Ein neues Modell der Internetbenutzung

Master Thesis

Author(s):
Lichtenauer, Matthias Scheller

Publication date:
2008

Permanent link:
https://doi.org/10.3929/ethz-a-005589713

Eine neues Modell der Internetbenutzung
A novel way to use the net

Matthias Scheller Lichtenauer

March 15, 2008

Abstract

The architecture of the world wide web is optimized for the use of passive resources with relatively low or fix update rates and high tolerance in timing.
If we want to monitor or process output of a remote process, sensor or device, the requirements for communication differ. Events may happen frequently, but not at a fixed rate. Furthermore, news of sporadic events may be needed within a resource specific time period. We propose a framework to use existing web infrastructure for this purpose.

Contents

1 Introduction

PUSH vs POLL

This work is about two different paradigms of communicating news. The POLL paradigm lets the receiver ask for news. The PUSH paradigm lets the sender bring the news. The existing system we analyze and possibly extend and improve is the world wide web, the information system build upon linked documents. These documents are requested and transferred with the help of the Hypertext Transfer Protocoll (HTTP). Communication in the world wide web is mostly based on the POLL paradigm.
There has been a technology called Pointcast based on the PUSH paradigm, designed to broadcast news. Its effects on bandwidth use were comparable to those we see today with peer to peer networks, namely to use a lot of bandwidth. We do not intend to say the Pointcast technology was inefficient, but make a purely quantitative statement. Pointcast was banned from corporate networks and the PUSH paradigm was pushed off the web with it, altough some network browsers still support a generic replacement mechanism (multipart/x-mixed-replace), for instance used by web cams. As a bottomline of this story, one of the design goals of a proposed web technology has to be defineable invasiveness of a solution with regard to bandwidth use.

The communication in the web is highly asymetric. Since links in HTML documents can be interpreted as edges in a directed graph, the number of edges from or to a document node can be counted as in-degree respectively out-degree.

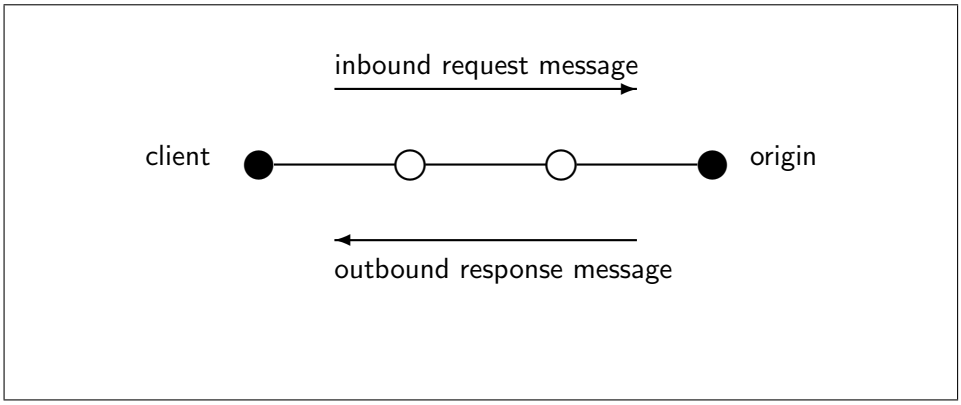

Figure 1: The world model of the Hypertext Transfer Protocol, the main transfer protocol in the world wide web, is a chain of stations called hosts. Request messages are sent from the client host towards the origin (inbound), response messages towards the client host. All other hosts are called intermediaries.

Altough this does not provide data about how many times a link is effectively accessed, we accept it as an indicator, perfectly aware of a possible systematic error. The distribution of these degrees is highly asymetric. This simply follows from the observation, that in samples, for instance by the Laboratory of Web technology of The University of Milano, distributions of in and out degrees are skewed. If we assume, that access patterns follow link patterns, we can postulate, that caching might be a good strategy to enhance performance, since the gain in efficiency of caching scales with the number of reuses of an item. But what, if the distribution of in- and out degrees will become less skewed, or if the internet should be used to transport data of billions of sensors, for instance mobile handsets?

The world model as depicted in Figure 1 does not show the fact that some hosts can not initiate connections to other hosts, due to firewalls. Here also, the web is highly asymmetric.

1.1 The net to scheme

We will first discuss examples of applications whose value depends on communicating news, emphasizing the economical aspects of the ideas presented afterwards. We will then have a look at communication application paradigms, focussing on two emerging technologies that address some of the problems identified. From the ideas contained in these technologies, we try to abstract and generalize in order to find a generic model. We then propose an implementation of this abstraction, and discuss implications.

Let us sketch two application designs in communication networks. Both designs have to deal with the cost of communication as a major limiting factor of their practical usefulness. The price of communication is, in today's market determined by the volume of data transmitted or by the time a predefined bandwidth is provided. There are no explicit availability guarantees for communication in the consumer market. We emphasize this in order to strain the need for an explicit detection of a broken link, or, more general, risk of failure as a cost factor. The energy spent for initiation, transporting data and keeping a connection alive could also be limiting factors, but are not relevant in the scenarios we consider here.

Monitoring is a repetitive task

To the first example: A small transport company ordered a software system allowing its customers to enter their requests in a form and transmit it to the system. Notification of received requests should be sent to a manager by two independent notification channels. Customers of this company accepted the implementation, and most of the time, it worked well and costs were in good relation to the effects, mainly to relieve the manager from standard tasks. The time gained could be spent for exceptional tasks that necessited human intervention, namely calling customers.
But one day per year, the company was in trouble, because the notifications arrived late or were lost in the electronic mail system. While the latter case could be detected by message numbering, there was no other economically feasible way to detect system failures than to regularly check the web interface listing the orders. The manager got preoccupied whenever there was no customer request for half an hour and started checking the web interface ever so often - even with his cell phone.

Use of Keepalive requires timing

The second example is an application for itinerary tracking presented by Christian S. Jensen from Aalborg University, Danmark in a colloquium at ETH in April 2007. The amount of data transferred could be massively reduced by predicting the position and transferring only deviations from the prediction. The fact, that the car is on track is news in such a scenario, but needs far less data to be transferred. On the other hand, at least a rough synchronization is needed.

Altough it is technically feasible to have synchronous, reliable links with bandwidth guarantees in a fixed line network, the cost of such guarantees in terms of money, energy and radiation is too high for many applications and mobile networks. While it is affordable for a transporting company with lorries, the profit an individual could have from sharing for instance its position does not yet pay the price. This might change in the near future, generating a completely different, not generically cacheable data flow pattern in the internet. We keep this in mind, when we now look for solutions that are better for isolated end to end connections.

1.2 Problem statement

The two scenarios presented have in common, that there is not primarily a state to be transferred, but *changes* of this state, meaning also, absence of change is news. From human factor designs we know it to be one of the hardest tasks for human minds to wait a long time for an event and reacting quickly when it eventually happens. If we can make a technology reliable enough to fulfill this task at an acceptable error rate, we claim it will be usefull. In the example of the transport company, it would have been enough to have a reliable notification mechanism, transferring only the fact that a predefined event has happened. We will call this *beeping*.

Absence of change may be news

The problem in the latter case lies in synchronization of two remote processes within some tolerance limits with a minimal number of messages. Both parties have to be able to perform the same computation, agree on minimal notification period and precision and the time the notification itself travels may have an impact on the update rate needed - not to forget the time it takes for computation, either.

1.2.1 A first abstraction

As a first abstraction, we can see this problem as a minimization problem of a cost function, whose arguments are cost of link, cost of data transfer and cost of failure. It may, under many circumstances, not be possible to estimate these function instances empirically, but some general observations can be made at this level of abstraction:

Where PUSH does beat POLL

- If the overhead per message is constant while sampling some sources at a constant rate, producing an almost constant amount of data, the overhead percentage increases with increasing update rate. In other words, if TCP/IP headers are 80 Bytes, it is better bandwidth use to transfer 1024 Bytes payload in a package than transferring 512 Bytes.
- The overhead of a POLL model is higher than the overhead of a PUSH model, if the same request is repeated over and over again, since we could ask the question once and get consecutive answers.
- If a small percentage of data changes, differential or incremental data can considerably reduce the volume of data transferred. If changes are limited in size, we can estimate a lower (sic!) bound for the achievable latency or the bandwidth guarantees needed. We might be interested in upper bounds, but lower bounds will provide us with means to detect and reject unsatisfiable constraints.

Before further developing the abstraction, we need to study existing models and implementation technologies for communicating applications.

2 The States of the Art

The client has to find and bind

We will discuss in this section a few approaches taken today in distributed environments with client-server interaction. The client is the machine *initiating* a transfer or change of data. We strain this definition, because it requires that the client must be able to find the server and know how to initiate an interaction (bind).

2.1 Interaction styles and optimization approaches

The styles of interaction in distributed applications vary. We have studied the following approaches:

- Terminal: This approach leaves data processing on the server and just transfers a user interface - in the case of VNC only bitmaps, in Telnet only character data. All processing load is on the server side, interaction just ships commands to the server and data streams in both directions. The major email transfer protocols are in their core terminal applications. Altough the terminal approach is clearly an online technology, email is the best known example of asynchronous, distributed transport on application layer.

- Remote Procedure Call (RPC): Executable code at different remote locations can interact with the help of a proxy object available in source code. The remote effect of the code shall be transparent to the programmer. The proxy object is tightly coupled to the remote code, altough compilations from source code to executable code can run separately at different locations. This results either in manually maintaining parallel versions or a complex system (middleware) handling this coupling.

- Mobile Code: this approach, for instance realized in JAVA or .NET products, ships precompiled code to be interpreted to the remote location, thus reducing the need to transform data in a generic representation and easing version handling. Aside of the possible invasiveness of the shipped code, this approach also requires a predefined and preinstalled execution environment and a mechanism to discover and transfer the code. The interaction model is RPC, only deployment and execution model changes. Altough Zonnon [6] would also fall in this category with regard to deployment, it supports message-based interaction style.

An in-depth discussion of these approaches with regard to network management applications can be found in [2]. This work is particularly interesting, since in network management protocols, data flow patterns appear, that might be similar to sensor data flows.

There are also some ideas how to reduce latency and volume of data transfers beyond the field of optimal encoding:

- Background prefetch: Microsoft introduced the asynchronous background transfer in a scripting language for use in web browsers. This technique was used to reduce *perceived* latency. The mail data stays on a central machine in such systems and is copied in the background to the user's machine. This does not reduce network traffic, but hides it from the user's attention.

- Caching: caching effectively avoids network traffic by storing and reusing answers to previous requests. Effectiveness depends on the number of reuses. The price to pay is complexity in order to maintain freshness and avoid abuse by overwriting cached data.

- Modularization: The Compound Document activity of W3C aims to achieve seamless integration of XML-based markup in one document or a tree of several documents. This would allow to have independent operations on two subtrees and separate type-scopes through namespaces. Reuse shared data becomes possible, but the prize is the same as for caching.

2.1.1 Refinement of the problem statement

As a first summary we can name some dimensions of a network application protocol design, that influence performance characteristics

1. interaction model (synchronous, blocking vs. asynchronous)

2. buffering model (tradeoff between overhead and latency)

3. communication model (command shipping vs. data shipping)

4. data model (modularization, ability to transfer differential or incremental data)

5. topological model (central hub vs. point to point)

We will now have a closer look at two emerging technologies, AJAX and XForms, under these aspects.

2.2 AJAX

Asynchronous Javascript and XML (AJAX) uses mobile code and background prefetch, thus reducing number of transfers and transfer bandwidth needed and allowing a more responsive user experience. But it is in its core still a POLL model.

AJAX polls

```
(...)
// Mozilla, Opera, Safari sowie Internet Explorer 7
if (typeof XMLHttpRequest != 'undefined') {
 xmlHttp = new XMLHttpRequest();
}
if (!xmlHttp) {
// Internet Explorer vor Version 7
try {
 xmlHttp = new ActiveXObject("Msxml2.XMLHTTP");
} catch(e) {
 try {
 xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
 } catch(e) {
 xmlHttp = null;
 }
}

if (xmlHttp) {
 xmlHttp.open('GET', 'http://www.example.com/path/to/resource',
true);
```

Figure 2: AJAX is a catch-all term for a bunch of XML technology and JavaScript. There are several implementations of the XMLHttpRequest object needed on client side. This and the synchronization and error handling needed can be wrapped by frameworks. Some authors consider this as a middleware revival.

Javascript can interpret data as code. In order to avoid abuse, the same origin policy is implemented in browsers. Same origin means, that asynchronous requests can only target resources from the origin of the document the requesting code is contained in. This security model has implications on what data may be accessed from a local source. If we strain it to its limits, even document type definitions like the definition of HTML could not be shared.

2.2.1 Summary on AJAX

To resume, AJAX enhances functionality of web applications by differential update techniques, asynchronous background transfer, and the embedding of mobile code with access to the data on client side. The fact that most interpreting engines are single threaded does not avoid race conditions if asynchronous requests are used. The absence of statically typed systems and a clear concurrency model in the language itself make many consider this technology as harmful, even when harm is not intended. One approach to alleviate the typing problem could be to cross compile a statically typed language to javascript. This approach is taken by Oberon Script [4]. But this would still not help to

```
xmlHttp.onreadystatechange = function () {
if (xmlHttp.readyState == 4) {
try{
var elem = owner.getElementById('ticker')
var position = elem.getAttributeNS(null, 'x1');
}catch(e){ return;}
elem.setAttributeNS(null, 'x1', position * 1 + 1);
elem.setAttributeNS(null, 'x2', position * 1 + 1);
elem.setAttributeNS(null,'y2',xmlHttp.responseText*1 );
}

};
```

Figure 3: JavaScript code is executed locally on the client side. The JavaScript runtime allows registration of event handlers. The event handler registered here will replace part of the document object model of the web browser. This allows to send partial updates to web pages already parsed and rendered in the browser. Since the user does not see this processing in the background, this creates the illusion of a local application. The code can also process user input. Sorting of a list or some input checking for instance can so be done without client-server interaction.

deal with the concurrency problems, which require a synchronizing framework on client side. If we want to build an application framework that handles AJAX functionality, namely to replace part of a remote model, in a generic, safe manner, we will therefore have to handle the type conversion problem (e.g. ticker becomes a 64bit integer on origin side), the id mismatch (the id ticker has been introduced to the model after the client has downloaded it) and the concurrency problem.

2.3 XForms

XForms polls

XForms is an application of XML. XForms separates models(instances in XForms parlance), actions and views within a document entity. Actions are triggered by events, either user interactions or changes in the model. Unlike the communication model of HTML forms, a request to the server does not have to replace the whole document, but may replace a model (or nothing at all). The data can be submitted in different ways, but especially as XML over a HTTP/1.1 transport (also SOAP). This makes XForms a potential client for web services technology.

```
<?xml version="1.0"?>
<?xml-stylesheet href="style.css" type="text/css"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html
xmlns="http://www.w3.org/1999/xhtml"
xmlns:xf="http://www.w3.org/2002/xforms"
xmlns:ev="http://www.w3.org/2001/xml-events">
<head>
 <title>eXample</title>
 <xf:model>
  <xf:instance id="in1">
   <data xmlns=""/>
  </xf:instance>
  <xf:instance id="in2">
   <data xmlns="">
    <q>4</q>
   </data>
  </xf:instance>
  <xf:submission
   action="http://www.example.com/random/int32
   method="post" id="sub1" replace="instance" instance="in2">
   <xf:send submission="sub1" ev:event="xforms-submit-done">
  </xf:submission>
 </xf:model>
</head>
<body>
 <xf:submit submission="sub1">
  <xf:label>start</xf:label>
 </xf:submit>
 <xf:output ref="instance('in2')/q" ></xf:output>
</body>
</html>
```

Figure 4: Example HTML document embedding an XForm. This code presents a button to the user. Clicking this button with the mouse starts an event driven, endless loop, replacing only part of the document by the response XML fragment. The output element takes its data by reference from the model.

2.3.1 Summary on XForms

The XForms approach exchanges XML data and uses a declarative markup language to specify an event-based processing model of these data. Within this markup, models, views and controller are clearly separated. The labels of a form may be taken from a model. Since models can be loaded as separate documents, this allows some modularization of data. A user may load french or chinese labels without having to reload the entire document.
We consider these features as essential from our perspective:

- The data transmitted can be described as an XML document. This document can be loaded from a server or be stored to disk.
- The data used by some elements of XForms can be loaded from a source outside of the document the element is contained in. If this data can be reused, quite some transfer can be avoided.
- The data in a model instance can be referenced in an input or output element. This can be used to allow the user to specify the target URI of a form submission *at runtime*, if the security model allows it. Submissions can also be directed to files and replayed later this way.
- Actions can be triggered by events in the Document Object Model underlying the processing of documents. Some errors are also events.
- As in spreadsheets, references can be bound to elements of the model (e.g. summing up values of some nodes representing numbers). Adding a reference can cause a cyclic reference error. Therefore, the whole model has to be checked again (topological sort, rebuild). If we change the value of a leaf node, only values of nodes referencing it have to be updated (recalculate).
- XForms is designed to be embedded in a host document. Since it is an XML application, its model contains the notion of document, especially uniqueness of identifiers within a document.

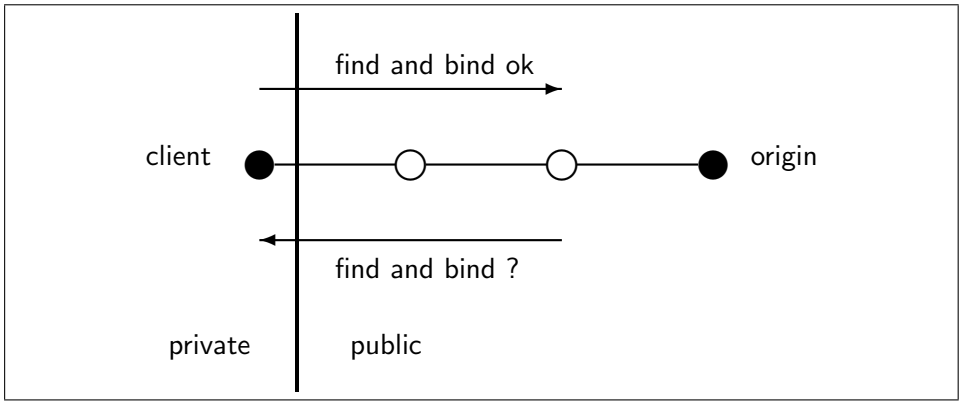

Figure 5: The resources targeted by HTTP requests have to be on hosts with public network addresses and must have a public, unique symbolic name that can be used to send requests to them. The web uses symbolic URI like *http://www.example.com/index.html* for this purpose. In order to scale, the translation from a symbolic URI to a set of network addresses should be relatively stable, else caching will loose effectivity.

2.4 About the REST

In order to further discuss the limitations of these approaches with regard to the goal of this work, namely to propose a framework for use of web technologies beyond publishing media, we will now have a look at its transport mechanism and architectural design. The Hypertext Transfer Protocol (HTTP) used as major transport mechanism in the web uses the POLL paradigm. The architectural properties of HTTP as well as the reasoning for them are discussed in [1], coining also the term of Representational State Transfer (REST) for these properties.

HTTP polls

REST is an architecture for hypermedia systems (say the world wide web). REST claims to achieve scalability, robustness and independence of components by the use of generic, uniform resource identifiers (URI), a simple client request - server response model of interaction and a representation of interaction such that no server has to maintain a state across several requests. The response data to a HTTP request has to contain all the information on how to proceed (e.g. links in HTML). RESTful applications communicate via messages over an ordered transport. The messages appear in request-response pairs and are passed along a chain from client to origin and back. The message types in a protocol conforming to REST have generic and request specific parts in order to allow intermediate stations in the communication chain to operate on the messages by just reading the headers. For instance a caching host in the chain should be able to read just a few lines of a message before deciding, if the message should be cached to profit from reuse.

If requests do not have to be routed to a particular, unique resource, replication of data and parallel distribution of server instances are easier to achieve. Once a request is handled, the server can clean up used resources, since all state is assumed to be on the client side.

A description of possible transitions is ideally sent to the client with each response. REST Applications are virtual state machines. The problem is, that the new state is only defined in the case, where the client receives the response. The problem of error recovery is left to the application programmer by HTTP.

While idempotent requests like HTTP GET may simply be repeated if the underlying connection breaks, we have to be more careful with mutation requests like POST. POST requests are designed to communicate the wish to change the state of a resource. If the request is for instance an order, the application should be able to detect and handle duplicates, since the protocol does not.

REST implies polling resources from client side, since the state and possible transitions reside on client side. In case the client *needs* to interact with a particular, dynamic source of events like a process, sensor, a network device or a queue, polling is under certain circumstances not as efficient in using bandwidth and keeping latency low as a push architecture.

3 Design

In the last section, we presented some technologies used in the web and a general model of the web. We have studied, how these technologies fulfill the following tasks:

- identify an entity
- transfer the request for an action with regard to this entity
- restrict effects of actions

We will now propose our design and present the reasoning for our design desicions.

3.1 Identifying and entities

It may sound academic, but let us recall, that identity consists of id, meaning self and entity, meaning a whole. In other words, to identify something means, we can test if something belongs to a whole and we can distinguish it from all other things within this whole.
Identification of entities is implemented in the web with cascaded symbolic names. The symbol www.example.com will be resolved by asking an authority in the scope of .com where to find the authority in the scope of example.com and so on. Length of identifiers can vary, but there is an upper bound of 255 Bytes for the symbol as a whole.
Within a host, the same cascaded steps are repeated within the file system. From a more abstract point of view, when cascading scopes, we should request, that any identifier is unique within its scope, and the scope of each identifier should be unambiguously identifiable. Unambiguity of scope is not a trivial requirement. JavaScript for instance, will not even warn, if a local variable has the same name as a global and simply use the global.
From a processing point of view, identity or uniqueness checking is a blocking operation, since a candidate identifier has to be compared to all identifiers in a scope before a statement is possible. If we can force an ordering of identifiers, it will still be a search operation in $O(\log|N|)$ for introducing identifiers or updating associated values. If we model references, further calculations in $O(|N| + |E|)$ may be needed to check absence of cycles. Removing identifiers runs in $O(\log|N| + |E|)$ with references. We will keep this in mind when designing the data structures, in order to make differential updates possible. As we have worked out in the sections before, it is possible to gain effectivity by transferring only changes in state, if the changes affect only part of the data. This has a heavy impact on how we may model our data and possible parallelism.

3.1.1 Compiler checked scoping rules

When we write code and data structures for local systems, the explicit identifiers are usually translated by a compiler process to addresses and offsets. The compiler can do that because, at one point in this process, uniqueness of an identifier within its scope is checked, so that all uses of an identifier will later, at runtime, resolve to the same memory location. Since this is a set difference operation, all the identifiers within a scope must have been read at this point. The transfer of identity from a symbolic name to address and offsets, for instance on a stack, is implicitly contained in the executable code. If we pass the same source code through a compiler on different systems, addresses and offsets for the same source code can completely differ due to alignment or differences in an outer scope, for instance a base class in object oriented systems. Even if we manage to get a generic representation of data that may be passed from one platform to an other, there has to be a mechanism to map this representation again to a memory location. This can be achieved by compiling such mechanisms along with the data representation (set and get procedures). Then, the problem boils down to ensure, that all parties involved have used the same version of the source code. This can still lead to some tricky problems with concurrent versions, compatibility of data between versions, modularity and distribution of the source code, leading to complex, highly interdependent systems. [2] identified this as 'My-Middleware-Is-Better-Than-Yours'-syndrome.

3.1.2 The intermediary representation approach

Another approach to the identity problem is to make the transferred representation so generic and yet expressive that generic transfer procedures can be used. It is the usual hub and spoke approach, if we wish to have transformations from one set of formats to any other format in this set. By having a hub representation, we only need O(n) transformation procedures instead of O(n^2). But the hub format must be able to represent everything contained in any spoke format. XML for instance contains the notion of document (scope), identity (ID attribute), type and reference (`IDREF`attribute). But the combination of type and reference (POINTER TO LONGINT) is not a basic concept of XML, but is added with XML schema. Since identity has to be expressed explicitly in XML, we can not send two elements with the same identity in one document in order to express two consecutive values of the same thing.

3.1.3 Reference and containment

In computer science we distinguish primitive things from things that can contain other things on the one hand, and values from references on the other hand.

While the containing translates to the mathematical notions of sets and elements, it is not clear, how value and reference shall be modeled. If we model reference as functions, we lack the notion of time and concurrency. If we model it as channels, the model will become more complex and many statements can be undecidable.

The distinction of value and reference is usually made when passing arguments to subroutine calls. Call by value copies the data, while call by reference copies a reference to the data. While a reference may be valid locally, it can not be accessed from a remote system. We also have to differentiate in words of dependency graphs, leafs from inner nodes. If we argue from the point of view of dependency graphs, we also cover things like formulas in spreadsheets as reference types. Replacing a leaf may result in a division by zero, but not in accessing a void reference.

3.1.4 Design decisions

We decided to strip down the data model to the simplest we could think of, namely a container with flat key value pairs. We will be explicit about identifiers and versions. We will use Lamport clocks to denote versions.

Our first design decision is to be explicit about scope identity and symbol identity:
Our general model is a set of identifiers with associated values. The set contains a special identifier denoting the identity of the model as a whole. The set contains a special identifier denoting the version of the model.
The set contains a special identifier denoting the version of the values.

Our second and most restrictive design decision is to ban reference types from the model data. This does not mean, that data transferred within the model may not be referenced in a view, but the model itself does not contain references. In other words, we operate on leaves. It is a voluntary restriction in order to gain possible parallelism.

We will use ordered streams of key value pairs in order to transfer changes.

3.2 Transferring Requests

The initial design of HTTP modeled file operations load, store, append and delete on textual data. But web clients soon became general viewers, also in the sense of the model, view, controller paradigm. But HTTP lacks a very important part of this paradigm, namely the notification mechanism from model to view.

As [1] points out, one of the properties of the HTTP protocol is genericity of methods. Intermediate hosts in the request chain do not have to understand the contents of the entity contained to decide about caching. We would like

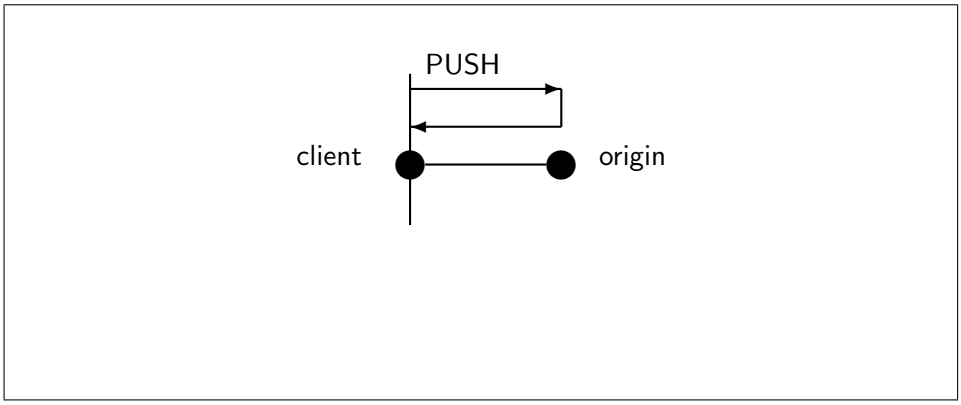

Figure 6: A direct PUSH request without any intermediaries can even be issued from a drain to a source on the same host in order to decouple two dynamic resources. Altough it is an overhead compared to a platform dependent view, the same mechanisms can be used on localhost or remote host. Instead of hiding the possible remote location as does RPC, we propose to explicitly state the origin of resources. This indirection will provide the runtime system with the possibility to physically map or remap a resource. This idea is also contained in Zonnon [6]. We'd like to thank Roman Mitin here for a lot of interesting discussions on this subject.

to push this idea a bit further. Since a lot of hosts at the edge of the internet do not have a stable address, there is no entry in the Domain Name System for them. The GSM phone system handles this by giving the mobile station a stable home address (what users know as their phone number) and a temporary phone number in the network of the provider whose antenna is sending to the mobile station.

Inspired by the GSM system, we suggest a public proxy station, that allocates a buffer, where remote stations can send data to[1]. Humanity managed to have globally interoperable solutions for the mail and phone system. In these systems, the sender pays the price for delivery to a standard location in public space. We therefore propose a model very similar to a public email box, with some features email boxes do not have:

- The address can be dynamically created and be valid for a single message or a time range.
- If the message is a stream of news, we can make sure that the box detects if there is no news for a certain time.

[1]There is a proposal called Internet Indirection Infrastructure (i3) pushing this even further, proposing a globally identifiable set of proxy stations providing a tuple space where data can be inserted, routed and multiplexed. While the mathematical properties of i3 are simple and elegant, we do not yet see how it shall socially work. How long shall who keep a piece of data in public space?

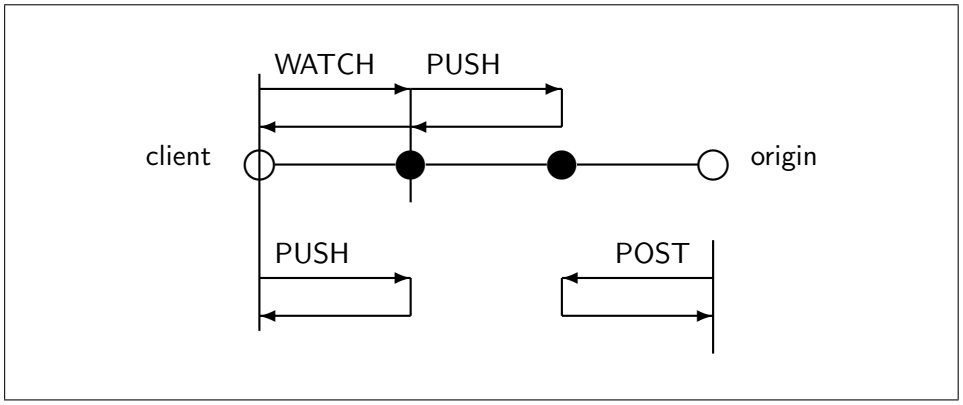

Figure 7: In order to decouple sending from receiving, we propose allocating buffers on hosts in the public, stable part of the network. The buffers shall be paid for by the party that requests the data to be filled in the buffer. PUSH request should always originate from the host containing the target buffer. We picture, that also the physical origin of the data could have a proxy in public. This is known and established technique. A new contribution could be to dynamically allocate and publish such proxies, but this is not subject of this work.

- Authentication of the sender is possible within the protocol.
- The sender can know the message has been delivered to the buffer.
- The sender does not know the message has reached its final destination without the receiver consenting.

Aside of the timing aspect, nothing changes for the business models of providers. What can be calculated and sold is network bandwidth and memory space. The social contract between providers and users does not change.

3.2.1 Design decisions

We provide a generic method for allocating a buffer for data from a source. The method for request to proxy shall be:

```
WATCH www.example.com/timeslicer
```

The URI in this request line refers to the source, not the buffer.

We provide a generic method to request a source to send its data or keepalives to a buffer. The generic method for request to origin shall be:

```
PUSH www.example.com/timeslicer
```

The URI in this request identifies the source. The URI of the buffer does not need to be known by the source, since it will use the connection on which it received the PUSH request. If the connection breaks, the server drops the request.

3.3 Restrict effects of Actions

3.3.1 Communication Model

If we have to process data residing at two different places, for instance a JOIN in a distributed relational database (All accounts of one global customer), we can either ship the data or ship the query. On local systems, the same architectural problem translates in transferring control (threading models) or transferring data (queueing model). There is no generally better decision here. The threading and queueing models are equivalent in their computational expressiveness , as Lauer and Needham pointed out[7]. Since web applications receive their input naturally in a queue (the link is assumed to be serial and ordered in HTTP), one might be tempted to opt for the queuing model to process data on the host. Furthermore, HTTP request handling follows a generic, phased pattern:

```
WHILE online DO
  Receive input
  Check headers and authentication
  Syntactically check input
  Semantically check input
  Perform operation
  Format output
  Send headers
  Send output
END;
```

At least the first two phases can be handled very generically, in case of simple file access even the complete process. This has led to a scientific debate about how to design the server application, threaded or queued. An overview of these papers can be found in the sources cited by [5].
In 2000, a first implementation of a staged, event driven server architecture (SEDA) was proposed [3]. It claims to scale well especially since waiting for a file from disk could be separated in a phase, so that handlers spent less time changing to waiting state. At this time, it was considered an acceptable design to fork a process, meaning creating a separate virtual address space for each new request. We also remark that resources are passive in this model. It has been argued that the overhead of thread switching can be reduced by using user level threads running within the same virtual address space. In threaded server architectures, a thread usually serves a connection as long as it is kept alive.

Since we want to design a system where observers should not influence each others, we opted for the threaded model.

3.3.2 Interaction Model

We have mentionned in the introduction two use cases, both depending on live connections. We also gave thought to a model where the origin could disconnect after receiving and acknowledging a request, perform an operation and later connect to a buffer allocated by the client and deliver the result. We abandonned the idea since it would imply that the buffer *must* be on a public host. Yet, we could argue that mail servers allow exactly the described offline semantics on the origin side. The problem with mail protocols lies in their invasiveness. A mail address is either public or not. A malicious user can pass a mail adress to a third party and this third party can make the practical use of this address, let us say, cumbersome. There are lots of authorization schemes out there, but they mostly have the same principal problem: one may either do something or not, depending if one belongs to a predefined group. There are capability based authorization schemes. They are used to eliminate duplicate requests, that are used in HTTP. The origin sends a salt value along with a form. If the client submits then the form, the origin can check the salt value, invalidate it and send a new one with the response. These schema, as lightweight as it is, can not help us either, since it provides use only to the host initially sending the salt value, let us say the capability. If an origin server wants to know, if a request may alter a resource on a third host, we do not know any other solution, than to send a request to this host or ask a mutually trusted third party. Aside of the creation of a connection, adding delay and so on, we would also have to avoid request loops.

We decided to stay RESTful here and to limit effects of a request on the origin server to either the end of the response or the lifetime of the connection sending the request, whatever comes first.

3.3.3 RESTful observer

Fielding [1] claims that if HTTP was successful, it was also due to the fact that servers could simply free all resources once a request was handled.

Our problem was to implement an observer pattern without introducing a stateful server, meaning a registration mechanism. If the observer can go away any time, we will need a mechanism on the model side to detect that and clean up resources. If an observer walks away, it should not influence any other observers. Some implementations of a model, view, controller pattern, for instance in the Eiffel language, build event notification mechanisms by inserting function pointers or delegate procedures in a container. A thread (event-handler) is going to execute all this code in case the event happens. In such a design,

a delegate can influence other delegates without control of the programmer of the model code. This pattern is not applicable in the HTTP world, since in HTTP the action has to be invoked by the observer, which is the client. We gave thought to a one-time-observer pattern, but discarded it, since it would still imply a polling model, even adding the overhead of timed-out requests to it.
We opted for one thread per observing connection to achieve independence of observers and statelessness. Registration happens by calling a method of the model object. The call is blocking until the event represented by this method has happened. This design allows parallel execution of event handlers, if they are readers and dont need to care about consistency, but also mutual exclusion if needed. If we know the maximal size of the model at compile time, we can also force observers to provide a buffer of appropriate size. With this design, even real time constraints could be met. We could calculate, how many time is needed to copy the buffer and limit the number of observers accordingly. Since, in our proposed design, observers handle control to the model monitor code, the programmer of the monitor code can enforce such policies. We also wanted to introduce per request timing parameter, allowing either a sampling semantic or a notification semantic, eventually a combination of both. From SEDA stems the idea of explicitly limiting the maximal number of threads serving requests in order to ensure an acceptable performance. this means that serving accepted connections has higher priority than accepting new connections.

The intended semantics of a generic observer on the sending side is in prose: send data as long as there is, flush the buffer if a predefined amount of data has been buffered, but at least at a fixed rate.

On the receiving end, the intended semantics in prose is: receive news as long as there is and the news is transferred in time. Transfer time includes buffering. A full buffer can trigger a receive timeout.

So we have two timing parameters in our model. One is there to explicitly trigger notification in case of inactivity, the other to explicitly trigger action if no news has been transferred for a certain time.

We propose a third parameter to describe a lease for the client-allocated buffer in case the connection from client side to the buffer breaks.

We are aware of the fact, that TCP provides a keepalive mechanism, but it is optional and we need end-to-end semantics.

We ship the data conceptually in a stream of tokens. If we want to achieve a maximal transfer time per token (at least in the average case), we need to limit the tokens in size. Since this requirement is transitive, it makes sense to introduce the maximum chunk size as a protocol parameter. A minimal chunk size is given by the fact that each host using the internet protocol shuld be able to accept packets with 512 Bytes payload.

In a heterogenouos, public, packet switching network as is the internet today, this maximal transfer time will not be a parameter that can be enforced end to end. But in its negation, we can use it as an indicator, when a connection should

be considered broken. The sender may set the TCP idle timeout accordingly, the receiver can guard the blocking receiver procedure with a timer.

3.3.4 Design decisions

We restrict the effects of actions by explicit timeouts. This allows any participant in the communication to determine, when a link is broken. We limit data tokens in size, so as to limit the time a link is occupied. We keep observers separated by allocating an independent thread to each observer. Our data model is a monitored producer consumer system, giving consumer privileged access before producers. The sampling consumer can run in parallel to a producer, but may miss some values if update frequency exceeds sampling frequency. We can also make a consumer block, but this requires, that the consumer thread provides a buffer the size of the whole data model in the respective encoding to avoid deadlock.

4 Implementation

We present a reference implementation of the Simple Network Application Protocol (SNAP). We do not intent to do away with HTTP as an application level protocol, but to extend its capabilities in order to deal with dynamic resources. As we have worked out, a dynamically changing network transporting dynamic data from edge to core will not profit from caching effects to the same extent. The Simple Network Application Protocol (SNAP) extends HTTP with two new methods, named WATCH and PUSH as shown in Figure 7 on page 18.
The WATCH request is addressed to any host in the request chain, as in the command pattern. In the extreme case, the answering server may be on the local host or on the origin. WATCH requests the responding host to allocate a buffer, send a PUSH request to the URI of the WATCH request and to return a URI to this buffer with status code 201. Figure 8 shows a successful message sequence.

4.1 Active Oberon System as reference platform

Active Oberon is a language in the Pascal, Modula and Oberon family. We chose it for a reference implementation for two reasons:

- It implements the eggshell model of concurrency. Threads waiting for a change in an encapsulated data structure are favored to get access before mutators. The observer pattern can therefore be implemented by a thread that is blocked in a monitor, until change happens in this monitor. We just have to be careful and notify all observing threads if the monitor shall be removed from the running system.

- Threads are first class citizens in Active Oberon. Therefore, a datastructure with an own activity is modeled as an Active Object. For instance a buffer with one producer and potentially unlimited consumers can be modeled with an Active Object, implementing the producing code as its body. The problem of finalizing resources in a specific order, for instance a buffer and a resource, can be naturally mapped in the sequencing of the statements in such a body.

Active Oberon comes with a modular runtime system. The notion of dynamic, unique resource corresponds to the module concept. Nevertheless, we had to model the request handler as object, since modularity in the sense of loader units and modularity in the sense of extensibility are orthogonal in the language. It is not possible to express, that a Active Oberon Module has to implement a certain interface. But since we consider that observing a resource that should be responsive is done best with a layer of indirection, say a wrapper, we could accept to work around.

The client initiates creation of a buffer:

```
WATCH snap://www.example.org/AA55 SNAP/1.0
Host: www.example.org
Delay: 100
Rate: 10
Lease: 3600
Capacity: 6
Accept: application/snap
```

Host your.proxy.org in the request chain handles the WATCH request and sends itself an inbound message:

```
PUSH snap://www.example.org/AA55 SNAP/1.0
Host: www.example.org
Delay: 100
Rate: 10
Accept: application/snap
```

Host www.example.org receives the request and accepts it:

```
SNAP/1.0 200 OK
Transfer-Encoding: chunked;
```

(www.example.org or any server in the request chain to origin will start to send data using this connection) Host your.proxy.org notifies the client, that the buffer has been successfully created and the origin has accepted to send data to it:

```
SNAP/1.0 201 Created
Location: snap://your.proxy.org/FA53
```

Figure 8: A successful WATCH request

4.2 Implementation of generic Identifier

As long as a program runs on one host, we might be tempted to take a physical memory address as an identifier. The big advantage is the hardware support for identity comparison and equal length of identifiers. Whenever the address size of the hardware doubles, such flat models reappear. The problem is, that the 'inherent' size of an identifier varies from platform to platform. The maximal size of an identifier therefore becomes a property of an origin server, not a protocol parameter. With regard to URI, RFC 2616 says in section 3.2.1:

> The HTTP protocol does not place any a priori limit on the length of a URI. Servers MUST be able to handle the URI of any resource they serve, and SHOULD be able to handle URIs of unbounded length if they provide GET-based forms that could generate such URIs.

We want to maintain the advantage of identifiers of equal size. We therefore propose to further restrict URI in length and to ignore URI parameters in GET requests. This will also relieve intermediary hosts from scanning Kilobytes of URI before accessing the information that is relevant *for them*. In order to keep identifier size constant within one server, we simply define an identifier to be:

```
Slot*= ARRAY SlotLen+1 OF CHAR;
```

Aside of the file system as container for resources, we also implement a container for dynamic resources in the form of a hashmap. This is realized in the `SNAPdictionary` object. The `SNAPdictionary` object implements the methods `GetHint*():LONGINT` and `IntToSlot*(h:LONGINT; VAR s:Slot)` which allow a request handler object to dynamically find an identifier unique for the host. Each host has its own `SNAPdictionary`. The dynamic request handlers (here the buf object) can be registered and their URI can be published, as illustrated by the code producing an answer to a successful WATCH request:

```
(...)
out.String("HTTP/1.1 201 Created");out.Ln;
out.String("Location: http://");out.String(request.host);
out.Char("/");out.String(buf.path);out.Ln;
out.String("Content-Length: 0");out.Ln;out.Ln;out.Update;
```

4.3 Implementation of timing

In order to allow modeling the dynamic behaviour, the following parameters should be explicitly specified in a WATCH[2] request:

- A maximal sample frequency
- A maximal transfer latency
- A maximal lifetime of a buffer in absence of consumer access

[2] 'to watch something' means to observe it, but the assonance with the timing device is also intended

The capacity of the buffer in number of messages should also become a parameter. Usually, space is specified in Bytes, but since 512 Bytes are the minimal size of an IP packet each host should be able to deal with, we define:

$$\text{capacity} * 512 \leq \text{size} \tag{1}$$

SNAP introduces new request headers `Delay`, `Rate` and `Lease` for this purpose.

The following code fragment shows the implementation of the receiver process for PUSH data. It makes use of the `Delay` parameter to guard the blocking `ReceiveChunk` method:

```
PROCEDURE handleTimeout;
BEGIN client.Close; queue.broken:=TRUE;
END handleTimeout;

BEGIN{ACTIVE}
 NEW(timer);
 res:=Ok;
 WHILE (client.state = AosTCP.Established) & (res=Ok) DO
  Objects.SetTimeout(timer,handleTimeout,maxDelay);
  ReceiveChunk;  (* blocking*)

  IF (res=Ok) & (chunkSize>0) THEN
   NEW(snappet);
   NEW(chunk,chunkSize);

   i:=0;
   WHILE i<chunkSize DO chunk[i]:= buffer[i]; INC(i); END;
   snappet.body:=chunk;

   queue.Post(snappet);
 END;

 IF res=AosIO.EOF THEN (*natural end of data pushed by this producer*)
  client.Close;
  NEW(snappet);
  queue.Post(snappet);
 END;

 Objects.CancelTimeout(timer);
 END;

 FINALLY
 client.Close;
 END SNAPReceptor;
```

4.4 Implementation of generic Request Handler

SNAP implements the error handling model of HTTP, using a generic error notification handler, where appropriate. When the deamon accepts a connection, a handling thread is created. This thread performs all the operations to fulfill the request as depicted in section 3.3.1. In particular, during the process phase,

the generic handler checks, if the URI maps to a registered dynamic request handler before checking for a file with this path. The dynamic request handlers, we name them SNAPins, inherit from this object:

```
SNAPin*= OBJECT
    VAR
      path*: Slot;
      done*, online*:BOOLEAN;
      host: Host;

  PROCEDURE &InitSNAP(CONST slot: Slot; h: Host);
  BEGIN
    host:=h;
    COPY(slot,path);
    path[LEN(path)-1]:=0X;
    online:=FALSE;done:=FALSE;
  END InitSNAP;

  PROCEDURE CheckLease;
  BEGIN
  END CheckLease;

  PROCEDURE CanHandle*(CONST request: Request;
                       CONST p: absPath) : BOOLEAN;
  BEGIN
    RETURN TRUE;
  END CanHandle;

  PROCEDURE Handle*(CONST host: Host; VAR request: Request;
                    VAR in: AosIO.Reader; VAR out: AosIO.Writer);
  BEGIN
    request.error:=WebHTTP.Gone;
  END Handle;

  PROCEDURE Unload*;
  BEGIN
    IF host.snapDic.RemoveSNAPin(SELF) THEN online:=FALSE; END;
  END Unload;

  END SNAPin;
```

The generic request handler can simply call the `CanHandle` method to check appropriateness before calling the `Handle` method. The `CanHandle` method may set an error code in the passed `request` record and let the generic handler produce the error notification. The dynamic buffer is also implemented as a SNAPin. The `CheckLease` method is called by the server's internal garbage collection process in order to detect buffers that have not been accessed for longer than specified with the `Lease` parameter or are overfull and therefore, the `Receptor` code as depicted on page 26 has timed out.

```
try{
        Receive Request
        Parse Request Line and Request Headers
        Resolve Origin URI
        Allocate Buffer
        Register Buffer in Ductionary
        Send PUSH Request to Origin
        Start a Receptor Thread
        Send Succes Notification and URI of buffer to client
catch(any exception){
        Send Error Notification to Client
}
```

Figure 9: A schematic definition of the WATCH request handler on the proxy.

4.5 Implementation of WATCH Request Handler

Most of the work has been spent to find a workaround for existing browsers. The semantics of a WATCH request are implemented using a POST request. On server side, we provide a handler that is loaded by the server, not by client request. The workaround handler is implemented in the `ProxySNAPin` object and registered in the dictionary with the path `(...)/snap`. As a remark, a mail handler, for instance supporting offline modus, could be implemented the same way.

4.6 Implementation of PUSH Request Handler

We have implemented a model object, containg key value pairs. In order to decouple encoding from this model, the coding code is provided in the form of a packer object (visitor pattern). This also provides some guarantees, that the versions of the producer code and the leaf objects are compatible.

```
PROCEDURE Get*(reference: LONGINT; w:AosIO.Writer;p: Packer;);
  VAR l: LeafObject;buf: FlatLeaf;res: LONGINT;
  BEGIN
    l:=first;
    WHILE (l#NIL) & (w.res=AosIO.Ok) DO
      IF reference<=l.leaftick THEN
        l.Get(buf,p,reference, res);
        IF res=Ok THEN
          w.String(buf);
          w.Update;
        END;
      END;
    l:=l.next;END;
  END Get;

PROCEDURE Observe*(CONST w:LogicallyChunkedOutStream; CONST p: Packer);
  VAR
    mytick: LONGINT;
  BEGIN
    Get(0,w,p);
    WHILE (w.res=AosIO.Ok) & running DO
      mytick:=tick;
      BEGIN{EXCLUSIVE}
        AWAIT(tick#mytick);
      END;
      Get(mytick,w.in,p);
    END;
  END Observe;

PROCEDURE Sample*(CONST w:LogicallyChunkedOutStream; CONST p: Packer; CONST rate: LONGINT);
  VAR
    timer: Kernel.Timer; internalrate: LONGINT;
  BEGIN
    ASSERT(rate>0,100);
    NEW(timer);
    internalrate:=rate*1024; (*seconds to milliseconds*)
    WHILE (w.res=AosIO.Ok) & running DO
      Get(0,w.in,p);
      w.Update;
      timer.Sleep(internalrate);
    END;
  END Sample;
```

Figure 10: A stateless observer can be realized by calling the `Observe` respectively the `Sample` method of a `Model` object. `FlatLeaf` is an bounded Array of Byte, thus enforcing that all flattened representations of a data item can be transferred within one chunk. Tokenizers on the other end of the connection can rely on this property.

5 Conclusion

Our aim was to find solutions for use of the world wide web to transport and display data, that are output of dynamic sources and therefore do not profit from caching and can not be prefetched the way audio or video replay are dealt with. Our second goal was to allow more symmetric communicating styles, for instance by sending a URI to a server.

We have proposed a notification mechanism from model to view in a model, view controller architecture over a transport extending HTTP. A proof of concept has been implemented in Active Oberon, keeping the observer stateless in the sense of REST.
We have proposed and implemented a flat data model with explicit identifiers for leafs, the model itself and the version of the model. The maximal length of a data item has been fixed in order to allow latency estimations. Application programmers can express different models of replication, either sampling semantics, or event driven. In the event driven model, we are able to further differentiate hard replication from best effort semantics due to the eggshell model of concurrency implemented in Active Oberon.

We have proposed and implemented means to dynamically allocate buffers in the public part of the internet. The client can reconnect to the buffer, should the connection to the buffer break, without the origin noticing. The possibilities to relocate resources contained in HTTP have been considered in design, but not explored in implementation. We think, that peer to peer networks with today's providers as participants could become possible scenarios. Therefore, we carefully designed the intermediate's view of a buffer as a sequence of bytes, not more.

We have not explored the possibilities of passing along the URI of a buffer, either. This would make sense, if the producing entity could also reconnect to a public buffer. We gave thought to a WAIT method, similar to a WATCH method, but without target, allowing this behaviour. It should also be possible to statically allocate handlers on a proxy server, for instance to implement send queues for offline use. We did not find a suitable solution for the triangle problem, meaning a host sends a message to another host to send output to a third host. Our PUSH approach just requires the establishment of two connections and the price of keepalives, while the triangle way uses three. There might be situations where this overhead is justified, for instance if the price has to be paid per connection time, not per transfer data volume. This might be subject of future work.

6 Sources

The Reference implementation is contained in three Modules:

- SNAP: the server
- WebHTTP: all errorcodes of HTTP
- TicTac: a sample resource

The respective client code is contained in:

- tick.xml
- embedtick.xml
- style.css

The client code was tested using Mozilla Firefox 2.0.0.12 and XForms Plugin V.0.8. Javascript has to be enabled.

References

[1] *Architectural Styles and the Design of Network-based Software Architectures*, Roy T. Fielding, PhD Thesis, University of California, Irvine, 2000

[2] *Web Based Management of IP Networks and Systems*, J.P. Martin-Flatin, PhD Thesis, EPFL, Lausanne, 2003

[3] *A Staged Event-Driven Architecture for Highly Concurrent Server Applications*, Matt Welsh, Berkeley, 2000

[4] *Oberon Script: A Lightweight Compiler and Runtime System for the Web*, Ralph Sommerer, Redmond, Microsoft Research, TR-2006-50, April 2006

[5] *Why Events Are A Bad Idea (for high-concurrency servers)*, Rob von Behren, Jeremy Condit and Eric Brewer, University of Berkeley, published 2003 in: Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems, Lihue, Hawaii

[6] *Project Zonnon: A Compositional Language for Distributed Computing*, Jürg Gutknecht and Roman Mitin, ETH Zurich, in: ICCBSS '08: Proceedings of the Seventh International Conference on Composition-Based Software Systems (ICCBSS 2008), ISBN 978-0-7695-3091-8, Washington, DC, 2008

[7] *On the Duality of operating system structures*, H. C. Lauer and R. M. Needham, in: Second International Symposium on Operation Systems, IR1A, 1978