Integration of XVSM Spaces with the Web to Meet the Challenging Interaction Demands in Pervasive Scenarios

eva Kühn, Johannes Riemer, Richard Mordinyi, Lukas Lechner
Institute of Computer Languages, Space Based Computing Group, Vienna University of Technology
Argentinierstraße 8, 1040 Wien, Austria, Europe
Email:{eva,jr,richard,lukas}@complang.tuwien.ac.at, Web: www.complang.tuwien.ac.at/eva

ABSTRACT

The current Internet is based on the REST (representational state transfer) architectural style to guarantee scalability and to decrease complexity. All interactions are stateless and the communication between client and server is carried out in a synchronous request/response way. However, applications are evolving towards more and more dynamics. In emerging Web2.0 scenarios, our devices will not consume data but report it: A participative communication style will dominate the Internet of the future. Space based middleware can ease collaboration in ubiquitous computing scenarios as it provides symmetric architectures with notification and replication. But the seamless integration of space based middleware with the current Internet is still an open issue. In this paper we analyze how the space based computing paradigm, represented by XVSM (an extensible virtual shared memory implementation), can be provided in the web without fully breaking with the REST principles and by adapting the AJAX (asynchronous JavaScript and XML) principles of dynamically and incrementally building up information at the client side. We propose the implementation of asynchronous transport based on the Bayeux protocol and show the advantages of the resulting architecture in a disaster scenario use case.

Keywords: Space Based Computing, Near-time Collaboration, AJAX

1 INTRODUCTION

Daily life shows that the web is the most successful and wide spread infrastructure for distributed applications of a vast range of domains. In [7] Fielding formalizes the architecture of the web and analyzes how this architecture helps to build scalable, flexible, evolvable distributed hypermedia applications.

Availability of and investments in existing web infrastructure and applications, as well as some properties of the web architecture, like interoperability and zero client maintenance, encourage to adopt the web for mobile computing, or at least to take the web as a starting point to leverage existing infrastructure and force an evolution of web technologies towards the needs of mobile computing.

There are, however, some fundamental limitations of the current web architecture, especially with regard to mobile computing.

a) Scalable information push: A general limitation of the web is the principle of client-initiated information pull. In the classic web architecture there is no possibility to push information from a server to its clients, which is a requirement for applications which rely on timely information, such as collaboration tools, near-time monitoring, or an application visualizing stock values. Currently such applications are typically implemented using AJAX and periodically poll the server to receive fresh data.

b) Connectivity: Mobile devices are characterized by limited internet connectivity with high latency, small bandwidth, and temporary availability. This asks for solutions based on data caching and replication. HTTP request caching based on expiration and validation [8] is limited and moreover web site content typically is dynamic and personalized and thus cannot easily be cached. Caching can therefore hardly be exploited in the context of server-initiated information push. Furthermore, HTTP has no generic support to buffer information produced by an offline client and to transport it to the server when the client gets connected later. As a result, web applications which support offline clients are rare, offer very degraded service in offline mode and are complex to build.

One could solve the mentioned problems by using the space based computing paradigm [30], [9], [14], [22], [5], [12], [2], [23], [21]. In contrast to message queues and topic or content based publish/subscribe, a space offers access to data in arbitrary order and more sophisticated mechanisms for events. It allows for multiple reads of data and the communication via a space is stateful. This means that data in the space reflect the history of

communication and asynchronous mode is provided which is of particular importance for mobile devices. A device may join at any time and can catch up the information in the space. The abstraction of a shared space is inherently suitable to overcome the limitations above:

a) Asynchronous notifications, as for example supported by JavaSpaces [25], Corso [13], and XVSM [15], combine the space based coordination model with the publish/subscribe model [6], which is a scalable technology for information push [3].

b) The shared space model allows distributed implementations supporting caching and replication, like Corso, GigaSpaces [10], and LIME [20], for decreased client-perceived latency, less server load and improved data availability.

c) A space decouples the rate of updating operations from the rate of reading operations, because – as a shared memory – it cumulates the effects of all updating operations. No matter at which rate a client decides to read information from a space (or gets notified about changes), it always accesses a cumulated, current state as an abstraction of all previous updating events.

However, to be able to exploit the advantages of space based computing, seamless integration with today's established web technologies is necessary. This means support of the space paradigm for mobile peers, web servers and web browsers as well. Some space based related approaches propose HTTP or SOAP [11] as protocol to access space servers [1],[26],[27] but do neither focus on space clients running in web browsers nor on how to solve blocking operations and asynchronous notifications over HTTP. We propose an alternative approach which better fits the needs of web browsers, leverages state-of-the-art event transport mechanisms over HTTP, allows scalable server implementations and enables extensions for further scalability and support for mobile devices.

The first building block is the extensible virtual shared memory implementation XVSM (see section 4) which provides a symmetric architecture where each peers hosts a space. The resources in such a space are called containers and are addressable in the Internet via URLs. The second building block is a web-compatible protocol we call "XVSMP/Bayeux" to access shared XVSM spaces in the web [16]. The protocol uses techniques for web-based event delivery known under the term "Comet" [31], which are abstracted and specified by the "Bayeux" protocol [24]. The third building block is a set of optional, transparent extensions such as client-side caching and intermediaries like caching and event-routing proxies, which make the architecture scalable and add support for mobile devices. This incremental approach allows adopting space based computing in the web with minimal initial effort and without changes in the web infrastructure.

In this paper we present the XVSMP/Bayeux protocol as well as prototype implementations of a client library and space-server component. Section 2 motivates the work by means of a use case scenario. Section 3 describes Bayeux, which is the base of our proposed protocol. In section 4 we briefly describe XVSM, the protocol XVSMP/Bayeux as well as implementations of a JavaScript client library and servlet-based space server component. In section 5 a solution for two major implementation challenges of the use case is presented. In section 6 we evaluate how XVSMP/Bayeux can overcome the limitations mentioned above.

2 USE CASE SCENARIO

The aim of this section is to demonstrate the field of application of the proposed protocol by describing an emergency scenario. The use case is a disaster scenario in which e.g. an earthquake occurred and several people were injured. Due to the catastrophe the infrastructure has collapsed and so there is no reliable network connection; unreliable ad hoc networks are the only means for communication at the disaster area.

The roles in that scenario refer to ambulances, doctors on-site, the hospital, and patients. Communication between doctors and between ambulances and doctors is by means of the unreliable ad-hoc networks. Spaces are a well suited approach for such kind of scenarios [18], [19]. In addition, the integration to the standard web technologies as used by ambulances and hospitals is required.

Patients are grouped into three different classes. The first class refers to patients who haven't been looked at yet. The second class represents patients who were already visited by doctors but have smaller non critical injuries. The third class refers to patients with critical injuries and who need to be transferred to hospital as soon as possible.

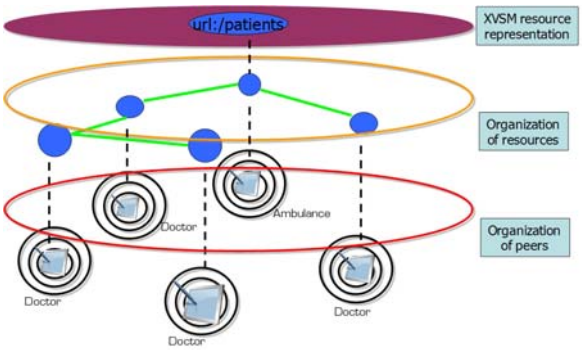

Figure 1: Sharing data in XVSM

Doctors are equipped with tablet pcs supporting wireless ad hoc network communication and hurry from one patient to the next one. The responsibility of a doctor is to inspect the patient, to label the patient physically according to the class the patient

belongs to and then to insert collected data like criticality, prescribed medicines, and if possible personal information into the shared space via the doctor's tablet pc. As shown in Figure 1, since the doctor's equipment supports wireless communication and XVSM provides P2P lookup and communication infra-structures [17] and replication protocols [20], [13], any data added to the space is replicated to any other doctor who is reachable via the ad-hoc network.

Distribution of newly added patient data to other doctors is not a complicated issue since there are no conflicts in sense of inconsistencies: Only one doctor will look at a patient at the same time. If another doctor has to update the patient data, because its state has become worse, there will be still no conflicts because the doctor is in turn the only one that updates this specific information. Since the first doctor has added her/his information the currently checking one may be able to contact her/him or a staff member in the ambulance or the hospital for consultation via chat-communication. So, by writing patient information into the space the doctors coordinate their activities so that no patient is treated twice, unless it is really necessary, and will not receive redundant medication.

Ambulances are responsible to pick up the patient with highest criticality and in the closest proximity and to bring her/him to the hospital for further treatment. In order to do so, each ambulance has a pc with a web server that is connected with the web server in the hospital, and a terminal with a web browser for monitoring the catastrophic site. Via the web browser's gui also the chat with the doctors in the fields shall be carried out, as well as the coordination with other ambulances. Here the need arises that the standard web technologies must be integrated with the space to be notified about the latest states regarding the patients in near-time. This is necessary since ambulances are dynamic in the sense that they move around and thus may leave a communication area without being reachable for anyone. Once they have all the information they can make a decision based on which patient is the most critical one, who is the closest to its position and whether there is any other ambulance on its way there. So, ambulances coordinate each other by writing their intention into the space by specifying the location of the patient they want to pick up next. This is necessary, since it would be a waste of resources if two ambulances were going for the same patient.

In the *hospital* web servers and web browser terminals exist. The collaboration here comprises the communication with the ambulances, the active pulling of information about certain patients (e.g. on the relatives' requests), and if special expertise or advice is needed, an on-line chat with the doctors at the emergency site.

The proposed XVSMP/Bayeux protocol is needed to integrate the ambulances and the hospital with the doctors; i.e. to notify them about any changes occurring in the space, and to enable the described collaborations among all stakeholders.

3 BAYEUX

As stated in the Introduction, our goal is to adopt the space based computing paradigm in a web environment. In our approach shared data spaces are hosted by web servers, which are accessed by web clients to read data from the space, write data to the space and receive notifications about data of interest from the space.

In the web architecture, web clients access web servers using the HTTP protocol to retrieve or submit representations, like HTML or XML documents, of resources, which are identified by URIs. A client initiates the communication by establishing a TCP connection with the server and sends an HTTP request. The server processes the request and sends an HTTP response using the established TCP connection. When the HTTP response is complete, further communication must be initiated by the client again. There is no way for a server to initiate communication with a client.

The request/response interaction style of HTTP fits well to realize non-blocking read and write operations of the space. There are, however, space operations which cannot be realized well on HTTP: *blocking reads* and server initiated *notifications*.

A *blocking read* is issued by a client to retrieve some data of interest. If, however, there is currently no such data available in the space, the read operation blocks until eventually such data arrives, i.e. is written to the space by another client. The space then sends the data back to the waiting client and the read operation unblocks.

A *notification* is used by clients and is a permanent subscription to changes in the space. A client registers its interest in some kind of data at the space. The space sends a notification to the registered client whenever matching data arrives in the space.

The blocking read operation and notifications are difficult to realize based on HTTP, because they break the stateless request/response interaction style. Firstly, the server must keep track of all clients which wait for a blocking read operation or have registered notifications. Secondly, in order to send notifications, a server needs the possibility to initiate communication with a client.

There are well known HTTP-based approaches to overcome these limitations of plain HTTP, like periodic polling, long polling and the Iframe technique, which are described below. Instead of adopting one of these approaches to realize communication between clients and servers hosting spaces, we decided to reuse the HTTP-based

protocol Bayeux, which supports *all* mentioned approaches and provides a protocol negotiation mechanism for interoperability reasons. Further, there are many existing client and server side implementations of Bayeux, which can be reused for the implementation of the XVSMP/Bayeux protocol.

Bayeux is a publish/subscribe protocol. Compared to other publish/subscribe approaches [6], the unique property of Bayeux is that it has been designed for the web. It takes into account the characteristics and limitations of web browsers, web servers and HTTP.

Bayeux allows a client, typically living in a JavaScript and DOM (document object model) enabled web browser, to publish events via channels (cf. topics in publish/subscribe systems [6]) to a server and to subscribe to channels in order to receive events from a server. The Bayeux protocol consists of a set of messages encoded in JSON [4] (JavaScript Object Notation), a lightweight, textual representation of JavaScript objects optimized for processing in JavaScript. Channel names are organized hierarchically (e.g. "stockvalues/XYZ") and subscription supports prefix matches on channel names (e.g. "stockvalues/*").

Although Bayeux is not limited to HTTP as underlying protocol, it has a strong focus on HTTP. Within the browser, an HTTP connection is usually created using a special JavaScript object called "XmlHttpRequest" (XHR). Via an XHR, a client can send any HTTP request such as GET and POST. When the reply of an HTTP request has been received by the XHR object, it calls a previously provided callback function. This function typically updates the web page by accessing the DOM of the page. A Bayeux client uses two HTTP connections, one to send Bayeux messages to the server and another one to receive Bayeux messages from the server.

The advantage of Bayeux compared to *periodic* AJAX-based *polling* is that it supports various transport mechanisms which allow messages to be delivered to clients in a more timely fashion. In addition to AJAX-like *periodic polling* Bayeux supports the transports *long polling*, *Iframe* (HTTP-based) and Flash, which provide significant lower event delivery latency. In the following we focus on some HTTP-based transport examples and assume a browser as client. Figure 2 shows the message exchanges between a client and a server for three different transports. All message exchanges start with a handshake.

Handshake. The transport actually used is negotiated by a handshake. The client sends a handshake message via an HTTP request using XHR and gets a response containing a unique client-ID, which is used by the client for all subsequent requests, and a list of all transports supported by the server. The client selects a transport and specifies it in the subsequent *connect* request. The further message exchange depends on the negotiated transport.

Periodic polling. (Figure 2.a) After the handshake the client sends a *connect* message using XHR. The server immediately sends a response containing all outstanding events – even if there are none. In case of periodic polling, the handshake response also contains an advice specifying a polling interval. The client waits this interval before sending a *reconnect* message for the next poll. The latency of events delivery depends on the rate of polling, TCP/HTTP request setup and actual data transmission time.

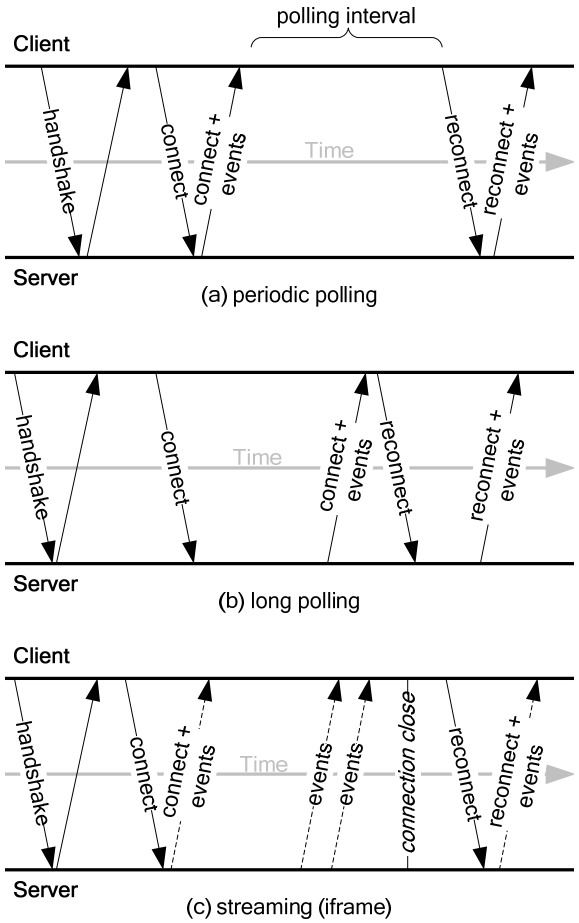

Figure 2: Transport examples of Bayeux, based on [24]. Solid arrows represent HTTP requests (client to server) and HTTP responses (server to client). Dashed arrows (server to client) represent partial HTTP responses.

Long polling (Figure 2.b) works similar to simple polling. When the server handles a *connect* request and there are no outstanding events, it does, however, not generate an HTTP response but keeps the HTTP connection open until eventually an event is to be delivered. After receiving a *connect* response, a client immediately issues a subsequent *reconnect*

request to receive further events. Considerable latency for event delivery is still caused by the overhead involved to set up a new HTTP request for repeated polling and the actual data transmission time.

Streaming (Iframe). (Figure 2.c) This transport allows to stream events over a long-lived HTTP-request which is never closed. This transport does not rely on XHR, but uses an HTML Iframe element for *connect* and *reconnect* requests. An Iframe is normally used to embed a web site within another one. The embedded web site is loaded independently from the embedding one. Whenever a script is completely loaded during loading an Iframe, the script is immediately executed by the browser *before* the Iframe itself is completely loaded. To send events through an Iframe, the Iframe keeps loading forever – resulting in a continually open HTTP request, called a *tunnel*. To send an event, the server streams a script carrying the event through the tunnel without closing the tunnel afterwards. When executed in the browser, this script calls some generic event dispatching mechanism. If the tunnel is closed for some reason, the client re-opens the tunnel via a *reconnect* request. By preserving an open HTTP channel, latency is reduced to a minimum and depends on the actual data transmission size only.

These different transports not only differ in the provided latency of event delivery, but also in browser compatibility, especially concerning older versions, the possibility to subscribe to servers distinct from the server hosting the application, also known as cross domain scripting, and others details. For that reason, Bayeux includes a negotiation mechanism for clients and servers to agree on a common transport. It further provides a mechanism to fall back to an alternative transport, if problems for a given transport are detected.

A drawback of Bayeux is its complexity. Advanced transports are far more complicated than issuing a simple, REST-compatible HTTP request. However, it is promising and simple enough that some vendors integrated it into their products or tools, like JavaScript toolkit dojo, Jetty Web Server, IBM WebSphere, Twisted Python Cometd Server and Sun Grizzley.

4 XVSM SPACES

XVSM stands for *extensible virtual shared memory* and defines a Linda [9] tuple space based extension that offers more structuring of the coordination space and that can be used in the Internet using an XML protocol. The basic concept is a container that is addressed by its “cref” (container reference URL) so that a container becomes accessible like any other resource in the Internet. A container can be bounded or unbounded: Bounded means that within a container there is a limited amount of places for so-called data entries. A container can be considered a named sub-space. Entries have no identity and can either be structured Linda tuples (cf. database records), any basic data type, XML data, RDF data, or references to other containers.

The basic operations on containers are *read* (read of entries), *write* (insertion of entries), and *take* (consuming read of entries), according to the classic Linda model. Also bulk operations are supported, so that it is possible to insert multiple entries into a container resp. to read/take multiple out of it in one step. Besides the fact that read and take can be used for synchronization, as both offer a blocking behavior if no suitable entries to be read/taken are in the container, also the write operation may block, if there is no place in the container. Therefore another operation was added called *shift* that behaves like write but instead of blocking if the data cannot be written to the container, it will replace an accordingly number of entries and succeed in any case without blocking.

The basic semantics of read, take, and write are analogous to the Linda model. Shift as explained above behaves like write if there is place in the container to write the entry/entries to it, otherwise it will replace one or more entries. In addition a *destroy* operation is supported which behaves like take but does not return the removed entries. Read, take and destroy are different flavors of selecting entries; they differentiate by either removing or not removing the selected entries and by returning them or not.

An entry can be tagged with named keys and/or labels when they are inserted into a container using write or shift. A key must be unique in a container, whereas a label need not: A write will block if an entry is written whose key is already occupied. Read, take and destroy use an “XVSM query” to select entries. In a query one can specify:

- Label and/or key values: a selected entry must provide all given labels and keys. It is also possible to specify a key/label range, or the minimum/maximum value of a key/label with an XVSM query.
- One or more match-maker functions, each with a template: the function using template must be fulfilled by the entry. Match-maker functions can be added dynamically, thus extending the expressiveness of the query. E.g., linda template matching, XML Xpath, RDF query mechanisms etc.

With these mechanisms, more complex coordination possibilities like fifo, filo, random can be implemented thus adding further expressiveness to the classic Linda model.

Aspects can be defined on containers that in contrast to reactions in LIME [22], trigger the execution of program code upon the execution of one

of the mentioned operations (read, write etc.) on a container, either before it is called or after it has been executed. With aspects, notifications and iterators can be implemented.

Our current open source implementation of XVSM is carried out in Java. It is provided as a version that can be embedded by a (mobile) Java peer, as a standalone Java API, and as a web server version (this one uses the derby database, and Tomcat as web server).

The XVSM architecture comprises a space core, and function profiles that are pluggable and may provide extensions for persistency, life cycle management, and several replication strategies between distributed containers (see Figure 3). In addition, the architecture comprises language bindings that are layered above the core and use the XML protocol. We currently support a Java binding at the interface layer; prototypes for .NET, .Scheme and JPhyton are under development.

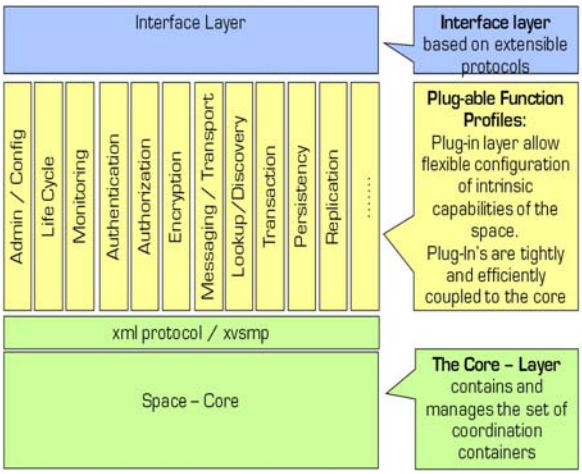

Figure 3: XVSM Architecture

In the following we use XVSM for combining web architectural style with the advantages of space based computing.

4.1 XVSMP/BAYEUX PROTOCOL

XVSMP/Bayeux uses Bayeux as underlying communication protocol to provide bidirectional asynchronous communication between web applications and XVSM spaces. Bayeux provides a publish-subscribe message exchange mechanism, however as we require direct communication between client and XVSM we run a request-response message exchange on top of it. We therefore define a common server channel "/xvsm/server" to support client to XVSM communication. Messages from XVSM to the client are published to the client's private channel. Within Bayeux every client has its own private channel, which it is automatically subscribed to upon connecting.

For our protocol we are using the Bayeux implementation of the web server Jetty. It includes an extensible security mechanism which we use to protect our system from malicious clients. It can be used to regulate subscription and publishing rights on channels. We use it to prohibit clients from subscribing channels, other than their private one, and from publishing to channels, other than the common server channel. This way, malicious clients are kept from eavesdropping or infiltrating other client's communication.

For our protocol we use JSON as message encoding format. The following example shows the message exchange to create a container within the XVSM space and write an entry to it:

```
{ "operation": "CreateContainer",
  "request": 1,
  "data": { "size": 10 } }
```

Every message is represented using a JSON object and contains three parameters. The "operation" parameter defines the operation to be performed in the XVSM space (in this case the creation of a new container). The "request" parameter is an increasing number to keep track of the message exchange: (E.g.: the responses of two write operations might arrive at the client in arbitrary order, depending on the time needed for their execution at the XVSM space). For correlation, a response to a specific request must contain the same number in the "request" parameter. The "data" parameter holds a JSON object itself, containing the necessary information to perform the specific operation (in the above example the container to be created shall be bounded to size 10). If successful the corresponding response from XVSM looks like this:

```
{ "request": 1,
  "operation": "CreateContainer",
  "data":{ "cref": "cref_117_0" } }
```

The "request" parameter correlates the response with the previously defined request. The "data" parameter again contains a JSON object. It holds the container reference "cref" of the newly created container. This cref is now used to perform a write operation on the container in the subsequent example:

```
{ "operation": "Write",
  "request": 2,
  "data": {   "cref": "cref_117_0",
             "entries":[{
              "type":"STRING_UTF8",
               "value":"test" } ]
          }
}
```

The parameter "cref" identifies the container to write to. The parameter "entries" contains a JSON array of entries to be written. Every entry is a JSON object with a type and a value parameter. Valid types

are strings, numbers, container references or tuples. A tuple can itself contain an arbitrary number of entries, permitting to build up more complex data structures.

4.2 Server implementation

The main component of the XVSM space is the XVSM Core. It can be embedded into a Java application (e.g. a servlet container) or it can be used as a standalone version. It consists of three parts:

The Core API (CAPI) provides a Java interface for the application to interact with the XVSM Core, hiding the concrete implementation.

The XVSM Core contains the main logic of the XVSM system.

The data storing entity. The current XVSM implementation uses the derby [29] database to persist data.

To provide access to XVSM we embed the XVSM Core into the web server Jetty. The server implementation consists of a servlet subscribing the Bayeux channel "/xvsm/server" and implementing the XVSMP/Bayeux protocol. It takes incoming messages from the clients and forwards them to the processing classes (every command to be executed is processed by one class extending a predefined interface). A security handler for subscribing and sending messages via channels is added as described in section 4.1.

Jetty is chosen instead of Tomcat, as in the original XVSM implementation, because it already includes a server implementation of the Bayeux protocol and it is designed to meet the new demands that event driven web applications put on the server infrastructure.

4.3 Client library

To simplify the development of web applications using XVSM we created a JavaScript API. This API offers methods to directly perform operations on XVSM. This way, the underlying Bayeux communication as well as the JSON based protocol is made transparent to the web application developer. Our JavaScript API makes use of the JavaScript toolkit dojo. The dojo toolkit is designed to facilitate the development of web applications with JavaScript. It offers a wide range of aid for JavaScript developers including widgets (a combination of HTML, CSS and JavaScript code), as building blocks for the user interface, a packaging system for code modularization and reuse and support for asynchronous communication (including a Bayeux client). A lot of widgets are already included in the toolkit, implementing the most commonly used design features of modern web applications. Additionally custom widget's can be created. Dojo's packaging system arranges code in a similar way like the Java packaging system and allows JavaScript coding in an object-oriented way. To illustrate the use of the API we take a look at the operations performed in section 4.1. First the JavaScript API must be instantiated and connected to the XVSM space:

```
var jsapi = new xvsm.jsapi.JSapi();
jsapi.connect("url/to/xvsm");
```

The following code segment shows how to create a container and write an entry to it (for reasons of simplicity the example doesn't include full package names):

```
var iface = new CreateContainerIface();
iface.containerCreated(cref) {
  var w = new Write();
  var e = new Entry();
  e.setString("test");
  e.setContainerRef(cref);
  w.addEntry(e);
  jsapi.write(w, new WriteInterface…);
}

jsapi.createContainer(new
 CreateContainer(10), iface);
```

Every function, to perform an operation on the XVSM space, takes two parameters as arguments. The first parameter specifies the necessary information to perform the operation (in this example the size of the container to be created or the data to be written) and the second parameter implements an interface for return values. The interface's functions are called when the operation on the XVSM returned (either successfully or with an error). This parameter can be omitted if the web application needs no information about the outcome of the operation.

In this example we first create an instance of the CreateContainerInterface. Its containerCreated method is called upon successful creation of the new container and returns its reference. Within this method we now create a "Write" object and add an entry to it. We than call the write method of the JavaScript API (for simplicity we also omitted the methods of the CreateContainerInterface being called upon an error).

Finally we can call the createContainer method of the JSapi with a "CreateContainer" object setting the size to 10 and the previously created interface object. The JavaScript API will transform the data into JSON messages equal to the ones in section 4.1 and send them to the XVSMP/Bayeux server implementation.

5 IMPLEMENTATION OF USE CASE ASPECTS

As a proof of concept and to illustrate how the presented technologies work together we have

developed two issues of the presented use case scenario (see section 2): a chat application (used by doctors, ambulances and the hospital) and a monitoring application (used by ambulances and hospital within a web browser). The following descriptions will explain only the web server/browser related part of the implementation. The implementation of a chat between XVSM peers is straight forward by writing to and taking from a container with fifo coordination (as described in section 4), and monitoring can be done using space notifications directly.

Figure 4 shows the system design of the two sample applications. The server implementation resides within the web server Jetty. The XVSMP/Bayeux implementation binds together the Bayeux server on the one side and the XVSM space on the other side. It subscribes the predefined Bayeux channel "/xvsm/server" to retrieve client messages which are interpreted and executed on the XVSM space. Responses are sent back to the client's private channel. On the client side the application lives in an internet browser environment. The web application calls the functions of the JavaScript API which itself sends messages through the Bayeux client to the server. Both web application and JavaScript API are developed with the help of the dojo toolkit and the Bayeux client is a part of it.

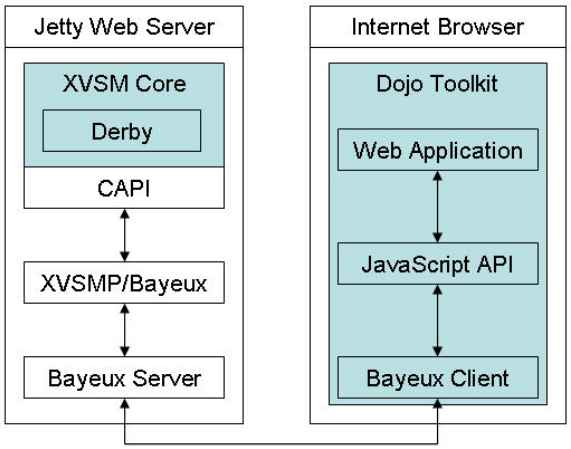

Figure 4: System Concept

5.1 Chat Application

The chat application allows the user to choose a nickname, log into the system, create channels, join channels and send messages to channels. The coordination and data exchange of the application is based on the following data structure:

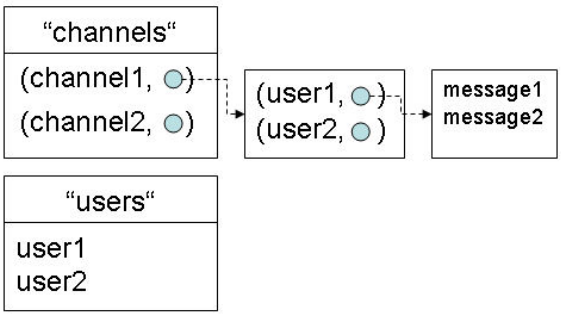

Figure 5: Chat application data structure

The application uses two named containers "channels" and "users" to keep track of all of its data. The references of these two containers can be obtained through the lookup mechanism of XVSM. Both containers use a coordination type based on a key/value structure. This coordination type ensures that every key can only be present once in the specific container, which we use to guarantee the uniqueness of nicknames and channel names. If the application tries to write an already existing nickname to the "users" container the operation will block and wait for the existing key/value pair to be removed. Every operation offering a blocking behavior takes a timeout as parameter to indicate how long the operation shall wait in the blocking state before giving up. By setting the timeout of the write operation to zero, every time the write operation blocks, it will immediately throw a TimeoutException and the application can inform the user about the already existing nickname.

While the "user" container only stores simple usernames, the "channel" container stores tuples, containing the name of the channel and a reference to the channel's container (references are shown as circles in Figure 5). Every channel has its own container storing the currently participating users with a reference to their container within the channel. Within a channel every user got his/her private container where messages are delivered for him/her. The retrieval of newly available messages is this way uncoupled from their creation and every application can read messages from the channel at its own speed, unaffected by the other application's operations.

To stay informed about the actions from other users every application uses multiple notifications. A notification is created upon the "channels" container to get notified whenever a user creates a new channel. Additionally when a user joins a channel two notifications are created on the container of the channel. One for entries being written to the container (a new user joins the channel), one for entries being removed from the channel (a user leaving the channel). A notification is also created to listen for entries written to the user's container within the channel. When a user leaves a channel his/her container in the channel is destroyed and the tuple with his/her username and container reference

is removed. As the user has no more interest in the channel's activities his/her notifications on the channel are canceled.

5.2 XVSM Viewer

The XVSM Viewer is a monitoring application used to show the current state and content of a XVSM space. It displays the currently existing containers, their entries and properties as well as existing notifications on the containers. The viewer displays the list of container references and uses two notifications (one that fires upon creation of a container and one that fires upon destruction of a container) to keep the list up to date. Filtering with regular expressions and selecting favorites facilitates the navigation within the list of references. The user can select a reference from the list to open the container's window. The container window shows the details of the selected container (its properties, entries and notifications). Additional notifications are added to notify the XVSM Viewer about operations performed on the specific container (write/shift/take/delete). Container references within entries can be used to directly open their container window (similar to following a hyperlink), allowing to quickly browse complex data structures. With a pause button the container window can be stopped from updating the container's content in order to create a snapshot. It is also possible to open a container more than once to make snapshots at different times. An MVC pattern is used to manage the container's data at a single point no matter how many windows displaying the container's content are opened. If all windows of a container are closed the notifications on the container are canceled and its data is discarded.

6 EVALUATION

The use case description shows the importance of timely information delivery in a scalable collaborative application, which is used by autonomous participants to coordinate their activities. Participants rely on timely information in order to plan the optimal use of available resources. The space based computing paradigm fits well for this kind of application, since it supports many-to-many coordination, dynamically joining and leaving participants, replication and timely event delivery. Web applications, on the other hand, have the fundamental advantage that besides a conventional web browser no client software needs to be installed and maintained. This is crucial for large scale applications used by various, autonomous organizations.

XVSMP/Bayeux enables space based computing within a web environment. Important properties of this protocol with regard to the limitations of the web architecture described in section 1 and various extensions to show the potential of our approach are outlined in the following.

a) Scalable information push. For scalability reasons, the HTTP specification restricts the number of simultaneous connections of a client to one server to two. For that reason, realizing each blocking operation with a separate long-lived HTTP request, as proposed in [25], is not appropriate. A workaround is to configure the browser to allow more than two connections, a web application should, however, not rely on that. XVSMP/Bayeux creates a tunnel using a single long-lived HTTP request instead and dispatches all events to the according callback functions. This way any number of blocking operations and notifications can be issued simultaneously.

Web applications using Bayeux as underlying transport got a significantly different traffic profile compared to traditional web applications, which needs to be addressed by the server architecture to keep up scalability. Traditional web server architectures are based on the "one-thread-per-connection" model. This model scales well when the connection can be closed while the user reads the content or fills out a form. This way the application can handle far more users than the web server can handle connections. However with long-lived HTTP requests a connection is maintained even if no data is transferred. This makes the "one-thread-per-connection" model unsuitable, as every user is constantly using at least one connection. To overcome this problem, the creators of Jetty introduced the concept of "Continuations" [28]. Continuations offer the possibility to suspend a long-lived HTTP request and free the processing thread. The request is resumed after a specific timeout or if another thread calls its resume method. With this technique open connections only need a processing thread whenever data is actually received or sent.

	Formula	Web 1.0	Web 2.0 + Comet	Web 2.0 + Comet + Continuations
Users	u	10000	10000	10000
Requests/Burst	b	5	2	2
Burst period (s)	p	20	5	5
Request Duration (s)	d	0.200	0.150	0.175
Poll Duration (s)	D	0	10	10
Request rate (req/s)	rr=u*b/20	2500	4000	4000
Poll rate (req/s)	pr=u/d	0	1000	1000
Total (req/s)	r=rr+pr	2500	5000	5000
Concurrent requests	c=rr*d+pr*D	500	10600	10700
Min Threads	T=c T=r*d	500 -	10600 -	- 875
Stack memory	S=64*1024*T	32MB	694MB	57MB

Table 1: Resource consumption of Bayeux applications (taken from [28])

Table 1 is a theoretical calculation estimating the impact of Bayeux applications on the resource consumption of traditional web servers and web servers using Continuations. Based on the assumption of 10.000 simultaneous users the concurrent requests are estimated. With a traditional web application (Web 1.0) 500 concurrent requests would be needed, resulting in 500 running threads at the server, consuming 32 MB of stack memory. A Bayeux application(Web 2.0 + Comet) running on a traditional server, would need 10600 requests resulting in an enormous need of 10600 threads and 694 MB of memory. With Continuations the amount of needed threads for 10700 requests is reduced to 875 (57 MB of memory), which is far more reasonable.

Further scalability can be achieved by replicating the space on a server cluster or by deploying shared XVSMP/Bayeux proxies.

b) Performance. Using Bayeux as underlying transport is supposed to provide significant performance advantages over classic Ajax polling regarding asynchronous notifications. To prove this expectation we conducted a simple benchmark comparing asynchronous notifications using Bayeux to asynchronous notifications using traditional Ajax polling. The benchmark consists of a thread, which is constantly writing entries, containing the current timestamp, to a container. The delay between two write operations is randomly determined within a certain interval. Two web applications, one using XVSMP/Bayeux and another one using Ajax polling register at the space to be notified about the write events. Whenever an entry is received the difference between the entry's timestamp and the actual time is displayed. Additionally the Ajax application is counting the polling attempts, where no new entry was available. To avoid clock synchronization problems the benchmark is executed on one machine.

Write delay	Polling freq.	Bayeux latency	Ajax latency	Ajax empty
500ms	2s	132ms	1243ms	0.34%
500ms	1s	114ms	675ms	3.78%
500ms	0.5s	102ms	399ms	28.1%
500ms	max	116ms	236ms	545%
200ms	max	134ms	219ms	750%
100ms	max	122ms	207ms	375%
50ms	max	128ms	221ms	320%
0ms	max	123ms	215ms	223%

Table 2: Benchmark between Bayeux (using long-polling transport) and classic Ajax

Table 2 shows the latency of both approaches depending on the different write delays and the different polling speeds. The maximum polling speed on the testing device was about one polling request every 60ms. Considering the benchmark results the latency using Bayeux is constant and much lower than with Ajax polling. At its best, the latency with Ajax polling is nearly double the latency of Bayeux. Additionally Bayeux is avoiding the overhead of empty polling messages.

c) Connectivity. Long-lived HTTP-requests have the inherent problem that they can break if the Internet connection is bad or times out, especially in the presence of intermediary proxies (which are not in control of the application hosting organization). Such broken HTTP requests are automatically masked by XVSMP/Bayeux.

Caching greatly reduces client-perceived latency and network traffic. While HTTP caching is basically limited to static contents like images, XVSMP/Bayeux enables to cache parts of the space and to keep them automatically consistent in near-time by receiving updates or cache invalidation messages from the origin server. This is particularly feasible if the read/write rate is comparable high. A prerequisite for that is a protocol like XVSMP/Bayeux which allows scalable, efficient event push. Caching and replication further assist in building clients which support offline operation. Advanced replication strategies allow to buffer data collected when being offline and to automatically transfer it to the space server when online again. Since our approach is based on JavaScript at client side, it is possible to integrate the cache and replication logic within the client library and avoid the need for proprietary browser plug-ins.

7 CONCLUSION

The demanding requirements imposed by increasing amounts of mobile clients and more dynamic application scenarios challenge the strict REST principles. The AJAX style is an approach that supports better scalability through finer granularity caching, but still does not solve the automatic notification problem for clients. We have adopted this idea of separating business logic and communication to achieve even more flexibility and scalability by integrating current web architectures with the space based computing paradigm which inherently offers the desired properties like caching and asynchronous near-time notification.

To support space based computing in a fully integrated way, the usage of REST and SOAP mechanisms would be obvious, but are not sufficient, because the server side pull must be simulated by inefficient polling. We have therefore implemented a new transport protocol termed XSVMP/Bayeux that supports synchronicity in communication and that is based on the Bayeux protocol which supports a publish/subscribe style for web browsers breaking

with the REST principle only in that one open durable channel is required. At the client side, a JavaScript program is loaded, comparable to the AJAX idea, that itself exhibits some space properties concerning caching and that provides feasible, limited application availability for mobile clients even in off-line mode. This script executes the XVSM API that communicates with the XVSM space located at the server via XSVMP/Bayeux. A prototype was implemented using the open source implementation of the XVSM.

This architecture contributes to better caching through effectively maintaining shared data in replicated spaces, and scalability through avoidance of polling. In addition it offers new application possibilities. These comprise near-time automatic refresh in web browsers, a scalable and configurable pushing of information to clients depending on the device's capacity and connectivity, and the easy support of on/off-line mode for mobile devices.

Future work will consider the benchmarking of different caching scenarios under different loads and the extension of the space footprint.

Acknowledgments: We would like to thank Christian Schreiber for his helpful comments on this paper.

8 REFERENCES

[1] C. Bussler: A Minimal Triple Space Computing Architecture, in Procs. of the WIW'05 Workshop on WSMO Implementations (2005).

[2] G. Cabri, L. Leonardi and F. Zambonelli: MARS: A Programmable Coordination Architecture for Mobile Agents, IEEE Internet Computing 4, 4, 26-3 (Jul. 2000).

[3] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf: Design and evaluation of a wide-area event notification service, ACM Trans. Comput. Syst. 19, 3, pp.332-383 (Aug. 2001).

[4] D. Crockford: The application/json Media Type for JavaScript Object Notation (JSON), RFC4627 (July 2006)

[5] N. Davies, S. P. Wade, A. Friday and G. S. Blair: Limbo: A Tuple Space Based Platform for Adaptive Mobile Applications, Proceedings of the International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP '97), Toronto, Canada, 27-30, pp. 291-302 (May 1997).

[6] P.T. Eugster, P.A. Felber, R. Guerraoui, and A.M. Kermarrec: The Many Faces of Publish/Subscribe, ACM Computing Survey 35, 2 (Jun. 2003).

[7] R.T. Fielding: Architectural Styles And The Design of Network-based Sofware Architectures, PhD Thesis, University of California, Irvine (2000).

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

[9] D. Gelernter: Generative Communication in Linda, ACM Transactions on. Programming. Language and Systems (TOPLAS), Vol. 7, No. 1, pp. 80-112 (1985).

[10] GigaSpaces: GigaSpaces Enterprise Application Grid Version 4.1 Documentation.

[11] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H.F. Nielsen: SOAP Version 1.2 Part 1: Messaging Framework, W3C Recommendation (June 2003).

[12] K. Hummel: Mobility-Aware Distributed Computing in Shared Data Spaces, PhD Thesis, Vienna University of Technology (January 2005).

[13] e. Kühn: Fault-Tolerance for Communicating Multidatabase Transactions, Proceedings of the 27th Hawaii International Conference on System Sciences (HICSS), ACM, IEEE, Vol. 4., No. 7, Wailea, Maui, Hawaii (1994).

[14] e. Kühn: Virtual Shared Memory for Distributed Architecture, Nova Science Publishers (2001).

[15] e. Kühn, J. Riemer, and G. Joskowicz: eXtensible Extensible Virtual Shared Memory (XVSM) - Architecture and Application, Technical Report, Institute of Computer Languages, Vienna University of Technology, Austria (June 2005).

[16] e. Kühn, J. Riemer, and L. Lechner: XVSMP/Bayeux: A Protocol for Scalable Space Based Computing in the Web, Workshop on Interdisciplinary Aspects of Coordination Applied to Pervasive Environments: Models and Applications (CoMA), at the 16th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE), Paris, France (June, 2007).

[17] Z. Li and M. Parashar: Comet:A Scalable Coordination Space for Decentralized Distributed Environments, In Proceedings of 2nd International Workshop on HotTopics in Peer-to-Peer Systems (HOT-P2P'05).

[18] M. Mor, R. Mordinyi, and J. Riemer: 2007. Using Space-Based Computing for More Efficient Group Coordination and Monitoring in an Event-Based Work Management System. In Proceedings of the the Second international Conference on Availability, Reliability and Security (April 10 - 13, 2007). ARES. IEEE Computer Society.

[19] A. L. Murphy and G. P. Picco: Using Coordination Middleware for Location-Aware Computing: A LIME Case Study, Coordination 2004, LNCS (2004).

[20] A.L. Murphy, and G.P. Picco: Using LIME to

Support Replication for Availability in Mobile Ad Hoc Networks, Coordination 2006, LNCS, 2006.
[21] A. Omicini and F. Zambonelli: TuCSoN: A Coordination Model for Mobile Agents, J. Internet Research 8, 5, pp.400-413 (1998).
[22] G.P. Picco, A.L. Murphy, and G.-C. Roman: Lime: Linda Meets Mobility, In Proc. of the 21st Int. Conference on Software Engineering (ICSE'99), ACM Press, Los Angeles, USA (1999).
[23] D. Rossi, G. Cabri and E. Denti: Tuple-based technologies for coordination, in Coordination of internet Agents: Models, Technologies, and Applications, Springer-Verlag, London, pp.83-109 (2001).
[24] A. Russell, G. Wilkins, D. Davis and M. Nesbitt: Bayeux Protocol - Bayeux 1.0 draft1, The Dojo Foundation, 2007.
[25] Sun Microsystems: JavaspacesTM service specification (2003).
[26] P. Thompson: Ruple: an XML Space Implementation, in Proc. of XML Europe 2002 Conference, Barcelona, Spain (2002).
[27] G.C. Wells: A Tuple Space Web Service for Distributed Programming, in Proc. of 2006 Int. Conf. on Parallel & Distributed Processing Techniques and Applications, Las Vegas, USA, (2006).
[28] G. Wilkins: Ajax, Comet and Jetty, available at http://www.webtide.com/downloads/whitePaper AjaxJetty.html (visited on Jan 18, 2008).
[29] P. C. Zikopolous, G. Baklarz and D.Scott, Apache Derby, IBM Cloudscape, Prentice Hall PTR, 2005.
[30] www.spacebasedcomputing.org (visited Jan. 2008)
[31] www.cometd.com (visited Jan. 2008)