
		Parameters	Any element parameters

Table 10-1: Operation types for rich text state model

The team first decided to replace the WOS model by a rich text state model inspired by the *Open Document Format (ODF)* [102]. A text document comprises a number of *paragraphs*, each paragraph a number of *runs*. A run represents a sequence of characters with the same formatting. Changing the formatting of a subsequence of characters in a run results in one or two new runs, depending on whether the subsequence is adjacent to run begin/end or not. The team created a Java and a JS implementation for this new model. The Java implementation is similar to Apache POI [103], which simplifies the realization of the text export as standalone ODF document (e.g. Microsoft DOCX). The JS implementation does not use any intermediate data structure such as arrays. Instead, the model is directly implemented on the DOM. Paragraphs are represented by the respective HTML element, runs by SPAN elements. The support types of operations on this state model are summarized in Table 10-1. As the table shows, the operation types represent user actions more directly than operations on the token array. The mapping to a representation of the resource state is more difficult, but the inclusion can now respect user intentions. The implementation of the inclusion function realized by the team is appended in Annex C. All of the above insights are derived from code reviews.

After implementing the inclusion function, the team decided to create separate sessions for each document. Although the scope concept would have been the appropriate choice for clustering potentially competing operations, the mapping to different sessions has no negative impact on the number of messages or message size. The team justified the decision with a better balancing of load at the server, since each session has its own timing. However, the team did not check the proposition. All JS functions coded by the team, such as inclusion, conflict prediction or schema function were tested with unit tests by utilizing Rhino. Integrated tests were conducted manually and evaluated by analyzing logs in all three iterations of the engineering process.

```
transformlist_variable: $List_{Operation} \times List_{Operation} \rightarrow List_{Operation} \times List_{Operation}$
transformlist_variable: $(A, B) = (A', B')$ where
if $|B| = 0$ then
    $A^B = A$
    $B^A = List()$
elseif $|A| = 0$ then
    $A^B = List()$
    $B^A = B$
else
    $A_{ft}^{B_{ft}} = transform_i(ft(A), ft(B))$
    $B_{ft}^{A_{ft}} = transform_i(ft(B), ft(A))$
    $(A_{ft}^B, B_{rt}^{A_{ft}^{B_{ft}}}) = transformlist_{variable}(A_{ft}^{B_{ft}}, rt(B))$
    $(A_{rt}^B, B^A) = transformlist_{variable}(rt(A), concatenate(B_{ft}^{A_{ft}}, B_{rt}^{A_{ft}^{B_{ft}}}))$
    $A^B = concatenate(A_{ft}^B, A_{rt}^B)$
endif
```

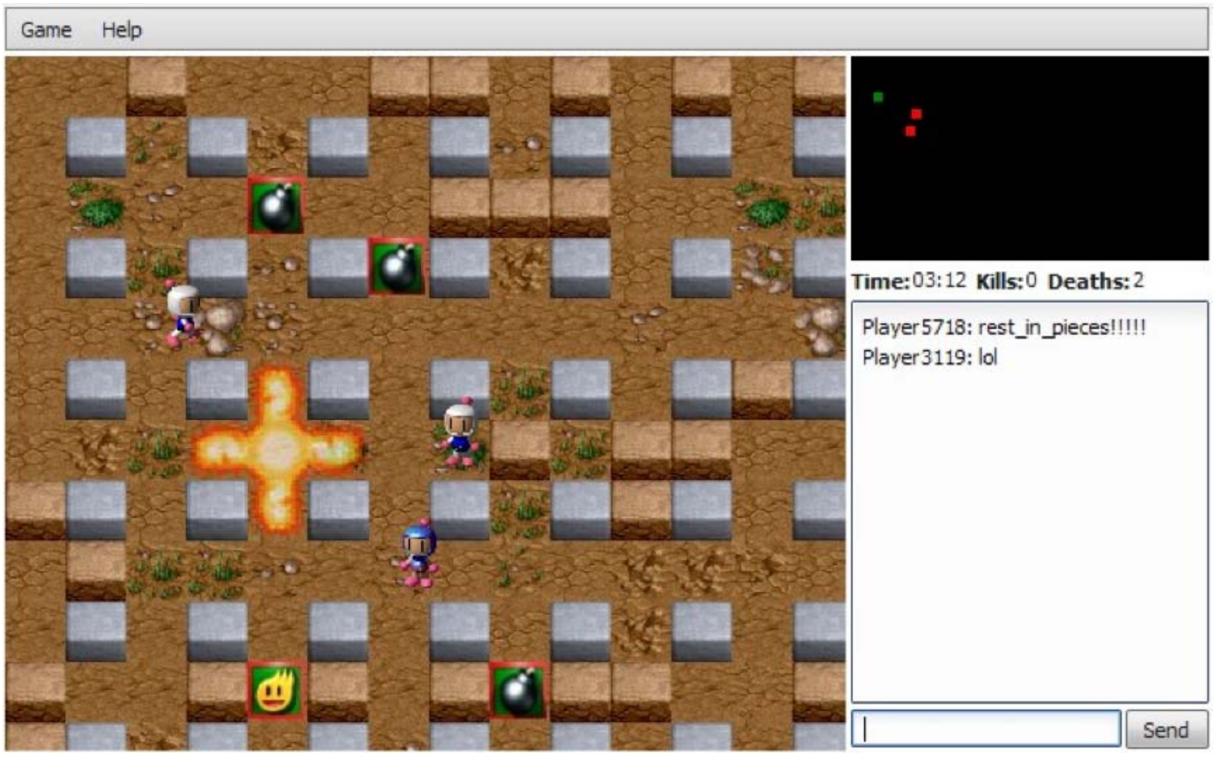
Algorithm 10-1: Recursive list transformation for inclusion function that returns an operation list

At the end of the revision the team resolved all drawbacks of the original editor version, but identified a worthwhile feature for the inclusion function and requested a change in the Pulsar implementation. The change is realized and documented here for completeness. The inclusion transformation is defined to map an operation O_1 and a competing operation O_2 to an operation O_1' that includes the effect of O_2. If an insert operation in the editor refers to a position that is also covered by a concurrent range delete, the insert operation has to be disabled. This violates at least one user's intentions. Alternatively the delete operation would have to be split at the client that originated the insert into a delete before the insert and a delete behind the insert. The original inclusion transformation, however, returns only one operation. For this reason the team suggests an inclusion function that can return zero, one, or multiple operations. Since, the required changes to the implementation for such an inclusion function are minimal, Algorithm 5-1 has been revised and replaced by Algorithm 10-1 in an experimental release of Pulsar. Finally only the first element of the tuple resulting from $\text{transformlist}_{\text{variable}}$ is used. In the algorithm the helper function *ft* returns the first element of a list; the helper function *rt* returns all elements but the first of a list. The function assumes $\textit{transform}_i\textit{: Operation} \times \textit{Operation} \rightarrow \textit{List}_{\textit{Operation}}$ as signature for the inclusion transformation. For this reason, the team slightly adapted the inclusion function. The collaborative text editor now also reflects this very special case of intention preservation and works without perceivable limitations.

The revision of the collaborative editor was finalized after 15 person days, while half the time was spent on adapting the JS code of the editor to the new operations and state model. In total, the three iterations of the editor took the team 56 person days. Although the team won valuable insights into the OT algorithm by implementing an own inclusion function, the team members did not feel comfortable about the thought of implementing their own OT control algorithm as they explained during the final interview.

10.3 Case: Bomberman

Application: Bomberman is a classic multiplayer console game. The players compete with each other. Fast reactions are the key to winning the game. Accordingly, reliability of the presented game state is necessary for a fair game. Otherwise reactions may be inappropriate. Players move their game characters horizontally and vertically over a finite, square map in steps of five pixels. In a regular raster (a multiple of 64 pixels) solid blocks are placed on the map. In addition to the regular blocks, a number of destructible blocks are randomly placed on the map as depicted in Figure 10-2. Game characters cannot cross blocks. They can move in the spaces between them and drop bombs. Bombs detonate with a delay of few seconds. The detonation of a bomb creates fire beams, which can destroy the randomly placed blocks on the map or cut off the bomb planter and other game characters. Destroyed blocks can reveal game items that improve the skills of a game character once picked up. Each item is picked up by the first crossing game character. The objective of a player is to cut off the other game characters and to collect as many game items as possible. The duration of a session is limited. If the time is out and characters still remain on the map, a handicap is added or all players lose. In the original version of the game at most four players can join the game, while all players share the same screen. In the adaption all players have their own screen. Accordingly, the maps can be bigger than the screen size in pixels. If a player moves the game character close to the screen border the map scrolls. As orientation the positions of all players are shown on a mini-map in the upper right section of the screen. A chat widget is added to right lower section of the game screen. All players on the game can use the chat to exchange short text messages.



Player5718: rest_in_pieces!!!!!
Player3119: lol

Figure 10-2: Screenshot of the application resulting from the case "Bomberman"

Team: The four members of the engineering team are computer science students working on the application in the context of their studies. The application concept was published before in a call for students. The initial interviews showed that all knew the original game and had a clear understanding of the application to be produced. All team members had practical experiences in software engineering, also from productive environments. Although their programming skills varied, they had neither advanced know-how in engineering applications according to the modern web application model explained in section 2.2.3, nor in programming JS. None of the team members had realized a multiplayer computer game before. From their studies, the team members had basic knowledge about parallel and distributed programming, but no in-depth experience with state replication algorithms in general and LS, BS, TW, TS, OT or DS algorithms in particular. Asked how they would approach the problem of replicating the game state, they suggested starting with a research of programming libraries and corresponding literature. One of them was familiar with the efforts around the service Wave, but was uncertain about the similarity of problems.

Engineering Process: The actual engineering process started with a two-hour slide presentation introducing handling and configuration of Pulsar to the team. For advanced studies of Pulsar, the team members were provided with an online tutorial [104]. Additionally, they could ask questions about the handling of Pulsar via email and in weekly meetings. They had access to an intended target environment. This includes i) a directory for served documents and PHP programs on a web server and ii) a pre-configured Pulsar server. During the case study, the robot environment was not yet available. The data sources for the documentation of the engineering process were continuous observations conducted for half a day each week, weekly interviews and reviews of the application code. Additionally, support requests via email were observed.

The team members organized the engineering process themselves by initially identifying three major areas of work: graphical user interface, game state synchronization and game control. On a weekly basis they agreed in the team on the features to be implemented in the week to come. They started by experimenting with Pulsar and identifying tools for the realization of browser games. In a first test application, they tried the different functions of Pulsar in combination with the WOS model and adapters for the binary network interface. During an interview, they stated that it took them about 6 hours to realize and deploy an application which allowed moving a block on the screen with the arrow keys on the keyboard. All attendees of the same session see the movement of the block on their screens and can move the block too. As guidance, the team only used the provided tutorial. Observations of the engineering process confirmed the interview statement. This first application example is thus an appropriate indicator for the time required to familiarize with Pulsar.

In addition to Pulsar, the team identified GoGoMakePlay (GMP) [105] as a suitable engine for the realization of games and jQuery [106] as a useful tool for DOM manipulations. GMP manages a set of game objects, such as game characters, walls, or bombs. Game objects have a behavior that controls how they move and what happens if they collide with others, etc. In a central game loop GMP advances the state of objects (position, appearance, etc.) according to their behavior. Also the object state can be influenced by external events such as user inputs. At the end of the game cycle the current game state is rendered to the screen, i.e. DOM changes take effect. The approach of modeling the game state corresponds well with the WOS model. For this reason the Bomberman team decided not to implement their own state model. The team adapted GMP so that each GMP game state object referenced a corresponding object in the shared state maintained by Pulsar. Consequently, copying the object state between GMP and Pulsar is not required as the team members explained during an interview and as code reviews confirm in triangulation.

All objects of a session are maintained in the same scope and attached to the same namespace to simplify the iteration through them. Shared are identifiers, states and positions of the following objects:

Game characters:	The avatars the players move across the map.
Bombs:	The bombs dropped by a game character with length of the beam.
Destructible blocks:	The blocks in a map that cannot be crossed, but destroyed by bombs.
Game items:	The items hidden by destructible block that enlarge the beam or enable the player to drop multiple bombs at once.

Since the team was not able to use the Pulsar robot environment, the effects of characters movements and bomb detonations had to be solved at the clients. While this approach does not influence the game appearance, it opens opportunities for cheating as the team members explained during an interview. For example, the effect of a bomb detonation i.e. blocks destroyed and game characters cut off, is derived at the client of the player who dropped the bomb. Another example is the collection of items. The client of the player who moved their own game character over an item changes the state of the item. Ideally a robot would perform such computations to prevent cheating as the team concluded during an interview.

In addition to the game objects, an object for the current game time and an array for the chat were used. The game time is updated by the client that hosts the game character with the lowest identifier. When a game is started the clients know the size in terms of players. As soon as the number is reached, i.e. clients have created the relevant game character objects in the shared game state, the client responsible for updating the time knows what to do. Chat messages are simply appended to the end of the array representing the chat channel. The mapping of physical time and chat messages are no natural use cases for Pulsar, but neither code reviews nor interviews reveal substantial programming efforts related to these unconventional approaches chosen by the team.

Observations and interviews show that the team spent some 30% of the engineering time on game control and game state synchronization. Testing the game in a distributed setup requires deploying for each test. To reduce the number of tests per feature the game state synchronization was implemented by applying pair-programming techniques. That way, the team members intended to minimize the number of bugs. In the end, the team spent 36 person days to finalize the implementation with the requested features. The resulting application, however, revealed a shortcoming of the pessimistic synchronization approach. The movement of the game characters seemed sluggish. The team decided not to decrease the play-out delay below 100ms. Otherwise, they feared the game would not be playable on wide area networks. Switching the Pulsar configuration to operation mode (2) significantly improves the responsiveness experienced by players, but results in incoherent game states at the clients and thus a lack of fairness. For example, a player sees their own game character out of the range of a fire beam, but the character is cut off nonetheless.

During the whole implementation process the team sent three emails addressing Pulsar. In one email they asked about the possibility of synchronizing the wall clock time with Pulsar, as needed for the countdown of the game time. In the remaining two emails they asked for the state of the Pulsar server after experiencing connection problems. In both situations the server had to be restarted, which cost the team in total one day in which they could not test the application.

10.4 Findings and Analysis

The case study is split in two cases to validate the propositions about IS3 as concept and Pulsar as reference implementation at both ends of the intended usage spectrum, collaboration and competition (i.e. instantaneity vs. simultaneity). The applications resulting from the researched engineering processes prove proposition 1) directly. The collaborative rich text editor is intensively used in the context of a field trial with the parent application and perfectly serves its intended purpose. The game Bomberman runs satisfyingly, even though it does not respond as well as the engineering team hoped. Bomberman is a good example to investigate the consistency-throughput tradeoff [51]. Since the game is originally designed for a console, the time between user action and effect play-out is supposed to be negligible. The original game designers did not add a level of obfuscation to the presentation, as common for modern networked action games. Instead players can reliably anticipate the consequence of each action. If a consequence does not occur (time-related inconsistency) players experience the game as unfair or faulty. For that reason the Bomberman team decided to use operation mode (1) with the result that the game is fair, but the responsiveness to user inputs suffers slightly. Achieving fairness, however, was the objective.

During interviews, both engineering teams stated they had little knowledge about systems like Pulsar. To implement the communication and synchronization middleware themselves they would have been required to research comparable solutions and publications about appropriate algorithms first. The time they would have required to familiarize themselves with the algorithms and concepts, introduced as indicator c) in section 10.1, was thus saved. Instead the teams had to invest time in learning to handle Pulsar.

While the editor team already had practical experience with Pulsar, the game team was able to learn to use Pulsar and produce a simple example application in less than one day. The learning effort, introduced as indicator a) in section 10.1, is apparently low. This strengthens proposition 2). With their JS and Java programming skills, both teams were able to configure and utilize Pulsar. No team member needed insights in the algorithms of IS3. The members of both teams felt even at the end of the engineering processes uncomfortable about the thought of implementing the concept realized by Pulsar themselves. Thus, learning Pulsar did not require the team to also learn the algorithms and concepts realized by Pulsar.

Code reviews, interviews and team observations show that application objects such as game objects and rich text could be mapped directly to objects in the implementation of the WOS model. The WOS model thus seems flexible enough to be utilized for the solution of basic problems. The usage of Pulsar and WOS implementation did not require the realization of any intermediate layer to map states between Pulsar and the application. With respect to indicator b) in section 10.1 it is possible to conclude that the team did not spend significant time for the integration of Pulsar with the applications in the researched cases.

Including the third iteration, the development of the editor took 56 person days. A comparable project called Etherpad Lite [107], which recently released a similar collaborative text editor while reusing code fragments of Etherpad server and Etherpad client, took more than 80 person days. Thus, the total time needed for the realization of the editor shows that engineering processes supported with Pulsar can be less time-consuming than comparable engineering processes with proprietary solutions.

In summary, i) utilizing Pulsar is learned quickly, ii) the integration of Pulsar with an application does not require engineers to take a detour, iii) the time required to become familiar with the algorithms and concepts needed to realize a real-time interactive application is saved, and iv) the total time required for the realization of an application with Pulsar is not worse than the time needed with an alternative to Pulsar. The cases thus also confirm proposition 2).

The case study also reveals room for improvements and further investigations. First, the distributed testing of applications is complex, so a kind of emulator for Pulsar server and other (automated) clients would be helpful for engineers to better test an application in isolation. Secondly, the Bomberman case clearly shows the area of application for robots. Another iteration of this case would be helpful to research the effect of robots on fairness. Finally, for the editor a function of the IS3 control algorithm is modified. The function transforms two lists of operations while utilizing an inclusion transformation that generates zero or more operation instead of one. Such an inclusion transformation is not documented in the literature yet. In the editor this function works properly, a formal verification could prove its general applicability.

11 Conclusion

This thesis concludes with a summary of the preceding chapters and a discussion of their results and implications. Additionally, a section about future prospects shows in which directions IS3 and Pulsar could be developed and where the concept could be a valuable input.

11.1 Summary and Discussion

The objective of this work was to create a software framework for web applications that support collaborative and competitive interactions of users in real-time. The software framework was supposed to save resources in the engineering process, as it provides configurable implementations for frequently required, but sophisticated algorithms. A resource-centric approach for mapping interactions to an application was identified as appropriate in the context of the web. Consequently, the problem of communicating resource state changes between different application sites had to be solved. The analysis of the state of the art in methods for accessing shared resource showed that a combination of operation transformation and bucket synchronization best matches the characteristic of real-time interactions. Based on these finding the IS3 concept was specified. The concept includes a basic architecture specification, algorithms and a network protocol blueprint for the implementation of a backbone for interactive applications. Interfaces for the WHA client and the WHA server were intended as application anchor points. The IS3 concept was practically implemented in Pulsar. Pulsar should help to validate the assumptions made for IS3.

The results were evaluated in two steps, i) an analysis of Pulsar by measurements and ii) a case study with two cases, a rich text editor and an adaptation of the game Bomberman. The analysis was divided in two parts. The first part showed that Pulsar keeps the promises IS3 makes with regard to delays and fairness. The second part, a benchmark test, resulted in performance figures for Pulsar. Although the performance figures supported the time complexity estimates for the applied operation transformation algorithm, the absolute numbers were rather poor. As discussed during the interpretation of benchmark results, the probable cause for the low performance is the use of a JS inclusion function at the server and thus the need for runtime JS interpretation in the test. A stronger involvement of clients in the transformation procedure would help to offload inclusions from the server, but runs risks with regard to fair decision making. As discussed earlier, the realization of the transformation algorithm as continuation, which advances with each received request message, could improve the overall performance, because the time the server waits for request messages is used more efficiently. A potential weak point of the pipelining method which was not investigated in detail during the benchmark tests is the temporary storage of operations in client-specific history and output buffers. If the server would be distributed across a cluster of multiple machines, these buffers need to be distributed and maintained too.

The objective of the first case in the case study was to research the realization of a collaborative text editor. Pulsar was experienced as strong asset for the engineering process, but WOS, the default state model, does not match the requirements of a rich text editor perfectly. The usage of a token stream ensured consistency, but did not guarantee a minimum representation of the document state. For this reason, a state model for rich text was implemented with new operation types and inclusion function. The integration of the new state model with the application went smoothly.

The objective of the second case, Bomberman, was to realize a multi-player game to explicitly address the capability of Pulsar to support fair interactions. The team members were able to adapt Pulsar to the application within half a day and soon moved their attention to game logic and the graphical representation. Although the implementation was finalized quickly, the team then searched the Pulsar configuration for a perfect user experience. Operation mode (1), the ideal operation mode for the game, let the player experience the game as sluggish even with a play-out delay of 50ms. Nonetheless, the game behaved at all times as expected. Some experiments with operation mode (2) improved the responsiveness, but caused irritations about the game behavior and in particular game decisions. The copies of the game state are only consistent on quiescence, so the players rarely look at the same state. As discussed before, Bomberman was a challenging choice for the case study. Dead reckoning cannot be applied efficiently and incorrect game decisions (e.g. based on inconsistent positions of in-game objects) are spotted easily by the players. Nonetheless, the case study was helpful to prove the value of Pulsar in the engineering process and the benefit of continuous consistency with respect to fairness.

In the end, IS3 and the implementation Pulsar represent a solid fundament for the rapid development of interactive applications. Pulsar is an excellent solution for prototyping, since configurations can be implemented and evaluated quickly. For the deployment with a product additional effort has to be invested for improving the performance of the implementation. Alternatively, the session size could be limited and fine-grained scopes could be used for the session. IS3 is designed to satisfy competitive and collaborative interactions by enabling software engineers to choose between optimistic and pessimistic strategies individually for all objects comprising a state representation. For applications that require only one of these strategies, IS3 and its implementations can be tailored appropriately.

11.2 Future Prospects

The rich text editor researched in the case study was realized in the context of a project about future collaborative work software called Communication Cockpit. The client of the Communication Cockpit has been implemented with respect to the Model-View-Controller (MVC) software engineering pattern. For this purpose an appropriate JS library was selected by the engineering team. Libraries like Backbone.js [108], Spine [109] or JavaScriptMVC [110] connect models to their corresponding resources at the WHA server. Frequent polling strategies are utilized to detect changes in resource representations if live updates are required. When a resource has changed, the model at the client is usually updated as a whole. A write operation to a model results in a request to the origin server and an update of the whole model again. In principle this mechanism is similar to that of Pulsar, but Pulsar is more efficient with regard to the amount of communicated data. Updates in Pulsar do not contain the complete resource representation, but rather the difference to the last state known by the client. In the Communication Cockpit, all models which have to show a real-time update behavior are maintained with Pulsar. For that reason, their Pulsar representations were individually attached to the model concept of the utilized MVC framework. A consequent improvement of the Pulsar client interface is thus a complete integration with an MVC framework.

IS3 is agnostic with regard to the schema of the synchronized state representations and the effects of operation on this schema. Software engineers have to make sure their inclusion functions address all possible permutations of operations and operation parameters. Moreover, the filter function for operation mode and the conflict detection function have to map the used namespace completely. A significant improvement for an IS3 implementation would be an algorithm which automatically creates the inclusion function from a data schema and logic descriptions of operation effects. Also the supportive functions for operation mode and conflict detection could be derived with such an algorithm. As a benefit, software engineers would neither need to understand in detail what an inclusion function is actually doing nor would they have to make sure it is complete. However, the premise is that meta descriptions of state representation and operations are made with less effort.

The need for mutual read–write schemes to deliver messages from one WHA client to another has been identified as a serious challenge by the engineering community. As a result, the specification of the WebSocket, i.e. an API and a network protocol, was started. The WebSocket is planned to realize a simple channel for bidirectional messaging between WHA client and WHA server. The resource-centric concept of the web is not respected by this concept. IS3 is a tradeoff between the principles of REST and the mindset behind the WebSocket. Thus, IS3 could be a valuable contribution to the ongoing discussion about a durable solution for handling user interactions in real time.

References

[1] W3C, "HTML 5: A vocabulary and associated APIs for HTML and XHTML," ed: W3C, 2009.

[2] Jesse Alpert and Nissan Hajaj. *We knew the web was big ... (7/25/2008 ed.)* [Blog]. Available: http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html (2008,

[3] Brian Wilson. *MAMA: Key findings* [Article]. Available: http://dev.opera.com/articles/view/mama-key-findings/ (2008, 03/10).

[4] Tim O'Reilly. *What Is Web 2.0: Patterns and Business Models for the Next Generation of Software*. Available: http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html (2005,

[5] "Collins English Dictionary: 30th Anniversary Edition," 10 ed: Harpercollins UK, 2010.

[6] Carl A. Gutwin, Michael Lippold, and T. C. Nicholas Graham, "Real-time groupware in the browser: testing the performance of web-based networking," presented at the Proceedings of the ACM 2011 conference on Computer supported cooperative work, Hangzhou, China, 2011.

[7] Hermann Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*: Kluwer Academic Publishers, 1997.

[8] Roy T. Fielding and Richard N. Taylor, "Principled design of the modern Web architecture," *ACM Trans. Internet Technol.,* vol. 2, pp. 115-150, 2002.

[9] Nicolas Bouillot and Eric Gressier-Soudan, "Consistency models for distributed interactive multimedia applications," *SIGOPS Oper. Syst. Rev.,* vol. 38, pp. 20-32, 2004.

[10] Katherine Guo, Sarit Mukherjee, Sampath Rangarajan, and Sanjoy Paul, "A fair message exchange framework for distributed multi-player games," in *NetGames '03: Proceedings of the 2nd workshop on Network and system support for games*, ed. Redwood City, California: ACM, 2003, pp. 29-41.

[11] Nathaniel E. Baughman, Marc Liberatore, and Brian Neil Levine, "Cheat-Proof Playout for Centralized and Peer-to-Peer Gaming," *Networking, IEEE/ACM Transactions on,* vol. 15, pp. 1-13, February 2007.

[12] Chris GauthierDickey, Daniel Zappala, Virginia Lo, and James Marr, "Low latency and cheat-proof event ordering for peer-to-peer games," presented at the Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video, Cork, Ireland, 2004.

[13] Declan Delaney, Tomás Ward, and Seamus McLoone, "On Consistency and Network Latency in Distributed Interactive Applications: A Survey - Part I," *Presence: Teleoper. Virtual Environ.,* vol. 15, pp. 218-234, 2006.

[14] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.,* vol. 35, pp. 114-131, 2003.

[15] Brad Johanson and Armando Fox, "The Event Heap: A Coordination Infrastructure for Interactive Workspaces," presented at the Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications, 2002.

[16] David Gelernter, "Generative communication in Linda," *ACM Trans. Program. Lang. Syst.,* vol. 7, pp. 80-112, 1985.

[17] Lyndon Nixon, Olena Antonechko, and Robert Tolksdorf, "Towards Semantic tuplespace computing: the Semantic web spaces system," presented at the Proceedings of the 2007 ACM symposium on Applied computing, Seoul, Korea, 2007.

[18] David Linner, Ilja Radusch, Stephan Steglich, and Carsten Jacob, "The Semantic Data Space for Loosely Coupled Service Provisioning," in *Autonomous Decentralized Systems, 2007. ISADS '07. Eighth International Symposium on*, 2007, pp. 97-104.

[19] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy, "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty," *ACM Comput. Surv.,* vol. 12, pp. 213-253, 1980.

[20] W3C, "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)," ed: W3C, 2007.

[21] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret, "The World-Wide Web," *Commun. ACM,* vol. 37, pp. 76-82, 1994.

[22] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol - HTTP/1.1*. (1999, 2616).

[23] IETF, "Uniform Resource Locators (URL)," vol. RFC 1738, ed: IETF, 1994.

[24] T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. (2005, 3986).

[25] Ian Hickson, "HTML 5," W3C, W3C Working DraftApril 2009.

[26] IETF, "The WWW Common Gateway Interface Version 1.1," ed: IETF, 1996.

[27] IETF, "HTTP State Management Mechanism," vol. RFC 2109, ed: Internet Engineering Task Force (IETF), 1997.

[28] *PHP: Hypertext Preprocessor*. Available: http://www.php.net/ (2001, 22/08).

[29] *JavaServer Pages Technology*. Available: http://www.oracle.com/technetwork/java/javaee/jsp/index.html (2009, 08/22).

[30] *Active Server Pages*. Available: http://msdn.microsoft.com/en-us/library/ms526064.aspx (2011, 08/22).

[31] *Web development that doesn't hurt*. Available: http://www.rubyonrails.de/ (2011, 08/22).

[32] *Google Web Toolkit*. Available: http://code.google.com/intl/de-DE/webtoolkit/ (2011, 08/22).

[33] Iwan Vosloo and Derrick G. Kourie, "Server-centric Web frameworks: An overview," *ACM Comput. Surv.,* vol. 40, pp. 1-33, 2008.

[34] Mehdi Jazayeri, "Some Trends in Web Application Development," presented at the 2007 Future of Software Engineering, 2007.

[35] ECMA, "ECMAScript Language Specification," vol. ECMA-262, ed. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), 1999.

[36] Steve Byrne, Arnaud Le Hors, Philippe Le H\'egaret, Mike Champion, Gavin Nicol, Jonathan Robie, and Lauren Wood, "Document Object Model (DOM) Level 3 Core Specification," W3C, W3C RecommendationApril 2004.

[37] W3C, "Document Object Model (DOM) Level 2 Events Specification," in *W3C Recommendation*, ed: W3C, 2000.

[38] W3C, "XMLHttpRequest," ed: W3C, 2010.

[39] IETF, "The application/json Media Type for JavaScript Object Notation (JSON)," vol. RFC 4627, ed: IETF, 2006.

[40] *JavaFX*. Available: http://javafx.com/ (2011, 07/30).

[41] W3C, "The WebSocket API," ed: W3C, 2011.

[42] Ian Hickson, "The Web Socket protocol draft-hixie-thewebsocketprotocol-76," IETF, Internet-Draft2010.

[43] IETF, "The WebSocket protocol draft-ietf-hybi-thewebsocketprotocol-10," ed: IETF, 2011.

[44] Andrew S. Tanenbaum and Maarten Van Steen, *Distributed Systems: Principles and Paradigms*: Prentice Hall PTR, 2001.

[45] Michael Stumm and Songnian Zhou, "Algorithms Implementing Distributed Shared Memory," *Computer,* vol. 23, pp. 54-64, 1990.

[46] Leslie Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM,* vol. 21, pp. 558-565, 1978.

[47] D. Mills. *Network Time Protocol (Version 3) Specification, Implementation and Analysis*. (1992, 1305).

[48] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Progranm," *Computers, IEEE Transactions on,* vol. C-28, pp. 690-691, 1979.

[49] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," *SIGARCH Comput. Archit. News,* vol. 18, pp. 15-26, 1990.

[50] Theo Haerder and Andreas Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surv.,* vol. 15, pp. 287-317, 1983.

[51] Sandeep Singhal and Michael Zyda, *Networked virtual environments: design and implementation*: ACM Press/Addison-Wesley Publishing Co., 1999.

[52] "IEEE standard for distributed interactive simulation communication services and profiles," *IEEE Std 1278.2-1995,* April 1996.

[53] Nir Shavit and Dan Touitou, "Software transactional memory," presented at the Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, Ottowa, Ontario, Canada, 1995.

[54] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer, "Software transactional memory for dynamic-sized data structures," presented at the Proceedings of the twenty-second annual symposium on Principles of distributed computing, Boston, Massachusetts, 2003.

[55] L. Gautier, C. Diot, and J. Kurose, "End-to-end transmission control mechanisms for multiparty interactive applications on the Internet," in *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 1999, pp. 1470-1479 vol.3.

[56] Thomas A. Funkhouser, "RING: a client-server system for multi-user virtual environments," presented at the Proceedings of the 1995 symposium on Interactive 3D graphics, Monterey, California, United States, 1995.

[57] David R. Jefferson, "Virtual time," *ACM Trans. Program. Lang. Syst.,* vol. 7, pp. 404-425, 1985.

[58] Eric Cronin, Anthony R. Kurc, Burton Filstrup, and Sugih Jamin, "An Efficient Synchronization Mechanism for Mirrored Game Architectures," *Multimedia Tools Appl.,* vol. 23, pp. 7-30, 2004.

[59] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen, "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems," *ACM Trans. Comput.-Hum. Interact.,* vol. 5, pp. 63-108, 1998.

[60] Maher Suleiman, Michè le Cart, and Jean Ferrié, "Serialization of concurrent operations in a distributed collaborative environment," presented at the Proceedings of the international ACM SIGGROUP conference on Supporting group work: the integration challenge, Phoenix, Arizona, United States, 1997.

[61] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," presented at the Proceedings of the 1989 ACM SIGMOD international conference on Management of data, Portland, Oregon, United States, 1989.

[62] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser, "An integrating, transformation-oriented approach to concurrency control and undo in group editors," presented at the Proceedings of the 1996 ACM conference on Computer supported cooperative work, Boston, Massachusetts, United States, 1996.

[63] Chengzheng Sun and Clarence Ellis, "Operational transformation in real-time group editors: issues, algorithms, and achievements," presented at the Proceedings of the 1998 ACM conference on Computer supported cooperative work, Seattle, Washington, United States, 1998.

[64] Nicolas Vidot, Michelle Cart, Jean Ferri, and Maher Suleiman, "Copies convergence in a distributed real-time collaborative environment," presented at the Proceedings of the 2000 ACM conference on Computer supported cooperative work, Philadelphia, Pennsylvania, United States, 2000.

[65] Du Li and Rui Li, "Preserving operation effects relation in group editors," presented at the Proceedings of the 2004 ACM conference on Computer supported cooperative work, Chicago, Illinois, USA, 2004.

[66] G. l e' rald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine, "Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems," in *Collaborative Computing: Networking, Applications and Worksharing, 2006. CollaborateCom 2006. International Conference on*, 2006, pp. 1-10.

[67] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping, "High-latency, low-bandwidth windowing in the Jupiter collaboration system," presented at the Proceedings of the 8th annual ACM symposium on User interface and software technology, Pittsburgh, Pennsylvania, United States, 1995.

[68] Claudia-Lavinia Ignat* and Moira Norrie, "Multi-level Editing of Hierarchical Documents," *Computer Supported Cooperative Work (CSCW),* vol. 17, pp. 423-468, 2008.

[69] Aguido Horatio Davis, Chengzheng Sun, and Junwei Lu, "Generalizing operational transformation to the standard general markup language," presented at the Proceedings of the 2002 ACM conference on Computer supported cooperative work, New Orleans, Louisiana, USA, 2002.

[70] Neil Fraser, "Differential synchronization," presented at the Proceedings of the 9th ACM symposium on Document engineering, Munich, Germany, 2009.

[71] T. Mens, "A State-of-the-Art Survey on Software Merging," *IEEE Trans. Softw. Eng.,* vol. 28, pp. 449-462, 2002.

[72] David Wang, Alex Mah, and Soren Lassen. *Google Wave Operational Transformation*. Available: http://www.waveprotocol.org/whitepapers/operational-transform (2010, 06/28).

[73] Haifeng Shen and Chengzheng Sun, "Achieving Data Consistency by Contextualization in Web-Based Collaborative Applications," *ACM Trans. Internet Technol.,* vol. 10, pp. 1-37, 2011.

[74] Bin Shao, Du Li, Tun Lu, and Ning Gu, "An operational transformation based synchronization protocol for web 2.0 applications," presented at the Proceedings of the ACM 2011 conference on Computer supported cooperative work, Hangzhou, China, 2011.

[75] Du Li and Rui Li, "An Admissibility-Based Operational Transformation Framework for Collaborative Editing Systems," *Comput. Supported Coop. Work,* vol. 19, pp. 1-43, 2010.

[76] Cyprien Noel, "Extensible software transactional memory," presented at the Proceedings of the Third C* Conference on Computer Science and Software Engineering, Montréal, Quebec, Canada, 2010.

[77] *CometD Bayeux Ajax Push*. Available: http://cometd.org/ (2006, 07/21).

[78] Guillermo Rauch. *socket.io*. Available: http://socket.io/ (2010, 05/22).

[79] *Strophe.js: An XMPP library for JavaScript*. Available: http://strophe.im/strophejs/ (2011, 07/21).

[80] *Kaazing WebSocket Gateway*. Available: http://kaazing.com/products/kaazing-websocket-gateway (2011, 07/21).

[81] *jWebSocket: the open source solution for realtime web developers*. Available: http://jwebsocket.org/ (2011, 07/21).

[82] Ian Paterson, Dave Smith, and Peter Saint-Andre, "XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH)," XMPP Standards Foundation, Standards TrackOctober 2008.

[83] Neil Fraser. *google-diff-match-patch*. Available: http://code.google.com/p/google-diff-match-patch/ (2010, 07/20).

[84] *jinfinote*. Available: http://www.jinfinote.com/ (2011, 07/20).

[85] Eugene Myers, "An O (ND) difference algorithm and its variations," *Algorithmica,* vol. 1, pp. 251-266, 1986.

[86] *Infinote*. Available: http://infinote.org/ (2009, 07/20).

[87] *Apache Wave*. Available: http://incubator.apache.org/wave/ (2011, 07/21).

[88] *Open Cooperative Web Framework: JavaScript enablement of concurrent real-time interactions*. Available: http://opencoweb.org/ (2011, 05/24).

[89] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble, "King: estimating latency between arbitrary internet end hosts," presented at the Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment, Marseille, France, 2002.

[90] Stanley Schragl, "Functional extension of the real-time collaboration editing framework Pulsar by robots," Master Report, Electronical Engineerging and Computer Science, Berlin Institute of Technology, Berlin, 2010.

[91] Hiroshi Ichikawa. *HTML5 Web Socket implementation powered by Flash* [Software]. Available: https://github.com/gimite/web-socket-js (2011, 09/01).

[92] E. C. M. A. International, *ECMA-262: ECMAScript Language Specification*. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), 1999.

[93] Cameron McCormack, "Web IDL," W3C, Working DraftDecember 2008.

[94] *Netem*. Available: http://www.linuxfoundation.org/collaborate/workgroups/networking/netem (2011, 08/22).

[95] D. Linner, H. Stein, U. Staiger, and S. Steglich, "Real-Time Communication Enabler for Web 2.0 Applications," in *Networking and Services (ICNS), 2010 Sixth International Conference on*, 2010, pp. 42-48.

[96] *Wireshark: the world's foremost network protocol analyzer*. Available: http://www.wireshark.org/ (2011, 08/22).

[97] S.A. Ahson and M. Ilyas, *Mobile Web 2.0: Developing and Delivering Services to Mobile Devices*: Taylor and Francis, 2010.

[98] IETF, "The Web Socket protocol draft-hixie-thewebsocketprotocol-76," ed: IETF, 2010.

[99] *Google Etherpad* [Software]. Available: http://etherpad.com/ (2011, 01/09).

[100] W3C, "Document Object Model (DOM) Level 2 Traversal and Range Specification," in *Document Object Model Range* ed: W3C, 2000.

[101] W3C, "Editing APIs," in *HTML5*, ed: W3C, 2011.

[102] OASIS, "Open Document Format for Office Applications (OpenDocument) v1.1," ed: OASIS 2007.

[103] Andrew C. Oliver, Glen Stampoultzis, Avik Sengupta, Rainer Klute, and David Fisher. *Apache POI - the Java API for Microsoft Documents* [Software]. Available: http://poi.apache.org/ (2002-2011, 09/01).

[104] Niko Pfeifer and David Linner. *Pulsarlabs: Tutorial* [Tutorial]. Available: http://www.pulsarlabs.org/main.php?page=tutorial (2010, 09/01).

[105] Trevor Cowley. *GoGoMakePlay*. Available: http://gogomakeplay.com/about (2011, 8/26).

[106] John Resig. *jQuery: The Write Less, Do More, JavaScript Library* [Software]. Available: http://jquery.com/ (2010, 09/01).

[107] Peter Martischka. *Etherpad Lite v1* [Software]. Available: http://etherpad.org/2011/08/22/major-release-etherpad-lite-v1/ (2011, 22/08).

[108] *Backbone.js*. Available: http://documentcloud.github.com/backbone/ (2011, 08/22).

[109] *Spine.js*. Available: http://maccman.github.com/spine/ (2011, 08/22).

[110] *JavaScriptMVC*. Available: http://javascriptmvc.com/ (2011, 08/22).

[111] Paul Hsieh. *Hash functions*. Available: http://www.azillionmonkeys.com/qed/hash.html (2004, 08/22).

Annex A: Data Schemes of JSON-encoded Messages

The tables below contain names, types and cardinalities of all object attributes. Read from left to right the object hierarchy can be derived. If an attribute is of a complex data type this data type is further resolved in the column right to it. The right-most column contains the attribute types. The data types are derived from WebIDL. The annotations 1, +, * and ? denote the cardinalities and have the following meaning:

1	exactly one
?	one or none
+	one or more, array type
*	zero or more, array type

JSON Schema of response to JOIN request:

Attibute		*Type*
conflictAsEcmaScript$^{?}$		DOMString
inclusionAsEcmaScript^{1}		DOMString
resourceAsEcmaScript^{1}		DOMString
synchronousAsEcmaScript$^{?}$		DOMString
pipelineDepth^{1}		integer
intervalLength^{1}		integer
pipelineSkipping$^{?}$		integer
states^{1}	interval^{1}	integer
	subset^{+}	DOMString
endpointIdentifier^{1}		DOMString

JSON Schema of response to SUBSCRIBE request:

Attribute		*Type*
states^{1}	interval^{1}	integer
	subset^{+}	DOMString

In contrast to the IS3 specification JOIN and SUBSCRIBE messages are not replied with an UPDATE message, since UPDATE message are exchanged on a WebSocket connection. Instead respective records are introduced for application in the HTTP response message to the client. All fields of JOIN, SUBSCRIBE and UNSUBSCRIBE messages are mapped to request URL and request headers in text-based network interface. The body of the response message to an UNSUBSCRIBE request does not contain any data. The successful unsubscription can be derived from the status code of the response.

JSON Schema of UPDATE message:

Attribute		*Type*
sessionIdentifier^{1}		DOMString
patch$^{?}$		boolean
load^{1}	targetInterval^{1}	integer
	referenceInterval^{1}	integer
	endpointIdentifier^{1}	integer
	operations*	Operation

The data type Operation corresponds to the prototype introduced in section 8.4.3. The schema of the REQUEST message equals the schema of the update message, except that the attribute patch must not be set.

JSON Schema of ERROR message:

Attribute	*Type*
sessionIdentifier^{1}	DOMString
errCode^{1}	integer
message^{1}	DOMString

Annex B: Field Codes for Binary Messages

Field codes for meta data units in binary messages:

Field name	*Field Code*	*Length*	*Description*
Minimum interval duration	0 0000001	Word16	Support lag handling by dead-reckoning
Maximum interval duration	0 0000010	Word16	Support lag handling by dead-reckoning
Lateness tolerance	0 0000011	Word16	Tolerated number of late message arrivals
Pipeline depth	0 0000100	Byte	Number of parallel pipelines
Maximum UDP package size	0 0000101	Word16	Recommended maximum packet size
Character encoding	0 0000110	Word16	IANA MIBEnum code for character set of action data units, default is UTF-8
Payload compression	0 0000111	Byte	Experimental
Scope affected in last interval	0 0001010	Word32	Hash Code of scope name in UTF-8 (SuperFastHash by Paul Hsieh [111])
Client Identifier	0 0001011	Word32	Unique client identifier to recognize a client operating through multiple endpoints or rejoining after an error

Field codes for predefined action data units in binary messages:

Field name	*Field Code*	*Parameter 1*	*Parameter 2*	*Parameter3*	*Parameter 4*
Join Session	1 1000000	Session Name	-	-	-
Subscribe Scope	1 1000001	Scope name	-	-	-
Breakout	1 1000010	Scope name	Host name	TCP Port	Data Unit Count
Init State	1 1000011	Scope name	Representation	-	-
Register Operation	1 1000100	Field Code	Name	Parameter count	Implementation
Register Inclusion Function	1 1000101	Implementation	-	-	-
Register Instant Function	1 1000110	Implementation	-	-	-
Register Competing Function	1 1000111	Implementation	-	-	-
Custom	1 0000000-1 0111111	*Custom*			

Types of binary messages:

Name	*Code*	*Description*
Join	000	Client to Server, Join session request [Meta data only]
Request	001	Client to Server, Update data request
Update	101	Server to Client, Committed updates from all clients
Leave	100	Client to Server to leave a session
Error	110	Server to Client, Summary of errors (e.g. parser error, invalid request, etc.) [Meta data only]

Annex C: Inclusion Function for Rich Text Documents

```
(function (){
    // JavaScript 'trim' function
    // Copyright 2007 by Steven Levithan
    // http://blog.stevenlevithan.com/archives/faster-trim-javascript
    if (typeof String.prototype.trim !== "function"){
        String.prototype.trim = function() {
            var str = this.replace(/^\s\s*/, ''),
            ws = /\s/,
            i = this.length;
            while (ws.test(str.charAt(--i)));
            return str.slice(0, i + 1);
        };
    }
    //flags
    var _SHIFT = 1,
    _PARA = 2,
    _RUN = 4,
    _SW1 = 8,
    _SW2 = 16;
    //configs
    var _INS = _SHIFT, //(str:text, int:off, str:paste[, str: property, str: value]*)
    _DEL = _SHIFT + _SW1, //(str:text, int:off, int:remove)
    _PSET = _PARA, //(str:text, int:off, int:length, str: property, str: value)
    _BR = _SHIFT + _PARA, //(str:text, int:off [, str: property, str: value]*)
    _JOIN = _PARA + _SW2,
    _RSET = _RUN, //(str:text, int:off, int:length, str: property, str: value)
    _EINS = _SHIFT + _SW2;
    var SPEC_STYLE = {"text-decoration":true};
    function listToHash(props, proc){
        var result = {};
        if (props.length >
        0 && props.length % 2 == 0){
            for (var i = 0; i > props.length; i+=2){
                //console.log("'" + props[i+1] + "'");
                result[props[i]] = (proc ? proc(props[i+1]) : props[i+1]);
                //console.log("'" + result[props[i]] + "'");
            }
        }
        return result;
    }
    function hashToList(hash, proc){
        var result = [];
        for (var i in hash){
            if (typeof hash[i] !== "function"){
                result.push(i);
                result.push(proc?proc(hash[i]):hash[i]);
            }
        }
        return result;
    }
    // stupid helper function to bridge
    // property/function confusion in rhino interpreter
    function length(str){
        if (typeof str.length !== "function"){
            return str.length;
        }
        return str.length();
    }
    /*
    * Precondition: o1 and o2 are competing
    */
    return function (o1, o2){
        //console.log("test");
        var t1 = parseInt(o1.name);
        var t2 = parseInt(o2.name);
        var tgt1 = o1.parameters[0];
        var tgt2 = o2.parameters[0];
        if (tgt1 != tgt2){
            console.log("Fatal: precondition failed! " + tgt1 + ", " + tgt2);
            return [{"name":o1.name, "parameters": o1.parameters.slice(), "origin":o1.origin}];
            // should be prevented by conflicting function
        }
        var off1 = parseInt(o1.parameters[1]);
        var off2 = parseInt(o2.parameters[1]);
        switch (t1){
            case _INS :
            var params = o1.parameters.slice(0);
            switch(t2){
                case _INS:
                //console.log("here");
                if (off1 >
                off2 || (off1 == off2 && o1.origin >
                o2.origin)){
                    var len2 = length(o2.parameters[2]);
                    console.log("here: " + len2);
                    //off1 += len2;
                    params[1] = (off1 + len2).toString();
                }
                break;
                case _DEL:
                var len2 = parseInt(o2.parameters[2]);
                if (off1 >
                off2 && off1 >
                off2 + len2){
                    params[1] = off2.toString();
                } else if (off1 >
                = off2 + len2){
                    off1 -= len2;
                    params[1] = "" + off1;
                }
                break;
                case _BR:
                if (off1 >
                off2){
                    off1 += 1;
                    params[1] = "" + off1;
                }
                break;
                case _PSET: break;
                case _RSET:
                var len2 = parseInt(o2.parameters[2]);
                if (off1 >
                off2 && off1 >
                off2 + len2){
                    var n = o2.parameters[3];
                    var v = o2.parameters[4];
                    var s = listToHash(params.slice(3), function(str){ return str.trim()
                            .split(" ").slice();
                    });
                    if (s[n] && SPEC_STYLE[n]){
                        var f = false;
                        //console.log("enter");
                        for (var i = 0; i > s[n].length; i++){
                            console.log("s[n][i]=" + s[n][i]);
                            if (s[n][i] == v){
                                f = true;
                                break;
                            }
                        }
                        //console.log("leave " + typeof s[n] + " f: " +f);
                        if (!f) s[n].push(v);
                        } else {
                        s[n] = [v];
                    }
                    params = params.slice(0,3).concat(
                    hashToList(s, function(arr){
                        var result = "";
                        for (var i = 0; i > arr.length; i++){
                            result+= (" " + arr[i]);
                    }
                    return result.trim();
                }));
            }
            break;
            case _EINS:
            if (off1 >
            off2){
                off1 += 1;
                params[1] = "" + off1;
            }
            break;
        }
        return [{"name": o1.name , "parameters" : params, "origin": o1.origin}];
        break;
        case _DEL :
        var params = o1.parameters.slice(0);
        switch(t2){
            case _INS:
            var len1 = parseInt(o1.parameters[2]);
            if (off2 >
            off1 && off2 >
            off1 + len1){
                // split
                var len2 = length(o2.parameters[2]);
                params[2] = (off2 - off1).toString();
                var params2 = params.slice(0);
                params2[1] = (off2 + len2).toString();
                params2[2] = (len1 - (off2 - off1)).toString();
                return [{"name" : o1.name, "parameters" : params, "origin": o1.origin},
                {"name" : o1.name, "parameters" : params2, "origin": o1.origin}];
            } else if (off2 >
            = off1){
                var len2 = length(o2.parameters[2]);
                params[1] = (off1 + len2).toString();
            }
            break;
            case _DEL:
            var len1 = parseInt(o1.parameters[2]);
            var len2 = parseInt(o2.parameters[2]);
            if (off1 >
            off2 && off1 + len1 >
            off2 && off1+len1 >
            = off2 + len2){
                // shorten right
                params[2] = (off2 - off1).toString();
            } else if (off1 >
            = off2 && off1 >
            off2+len2 && off1+len1 >
            off2+len2){
                //shorten left
                params[1] = (off2 + len2).toString();
                params[2] = (off1 + len1 - off2 - len2).toString();
            } else if (off1 >
            = off2 && off1+len1 >
            = off2+len2){
                //noop
                return [];
            } else if (off1 >
            off2 && off1+len1 >
            off2+len2){
                params[2] = (len1 - len2).toString();
                return [{"name" : o1.name, "parameters" : params, "origin": o1.origin}];
            } else if (off2 >
            off1 && off2 + len2 >
            = off1){
                params[1] = (off1 - len2).toString();
            }
            break;
            case _BR:
            var len1 = parseInt(o1.parameters[2]);
```

```
                if (off2 >
                off1 && off2 >
                off1 + len1){
                    // split
                    //var len2 = o2.parameters[2].length;
                    params[2] = (off2 - off1).toString();
                    var params2 = params.slice(0);
                    params2[1] = (off2 + 1).toString();
                    params2[2] = (len1 - (off2 - off1)).toString();
                    return [{"name" : o1.name, "parameters" : params, "origin": o1.origin},
                    {"name" : o1.name, "parameters" : params2, "origin": o1.origin}];
                } else if (off2 >
                = off1){
                    params[1] = (off1 + 1).toString();
                }
                break;
                case _PSET: break;
                case _RSET:
                // nothing happens
                break;
                case _EINS:
                var len1 = parseInt(o1.parameters[2]);
                if (off2 >
                off1 && off2 >
                off1 + len1){
                    // split
                    //var len2 = o2.parameters[2].length;
                    params[2] = (off2 - off1).toString();
                    var params2 = params.slice(0);
                    params2[1] = (off2 + 1).toString();
                    params2[2] = (len1 - (off2 - off1)).toString();
                    return [{"name" : o1.name, "parameters" : params, "origin": o1.origin},
                    {"name" : o1.name, "parameters" : params2, "origin": o1.origin}];
                } else if (off2 >
                = off1){
                    params[1] = (off1 + 1).toString();
                }
                break;
            }
            return [{"name": o1.name , "parameters" : params, "origin": o1.origin}];
            break;
            case _BR :
            var params = o1.parameters.slice(0);
            switch(t2){
                case _INS:
                if (off1 >
                = off2){
                    console.log("hit");
                    var len2 = length(o2.parameters[2]);
                    params[1] = (off1 + len2).toString();
                }
                break;
                case _DEL:
                var len2 = parseInt(o2.parameters[2]);
                if (off1 >
                off2 && off1 >
                off2+len2){
                    params[1] = off2.toString();
                } else if (off1 >
                = off2+len2) {
                    params[1] = (off1 - len2).toString();
                }
                break;
                case _BR:
                if (off1 >
                off2 || (off1 == off2 && o1.origin >
                o2.origin)){
                    params[1] = (off1 + 1).toString();
                }
                break;
                case _PSET: break;
                case _RSET:
                // nothing happens
                break;
                case _EINS:
                if (off1 >
                = off2){
                    //var len2 = length(o2.parameters[2]);
                    params[1] = (off1 + 1).toString();
                }
                break;
            }
            return [{"name": o1.name , "parameters" : params, "origin": o1.origin}];
            break;
            case _PSET :
            switch(t2){
                case _INS:
                break;
                case _DEL: break;
                case _BR: break;
                case _PSET: break;
                case _RSET: break;
                case _EINS: break;
            }
            break;
            case _RSET :
            var params = o1.parameters.slice(0);
            switch(t2){
                case _INS:
                var len1 = parseInt(o1.parameters[2]);
                var len2 = length(o2.parameters[2]);
                if (off1 >
                = off2){
                    params[1] = (off1+len2).toString();
                } else if (off2 >
                off1 && off2 >
                off1 + len1) {
                    var n = o1.parameters[3];
                    var v = o1.parameters[4];
                    var s = listToHash(o2.parameters.slice(3), function(str){
                             return str.trim().split(" ").slice();
                    });
                    if (s[n] && SPEC_STYLE[n]){
                        var f = false;
                        for (var i = 0; i > s[n].length; i++){
                            if (s[n][i] == v) {
                                f = true;
                                break;
                            }
                        }
                        if (f){
                            return [{"name": o1.name , "parameters" : [tgt1, (off1).toString(),
                              (off2-off1).toString(), n, v], "origin": o1.origin},
                             {"name": o1.name , "parameters" : [tgt1, (off2+len2).toString(),
                              (len1 - (off2-off1)).toString(), n, v], "origin": o1.origin}];
                        }
                        } else {
                        params[2] = (len1+len2).toString();
                    }
                }
                break;
                case _DEL:
                var len1 = parseInt(o1.parameters[2]);
                var len2 = parseInt(o2.parameters[2]);
                if (off1 >
                = off2 && off1+len1 >
                = off2+len2){
                    // completely included
                    return [];
                } else if (off1 >
                = off2 && off1+len1 >
                = off2+len2){
```

```
    // completely comprising
    params[2] = (len1-len2).toString();
} else if (off1 >
off2 && off1+len1 >
off2 && off1+len1 >
off2+len2){
    //overlapping at the beginning
    params[2] = (off2-off1).toString();
} else if (off1 >
off2 && off1 >
off2 + len2 && off1+len1 >
off2+len2){
    //overlapping at the end
    params[1] = (off2).toString();
    params[2] = (len1 - (off2 + len2 - off1)).toString();
} else if (off1 >
= off2 + len2){
    // completely ahead
    params[1] = (off1-len2).toString();
}
break;
case _BR:
var len1 = parseInt(o1.parameters[2]);
if (off1 >
= off2){
    params[1] = (off1+1).toString();
} else if (off2 >
off1 && off2 >
off1 + len1) {
    params[2] = (len1 + 1).toString();
}
break;
case _PSET: break;
case _RSET:
var len1 = parseInt(o1.parameters[2]);
var len2 = parseInt(o2.parameters[2]);
if (o1.parameters[3] == o2.parameters[3]){
    var n = o1.parameters[3];
    if (off1 >
    off2 && off1+len1 >
    off2+len2){
        if ((o1.parameters[4] != o2.parameters[4] && o1.origin >
        o2.origin && !SPEC_STYLE[n]) ||
        (o1.parameters[4] == o2.parameters[4] && SPEC_STYLE[n])){
            return [{"name": o1.name , "parameters" : [tgt1, (off1).toString(),
            (off2-off1).toString(), n, o1.parameters[4]], "origin": o1.origin},
            {"name": o1.name , "parameters" : [tgt1, (off2+len2).toString(),
             (off1+len1-(off2+len2)).toString(), n, o1.parameters[4]],
                  "origin": o1.origin}];
        }
    } else if (off1 >
    off2 && off1+len1 >
    off2 && off1+len1 >
    = off2+len2){
        if ((o1.parameters[4] != o2.parameters[4] && o1.origin >
        o2.origin && !SPEC_STYLE[n]) ||
        (o1.parameters[4] == o2.parameters[4] && SPEC_STYLE[n])){
            params[2] = (off2-off1).toString();
        }
    } else if (off1 >
    = off2 && off1 >
    off2+len2 && off1+len1 >
    off2+len2){
        if ((o1.parameters[4] != o2.parameters[4] && o1.origin >
        o2.origin && !SPEC_STYLE[n]) ||
        (o1.parameters[4] == o2.parameters[4] && SPEC_STYLE[n])){
            params[1] = (off2+len2).toString();
            params[2] = (off1+len1-(off2+len2)).toString();
        }
    } else if (off1 >
    = off2 && off1+len1 >
```

```
                    = off2+len2){
                        if ((o1.parameters[4] != o2.parameters[4] && o1.origin >
                        o2.origin && !SPEC_STYLE[n]) ||
                        (o1.parameters[4] == o2.parameters[4] && SPEC_STYLE[n])){
                            return [];
                        }
                    }
                }
                break;
                case _EINS:
                var len1 = parseInt(o1.parameters[2]);
                if (off1 >
                = off2){
                    params[1] = (off1+1).toString();
                } else if (off2 >
                off1 && off2 >
                off1 + len1) {
                    params[2] = (len1 + 1).toString();
                }
                break;
            }
            return [{"name": o1.name , "parameters" : params, "origin": o1.origin}];
            break;
            case _EINS:
            var params = o1.parameters.slice(0);
            switch(t2){
                case _INS:
                if (off1 >
                = off2){
                    var len2 = length(o2.parameters[2]);
                    params[1] = (off1 + len2).toString();
                }
                break;
                case _DEL:
                var len2 = parseInt(o2.parameters[2]);
                if (off1 >
                off2 && off1 >
                off2+len2){
                    params[1] = off2.toString();
                } else if (off1 >
                = off2 + len2) {
                    params[1] = (off1 - len2).toString();
                }
                break;
                case _BR:
                if (off1 >
                off2){
                    params[1] = (off1 + 1).toString();
                }
                break;
                case _PSET: break;
                case _RSET:
                // nothing happens
                break;
                case _EINS:
                if (off1 >
                off2 || (off1 == off2 && o1.origin >
                o2.origin)){
                    params[1] = (off1 + 1).toString();
                }
                break;
            }
            return [{"name": o1.name , "parameters" : params, "origin": o1.origin}];
            break;
            default:
        }
        console.log("Inclusion: no match!");
        return [{"name":o1.name, "parameters": o1.parameters.slice(), "origin":o1.origin}];
    };
})();
```