# High-Level Design and Architecture of an HTTP-Based Infrastructure for Web Applications [1)]

Gustaf Neumann
Department of Information Systems
Vienna University of Economics and BA
Austria
*gustaf.neumann@uni-essen.de*

Uwe Zdun
Specification of Software Systems
University of Essen
Germany
*uwe.zdun@uni-essen.de*

[1)] Accepted for publication in the WWW Journal, February 2000.

**Abstract**

xoComm is a communication infrastructure for web applications based on the HTTP protocol. It provides an HTTP server and client access. Furthermore it is the basic communication service for the ActiWeb web object and mobile code system. The HTTP server component of xoComm is used to implement ActiWeb places. The places use the HTTP client access to provide the communication means for their agents. We present the design and architecture of xoComm on several crucial excerpts of the design. These are closely related to their implementation in the object-oriented scripting language XOTcl. We discuss how a dynamic and reflective environment, high-level language constructs, and concepts like design patterns influence the design and architecture.

**Keywords:** HTTP server/client, web object system, design patterns, dynamics, reflection, interception techniques.

## 1 INTRODUCTION

In the current practice the WWW is mostly used by exchanging HTML pages and associated files with the HTTP protocol and by displaying them in a web browser. This simple principle has provided a dominating position for internet-based information systems. Reasons are, that human beings can understand the presented information directly, that new information can be provided easily and that information pieces are easily connectable through links. But this simple architecture has several drawbacks for information system development: it lacks support for interactive or collaborative multi-user applications and is not really capable to exploit the benefits of distributed applications.

Two forms of interaction are usually used to meet such application's interaction requirements: Server-side (like the CGI interface, servlets) and client-side execution (like Java applets, scripts in diverse scripting languages). Both approaches lack the ability to exchange information directly, which may be gained through middleware approaches, like CORBA or Java RMI. This paper presents a simple and intuitive communication infrastructure, which achieves the benefits of today's web usage forms, does neither suffer from the stated problems nor from the complexity of middleware systems, and is based on the native means of the web.

Distribution can be achieved through remote procedure call (RPC) or remote programming (RP) [White 1995]. RPC implements distribution by calling procedures on a remote computer. Approaches, like CORBA or Java RMI, call methods of distributed objects instead of procedures, but the general principle is the same. RP (or code mobility) implements distribution via enabling not only to call but also to provide the procedure executed on a remote computer. RP's benefits are a higher performance, because of local execution of computations, and a higher configurability, flexibility and extensibility [Fuggetta *et al.* 1998], since the providing of code enables configuration of the server with behavior instead of just procedure arguments.

This paper describes the xoComm communication infrastructure and we will present its design and architecture. Our design decisions base on the assumption that the implementation is done in a dynamic and reflective environment. We will describe how this assumption influences the resulting object-oriented framework, the used design patterns, and the component concept. We will discuss various excerpts, which we consider as crucial spots for extensibility in detail. Finally we summarize the results in Section 6.

## 2 CONCEPTS AND BASIC ARCHITECTURE

In this section we discuss the underlying concepts of the design and the implementation. We emphasize benefits of object-orientation in the design of complex systems and discuss certain liabilities. We afterwards present some solutions in the concepts of the language XOTcl [Neumann and Zdun 2000b], which was used for implementation of xoComm. Further concepts, like pattern-based design, are sketched

afterwards. Finally the base-line architecture of ActiWeb is presented as a general application of xoComm, imposing several functional requirements, like support for local invocations, remote invocations via RPC, and migration/cloning via RP. We will present an infrastructure enabling flexible extension with such/other technologies and their requirements.

### 2.1 Object-Orientation and its Problems

Object-orientation is based on principles of information hiding and abstraction through encapsulation/specialization through inheritance. These principles mean a significant step towards reduction of complexity of software architectures. Object-orientation helps to decompose complex applications into manageable conceptual entities of the modeled world (objects of certain types). Objects are characterized through their behavior, but are structured around the data (i.e. their state). Design and implementation techniques derived from this simple, but powerful, concept should enable the minimization of development times, ease software maintenance, encourage and ease software reuse, and help to solve several other general problems. But many approaches have certain obstacles and limitations.

E.g. Hatton [Hatton 1998] points out that non-localities, as in the inheritance-/polymorphism-model in languages, like C++, do not match current models of the human brain very well. Non-localities break up the advantages of encapsulation that enables us to develop and study an object in isolation. Because of sole manipulation in the short term memory, the objects become easier understandable. Non-localities enforce manipulation in the long-term memory, making it more difficult to gain an overview and insight into a software architecture. In [Neumann and Zdun 1999b] we point out that language constructs on a high abstraction level and a unique string interface, instead of polymorphism, do avoid several such problems.

Most object-oriented languages and design approaches focus on single classes, which are able to describe the properties and the behavior of their instances in detail, but they normally do not entail powerful features to express how objects and classes are composed. Reflective techniques and a dynamic object/class system allow the objects and classes to flexibly adapt their interfaces to different clients requirements. An elementary approach to implement such techniques are meta-object protocols, as in [Kiczales *et al.* 1991]. They divide a system into a base-level and a meta-level for controlling the object-/class-system.

We propose the usage of *higher-level constructs* (such as design patterns) in the design and implementation of complex systems, which define the inter-relationships of components in order to provide an architectural view of the system. The programming language XOTcl provides language support for such higher-level constructs. Note, that we propose to use these constructs in the first design steps, even if there is no explicit language support for these constructs in the implementation language available. We emphasize that the architectural aspects should be explicit entities of the design language, since the used

notation determines the expressibility of the language. In a second step the design can be refined to a design implementable in the targeted language. Furthermore, often more high-level solutions can be brought into programming languages by hand, e.g. by enhancing the C language with a library that implements object-oriented concepts. Therefore we will use the language XOTcl in this paper rather as a design language than as an implementation language.

### 2.2 Pattern-Based Design

Complexity of software systems often makes the design decisions difficult. The main intent of design patterns [Gamma *et al.* 1994] is to preserve good design ideas. Object-oriented software design patterns describe situations in which several classes cooperate on a certain problem [Soukup 1995]. The concrete structure (as in an example implementation or in OMT diagrams) is not the main contribution of a pattern. In contrast, we think, patterns are mainly characterized by their intent, the forces/motivations of the design decision and the benefits/liabilities of the solution. A pattern can not be reduced to a (reusable) structure that can be customized solely through parameterization, because it has to fit into its context (design/implementation language, application domain, related patterns in the pattern language, etc.).

Since every context is different, making a pattern fit into a context takes hand-crafting in order to find the best possible design solution in the context. Several instantiations of one and the same pattern have (from a structural point of view) only their parts and their relationships at a fairly abstract level in common. Any sensible application of the pattern idea requires to customize the pattern to its context. Pattern implementations as in [Gamma *et al.* 1994; Buschmann *et al.* 1996] often suffer from problems due to the targeted language [Bosch 1998; Soukup 1995]. It is desirable to find pattern variants in other environments, that implement the pattern in a better way or even language support it. By language support we mean to program a common implementation of a pattern variant once and reuse it later by adapting the pattern to the new context. Common problems in pattern implementation are the *traceability* of the pattern [Soukup 1995], the *reusability* of a pattern implementation [Bosch 1998], the *implementation overhead* of trivial methods in several patterns [Bosch 1998], and the *self-problem* [Lieberman 1986], which denotes the loss of the self-reference when a message is forwarded to another object. Pattern-parts resemble roles (as in [Kristensen and Østerbye 1996]) rather than classes. Roles can be flexibly attached/detached to a single object. One and the same object can play a pattern role in several different patterns. In [Neumann and Zdun 1999c] and [Neumann and Zdun 1999b] we present the language constructs filter and per-object mixins (see Section 2.3) for implementation of pattern-roles on the class- and on the object-level.

### 2.3 The Object-Oriented Scripting Language Extended Object TCL (XOTCL)

The underlying language of our design is XOTCL [Neumann and Zdun 2000b] (pronounced *exotickle*), which is a value-added replacement of OTCL [Wetherall and Lindblad 1995]. In this section we describe its concepts briefly, because we consider them also as usable conceptual constructs (used at least in the first design phases, but in the ideal case throughout design and implementation). Both XOTCL and OTCL are object-oriented flavors of the scripting language TCL (Tool Command Language [Ousterhout 1990]). TCL offers a dynamic type system with automatic conversion, is extensible through components, and is equipped with read/write introspection. These functionalities ease the glueing process in a component framework.

In XOTCL every object is associated with a class over the `class` relationship. A class is a special object providing methods to create and destroy instances, and a repository of methods for its instances. Furthermore, a class provides a `superclass` relationship that supports single and multiple inheritance. All inter-object and inter-class relationships are fully dynamic and can be changed at arbitrary times. Since a class is a special (managing) kind of object it is managed itself by a special class called "meta-class". The XOTCL extensions focus on management of complexity/adaptability in large object-oriented systems:

- *Dynamic Object Aggregations* language-support the part-of relationship [Neumann and Zdun 2000a].
- *Nested Classes* reduce the interference of independently developed program structures.
- *Assertions* reduce the interface and the reliability problems caused by dynamic typing.
- *Meta-Data* enhance self-documentation of objects and classes.
- *Per-Object Mixins* are classes that are dynamically attached/detached object-specifically to an object. They intercept every message to the object and can handle the message before/after the original receiver. They are ordered in a chain and inherit from super-classes [Neumann and Zdun 1999a].
- *Filters* are special instance methods which are dynamically registered/deregistered for a class $C$. Every time an instance of $C$ or of any of its sub-classes receives a message, the filter is invoked automatically and intercepts this message. They are also chained and inherited [Neumann and Zdun 1999b].

Per-object mixins are a interception technique on the object-level, filters are an interception technique adapting all instances of a class hierarchy. In this paper we assume that the environment contains high-level language constructs of this/equal expression power (or that they are extracted from the design in a second step). E.g. some layers in LayOM [Bosch 1998] are of comparable expressiveness as certain filter applications, while roles, as in [Kristensen and Østerbye 1996], are comparable to per-object mixins.

### 2.4 Component Frameworks

Scripting languages, like Tcl [Ousterhout 1990] differ significantly from so-called system programming languages [Ousterhout 1998], like C, C++ or Java, where the whole system is developed in only one language. Scripting languages follow an approach, which distinguishes two levels: reusable components (written in various languages, like C, Tcl or XOTcl) and "glueing code" that combines components in the scripting language to a "component framework" [Ousterhout 1998].

We see a component as an entity, which provides services to its clients and which makes them accessible through the interfaces to the operations of the classes within the component framework. A component uses eventually other components to realize its services. Components are ordered in a directed acyclic "uses"-graph. Components declare the components that they use. Components are composable into larger components. Such components are suitable to build object-oriented structures on a level, that is higher than the level of classes. So-called "legacy"-components can be integrated and can achieve an object-oriented representation, e.g. using the wrapper facade pattern [Schmidt 1999], like the C components of xoComm.

### 2.5 Base-Line Architecture of ActiWeb

ActiWeb is a mobile code and an active web object system, which uses xoComm as its communication infrastructure. Basis for the architecture is XOTcl, which itself is a Tcl-compliant component written in C (see Section 2.3). On top of the XOTcl layer a set of basic services (also components) are implemented. Generally compatible components can be substituted, e.g. xoComm can be exchanged against another HTTP implementation (or e.g. through CORBA). In Figure 1 the base components are ordered in layers (as in the layers architectural pattern [Buschmann *et al.* 1996]) that defines the usage relation without drawing the edges. xoComm provides an object-oriented implementation of an HTTP server and HTTP access. All communication in ActiWeb relies on this service. Places, the basic execution environments of ActiWeb, contain exactly one HTTP server identified by host and port. ActiWeb objects (e.g. agents) have access to other (remote) ActiWeb objects via HTTP Access.

A metadata service provides an object-oriented implementation of an XML- (called xoXML) and a RDF-parser/-interpreter (called xoRDF). RDF-Metadata are used in ActiWeb as a unique data representation. xoStore is a general persistence service for XOTcl, which makes objects and their data transparently persistent. A registry service xoReg, enables registration of ActiWeb objects, e.g. to find an object through the specification of certain properties. xoMOS uses these services to implement a mobile object system. xoAWO makes web documents active objects (and programmable) and it gives agents various web representations, like HTML. On top, an application layer uses the whole system. This paper focuses on xoComm,

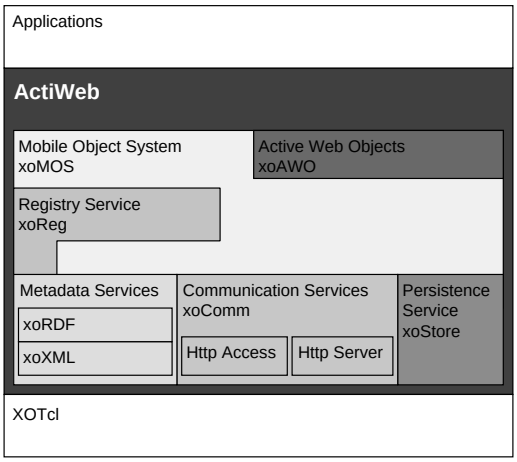


Figure 1: ActiWeb: Basic Architecture

but we use ActiWeb to show the connection of xoComm with other components. The sub-components of xoComm are depicted in Figure 2 and described in the later sections.

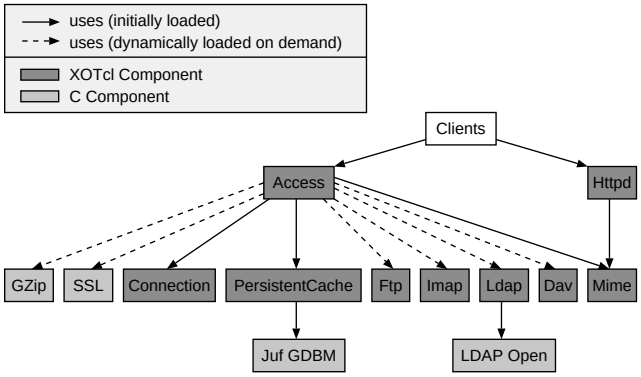


Figure 2: xoComm Component Architecture

## 3 HTTP ACCESS

The `Access` class provides client-side communication access. The class defines some basic parameters, like the `blocking` parameter which specifies whether a communication request is blocking of not. `url` specifies the used URL. Its simplified class definition is:

```
Class Access -parameters {{blocking 0} url ...}
```

### 3.1 Request Creation

In order to use the `Access` class, requests of various types, e.g. Ftp, Http, Files, etc., have to be created. A central place should control all creations, in order to easily trace request creations and to keep overview of all the various created requests (e.g. when a request is canceled, all dependent sub-requests have to be canceled as well). A factory method [Gamma *et al.* 1994] would serve these purposes, by abstracting the creation process into a creating method, which is specialized in sub-classes to concrete creation products.

But we have more requirements. A request can not always be classified once and then remains unchanged. E.g. if the request is redirected to another server via HTTP redirect, the request has to be classified again. In such a scenario the factory method solution requires that a new request of the new type has to be created. Then the old request object's data has to be copied and it has to be destroyed. In order to refer to the same entity of the real world (one and the same request) two objects have to be created. This is an inconsistency to the object-oriented paradigm due to the fact that in non-dynamical class-systems a dynamic change to the new request type is not possible. Furthermore, with dynamic classes components, handling additional request types, can be loaded on demand and new types can be easily and flexibly added.

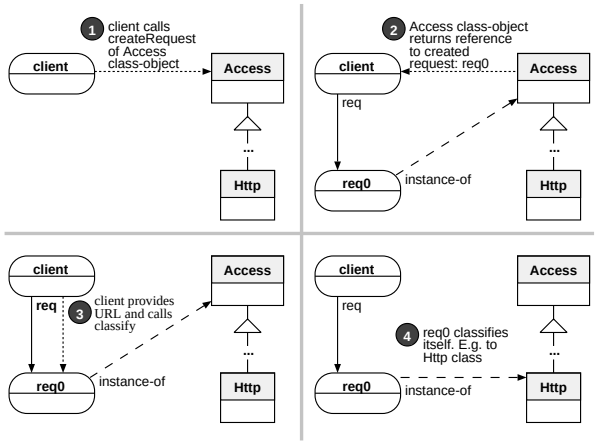


Figure 3: Class-Object-Specific Factory Method With Re-Classing from Access

For these reasons and since we implemented the design in a dynamic and object-specific language we used a *class-object-specific factory method* `createRequest`. This method is an object-specific method of the `Access` class-object. It initially creates a new request of the type `Access`. Afterwards it calls the `classify` method, which classifies the new request from `Access` to the proper sub-type by re-classing (see Figure 3). `classify` may be called at arbitrary times again to re-class from the sub-type to another sub-type, as in the HTTP redirect example, where an HTTP request might be re-classed to FTP. Here, we adapted the idea

of a factory method to the context of a dynamic language and to the application of request creation. An example of a access creation creates a new HTTP request, and informs the actual object (denoted by `self`):

```
Access createRequest -url http://www.somehost.com/ -informObject [self]
```

### 3.2 Flyweights for Request/Connection Sharing

The reuse and sharing of resources is an important aspect of an efficient implementation, what is also the intent the flyweight pattern [Gamma *et al.* 1994], where a pool of created resources is kept for later reuse. The pool can be changed dynamically at runtime. The reuse can be achieved by a centralized creation method that checks this pool before every creation.

One application of this pattern in xoComm is *request joining*. Often an HTML page contains several references, for example, to the same bullet image. For these images the HTML parser triggers multiple requests to the same resource. It is preferable to trigger only one request and to provide a mechanism for subsequent requests to participate from the results of this single transfer. All running requests are stored in a pool of requests, which is checked for a specified URL before creation. If a request for the same resource is running and joining is activated, the creation method returns the already running request. Otherwise a new request is created and returned. After the request has finished it is removed from the pool.

Another similar application of flyweights is the implementation of persistent connections, as defined in HTTP/1.1. A single connection to a host is shared by multiple requests. The HTTP commands are pipelined on this connection. A pool of open connections is maintained. The connections are held open until a timeout expires (or a server requests to close the connection explicitly). After closing, the connection leaves the flyweight pool. A third example is the request re-classing (as in the previous section), if it re-classes a request to a file request for the local cache, which forms also a pool of reusable requests.

### 3.3 Access Callback Interface

Using the `informObjects` method a client can specify a set of objects which are dependend of state changes of a specific request. A callback interface is implemented as an observer pattern [Gamma *et al.* 1994], where clients (observers) are notified using the push model [Buschmann *et al.* 1996] with five different types of notification. The request objects (subjects) store a list of references to their observers, specified through `informObjects`. The state changes are induced by corresponding network events.

A central method `doCallbacks` is called every time a network event occurs with the proper callback message as argument. The method forms a central hook for specializing callback invocations (as `notify` in the observer of [Gamma *et al.* 1994]). Every object of class `Access` informs the (possibly empty) list of `informObjects` during its lifecycle with the following abstract callback interface:

```
AccessCB abstract instproc startCb request    ;# Triggered when the request is created and its name is determined.
AccessCB abstract instproc notifyCb request   ;# Triggered when the content type of data is determined.
AccessCB abstract instproc incCb request      ;# Triggered when new data is available incrementally.
AccessCB abstract instproc endCb request      ;# Triggered when the request has ended successfully.
AccessCB abstract instproc cancelCb request   ;# Triggered when the request has ended not successfully.
```

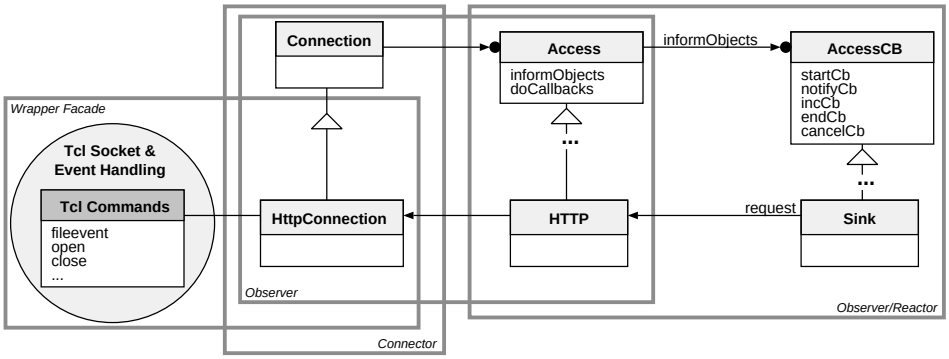


Figure 4: Callback Interface and Reactive Event Handling

Figure 4 shows the callback-based design. Access objects (the subjects) can be attached/detached by `informObjects`. The concrete subjects of the observer are the various specialized access types (see Section 3.4), in the figure: `HTTP`. All observers must implement the `AccessCB` interface, like the sinks in Section 3.6.

This special observer variant, that is able to handle various different (synchronized) event types, is also a client side variant of the reactor pattern [Schmidt *et al.* 2000]. The callbacks are event handlers that handle events by utilizing the synchronous event demultiplexing of the Tcl event system. The `Access` class has the role of a reactor that dispatches the handlers.

The `Connection` class wraps the Tcl commands for event handling and opening/closing of a socket connection. On the client side it is a connector in an acceptor/connector pattern [Schmidt *et al.* 2000], that creates a request on a specified port. The class is a wrapper facade [Schmidt 1999] that gives the Tcl commands a consistent object-oriented interface. The connection itself is observed by the `Access` classes in a second observer pattern.

### 3.4 Special Access Types

The access classes provide communication access for several web-based applications, like the ActiWeb system and the extensible web browser Cineast [Köppen *et al.* 1997]. Several different types of access are necessary to satisfy these applications' communication needs. All sub-classes of the `Access` class, to which the requests are flexibly classed (as described in Section 3.1) are presented in Figure 5.

A file access class is base class for all forms of access to the local disk. A request is classed to its

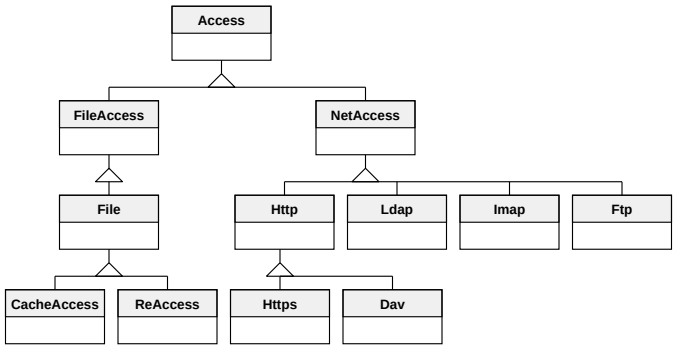


Figure 5: Access Class Hierarchy

sub-class `File` if the specified URL starts with `file://`. `CacheAccess` is used, if the caching is turned on and a URL is validated by the persistent cache. For revisitation of an URL (e.g. using the back button of the browser) the `ReAccess` class is used. The class `NetAccess` is base class for various net accesses. Its subclass `Http` implements the client side of HTTP/1.0 or /1.1 (see Section 3.5). A sub-class `Https` implements a secure HTTP Access. It encapsulates a C component with object-oriented methods for a secure socket layer (SSL), which is a shared library and is dynamically loaded on demand. All these classes are part of the Access component. There a four components which implement further `Access` sub-classes. The Dav component implements a distributed authoring and versioning protocol (WebDav). The Ftp component implements the file transfer protocol. The Imap component implements the IMAP protocol for access to mail servers. The Ldap component implements the LDAP protocol for accessing online directory services.

### 3.5 HTTP Access Type

The `Http` class implements the client side of the HTTP protocol. The additional interface parameter methods for creating requests of HTTP type are summarized in Table 1. The different HTTP methods are handled by same-named operations of the `Http` class, like GET, POST, PUT, HEAD, etc. All these HTTP method handlers call a central (hook) operation `open` which is responsible for establishing a connection. Additions, like proxy filters or SSL hooks, can easily be placed into this method without a necessity to change all HTTP method handlers. The HTTP method handlers form templates for the algorithm how they invoke `open`. The `open` operation is a standard implementation, which can be extended in a sub-class or through an interceptor, like filters or per-object mixins. The HTTP method handlers can be seen as template methods [Gamma *et al.* 1994] calling a standard implementation per default (hook operation in [Gamma *et al.* 1994]) making both operations exchangeable and extensible.

The process of handling a request is not handled by the `open` method alone. For non-blocking requests, different event handlers must be registered to handle the incoming data accordingly. These event handler

| | |
|---|---|
| `method` | Specifies the HTTP method, like GET, PUT, POST, which is used. |
| `httpVersion` | Specifies the HTTP version, which is used for a request. Default is 1.1. |
| `contentType` | Specifies the MIME type of the content of the sent data. |
| `data` | Specifies the data for PUT. |

Table 1: The Interface of the HTTP Class

methods are called repetitiously until the state of the request changes. Then the next event handler is registered using `event` method of the `Connection` class. This event processing (see Figure 6) is started in the `open` method, which succeeds in a `writable` connection. First the HTTP server is asked if it accepts the connection. If no error occurs the next `readable` event triggers that the start of the header is checked by `headerStart`. Afterwards the `header` method determines the next actions according to the header content. E.g. if the transfer is encoded a handling for decoding is initiated. Normally the next readable event is the method `readData` which reads the data until it can finish the connection.

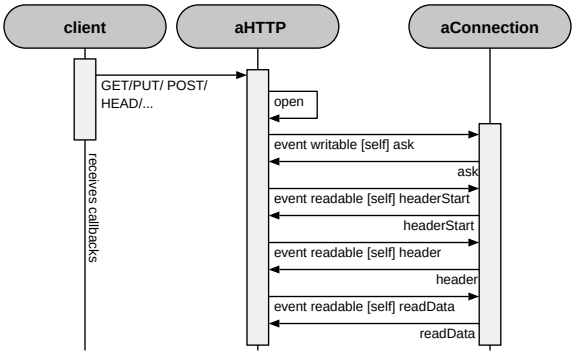


Figure 6: States Changes of Event Processing

### 3.6 Sinks

In order to receive data from requests generated by the `Access` class, clients must implement the `AccessCB` interface. In several situations a general means to receive a data-stream incrementally is needed. In order to make data-stream handling flexibly extensible and changeable, we derive a general sink class from `AccessCB`. Sink objects decouple the data-stream handling from the requests.

The whole-part pattern [Buschmann *et al.* 1996] is a special form of aggregation, where clients see the aggregating object (the "whole") as an opaque object. The aggregated parts are of arbitrary type and are only accessed through the whole. In XOTcl the whole-part pattern is language supported through dynamic object aggregations [Neumann and Zdun 2000a]. The presented sinks are aggregated by their `Access` object

and are only accessed through the `Access` object. They are parts in a whole-part pattern. We extend the general sink with several different sinks for special data-stream handling as in Figure 7.

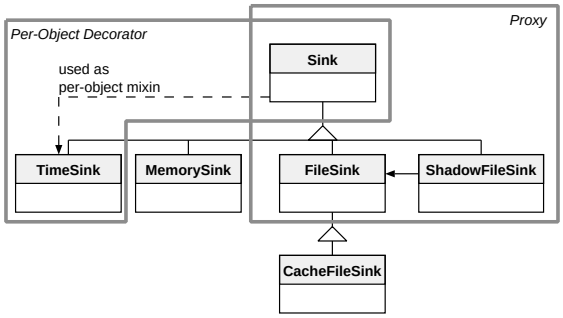


Figure 7: Sink Classes

The `MemorySink` just writes every incoming data into the memory. The `FileSink` lets the client specify a file name and if the file name is specified the data is written into that file. If the sink learns after its instantiation, that the client has not provided a filename, it must change its behavior completely. Again, this is a case for dynamic class relationship. If the first data arrives and the file name is absent, the instance can not act as a file sink and changes its type to `ShadowFileSink`. This class lets the sink act as a proxy [Gamma *et al.* 1994] that delegates its work to some other file sink (see Figure 8). Note, that in contrast to a role, which adds additional extrinsic properties to an object, this change affects the intrinsic properties of the object. Re-classing is used to express these new intrinsic properties. A special file sink is the `CacheFileSink` which makes an entry in the persistent cache entry (for some of its metadata such as the content type, modification date, etc) in addition to the writing of the file.

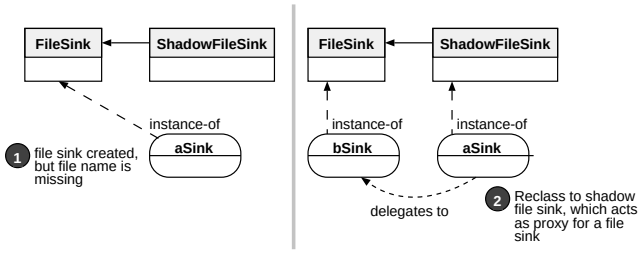


Figure 8: Re-Classing of a File Sink to a Shadow File Sink

The `TimeSink` class measures time periods of receiving data for a particular sink. It is a so-called supplemental class (see [Neumann and Zdun 1999a]) which can be added to every sink class. A delegation to a timer would not represent the timed sink as one entity of the design/implementation. (Multiple) inheritance requires to derive a set of unnecessary intersection classes, one for each combination of a sink with the supplemental class `TimeSink` and makes dynamical adding/removing of timing hard to accomplish.

A design using per-object mixins is a better solution. It adds timing like a role or extrinsic property of the sink object, does entail decomposition into classes, but does not split the conceptual entity into two objects of the design/implementation. We use a per-object decorator pattern [Neumann and Zdun 1999c], since the time sink decorates other sinks with timing. In the usual case sinks are instantiated and if needed the instances get the per-object mixin `TimeSink` dynamically attached/dettached to obtain timed sinks. E.g. a `MemorySink` instance `sink1` acquires the timing property after creation (see Figure 9):

```
MemorySink sink1 -mixin TimeSink
```

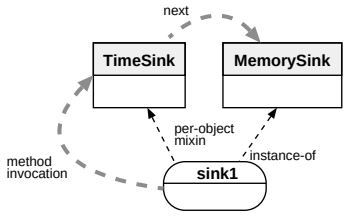


Figure 9: Time Sink as Mixin

## 4 HTTP SERVER

In this section we describe the extensible design of an HTTP/1.1 compliant HTTP server in the `Httpd` class. The class has a set of parameters, like `port`, `root` directory, and logging directory. Furthermore a name of a class can be specified which is an HTTP worker. Essentially, the class `Httpd` configures the server instance and listens on the specified port, while the worker instances handle the incoming requests in an asynchronous fashion. The constructor of `Httpd` creates the socket an starts logging, while the destructor `destroy` stops listening, destroys the socket and terminates logging.

```
Class Httpd -parameters {{port 80} {root /} {logdir ~/.log} {httpdWrk Httpd::Wrk}}
Httpd instproc init args {...}
Httpd instproc destroy args {...}
Httpd instproc accept {socket ipaddr port} {...}
```

The HTTP server provides the client with a MIME content type of resources, which it has to be guessed from several indicators (such as file extensions) of the files in its document pool. A class `MimeTypes` of the Mime component handles this task. In order to allow clients to customize the mime type guessing transparently, we can define a per-object decorator [Neumann and Zdun 1999c] `MimeTypeLoader` which reads in a user-defined customization file:

```
MimeTypes Mime -mixin MimeTypeLoader
```

### 4.1 Accepting Connections

The method `accept` accepts a new connection and sets up a new worker. It instantiates a worker with an automatically generated name as aggregate of the server object. This worker object is of the class specified by the reference stored in the parameter `httpdWrk`. By changing this parameter, the worker of any connection can be changed at runtime to another worker class. `accept` is a method that constructs new worker objects, but it is extensible for clients that can specify the used worker type.

The intent of this design is the same as in the factory method pattern [Gamma *et al.* 1994]. But the factory method achieves extensibility through sub-classing. The used sub-class decides which class is implemented and afterwards used. In languages in which classes are also objects (and can be referenced), the solution of a *class-referencing factory method*, where a parameter specifies the concrete product, which is instantiated, is a superior variant of the pattern, because the sub-classes are solely used to specify the concrete product type. This means they are an unnecessary implementation overhead.

### 4.2 HTTP Worker

The necessary excerpt for the abstract worker interface to respond to HTTP requests is the following:

```
Class Worker
Worker abstract instproc header {}
Worker abstract instproc receive-body {}
Worker abstract instproc respond {}
Worker abstract instproc respond-GET {}
Worker abstract instproc respond-PUT {}
Worker abstract instproc respond-POST {}
```

`header` reads and evaluates the header. `receive-body` reads and evaluates the body of a request. `respond` generally responds to a request. In the simplest implementation it just calls the proper respond method according to the desired HTTP-method, i.e. GET, PUT, POST, etc. method handling. An inner class of the class `Httpd`, called `Httpd::Wrk` implements all these methods with a standard implementation according to HTTP/1.0 and /1.1 (see Figure 10). Socket, port and IP address are specifiable as parameters.

```
Class Httpd::Wrk -parameters {socket port ip}
```

The workers are aggregated by the server using the whole-part pattern [Buschmann *et al.* 1996]. They are event handlers in a similar reactive event dispatching mechanism as on the client side. Here, we also use a reactor [Schmidt *et al.* 2000] to dispatch the connection events, but the `Connection` class dispatches the event handler itself. Since the connection awaits incoming requests on the server side, the connection gets

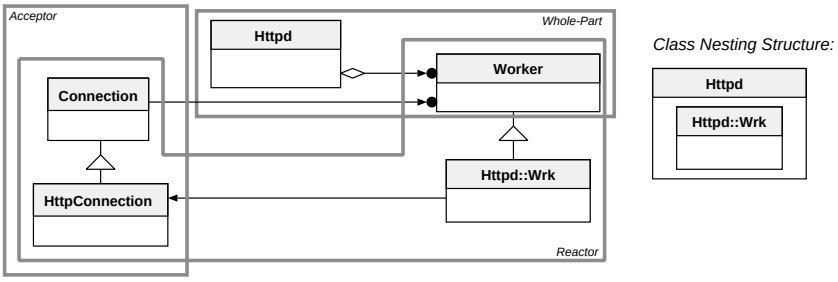


Figure 10: HTTP Worker

the additional role of being an acceptor in an acceptor/connector pattern [Schmidt *et al.* 2000], that listens on the port and establishes a socket connection for incoming requests.

### 4.3 Access Control and Authentication

The definition of HTTP/1.1 [Fielding *et al.* 1999] contains some means for access control of web pages, called basic authentication scheme. This simple challenge-response authentication mechanism lets the server challenge a client request and clients can provide authentication information. When the client requests a protected resource (without the necessary credentials) the server may respond with a 401 (Unauthorized) response containing the name of a realm defining the users which are authorized to access it with the specified method. The client queries the user for his name and password. If it is provided, the client resubmits the request with the user name and password encoded in the credentials (using the base64 encoding). The server receives the request again, obtains the user name and password from the credentials, and compares it to the set of user name and password pairs denoted by the realm. On success the access to the resource is granted.

The basic authentication scheme is not considered to be a secure method of user authentication, since the user name and password are passed over the network in an unencrypted form. Digest Access Authentication, defined in RFC 2617 [Franks *et al.* 1999], provides another challenge-response scheme, that does never send the password unencrypted, which is the most serious flaw of basic authentication. The following architecture for the presented server enables usage of both schemes and is flexibly adaptable to new schemes.

The `respond` method of each worker should be (transparently) enhanced with an access control decoration if needed. Each decorator may be applied only once, but several different kinds of decorators are composable, e.g. some resources may be protected by basic and some by digest access control at the same time. We implement the decorators as per-object decorators (using per-object mixins as in [Neumann and Zdun 1999c]). These are ordered in a chain of responsibility [Gamma *et al.* 1994] on the server and on the worker. The first decorator that matches the resource is used. The standard `respond` method is the fallback

method when the resource is not protected. Following the architecture of the `Httpd` class we distinguish for the extension between per-server concerns (such as configuration of the `Httpd` instance) and of per-request concerns (the workers). The per-server concerns of access control are defined in the class `AccessControl`:

```
Class AccessControl
AccessControl abstract instproc protectedResource {fn method varAuthMethod varRealm}
AccessControl abstract instproc credentialsNotOk {credentials authMethod realm}
AccessControl abstract instproc addRealmFile {realm authFile}
AccessControl abstract instproc addRealmEntry {realm passwds}
AccessControl abstract instproc protectDir {realm path methods}
```

The various decorators, like `BasicAccessControl`, are mixed-in and managed by the configuration part of the server. The methods for defining the protected resources and the realms are added to the interface (such as `protectedResource` etc.).

The server instance maintains a list of extensions `workerMixins` which is used to refer to the mixin-classes for the worker which essentially overload the standard instproc `respond`. When basic access control is configured it registers its worker to the list of worker mixins.

```
Class BasicAccessControl -superclass AccessControl
BasicAccessControl instproc init args {
  next
  lappend [self]::workerMixins [self class]::Wrk
}
```

The instance method `respond` of the decorator-specific worker intercepts the message and determines if the decorator matches or not.

```
Class AccessControl::Wrk
Class BasicAccessControl -superclass AccessControl
BasicAccessControl instproc respond {} {...}
```

When the mixin intercepts the respond call it checks via the instproc `protectedResource` whether it should check the credentials. If these are not ok (checked by the method `credentialsNotOk`), the reply code is set to 401 and `respond` returns directly. Otherwise the next decorator is checked, until the respond method of the worker is reached.

The worker mixins are registered automatically for each incoming request in the class-referencing factory method `accept` of `Httpd`. This central hook for creation has made it easy to provide the additional functionality on all workers dynamically and transparently. Both mixin types and order are dynamically changeable (see Figure 11). E.g. the following protects the root dir of a server `httpd1` with the realm `users`.

```
Httpd httpd1 -mixin BasicAccessControl -addRealmEntry users {guest pw1} -protectDir users "" {}
```

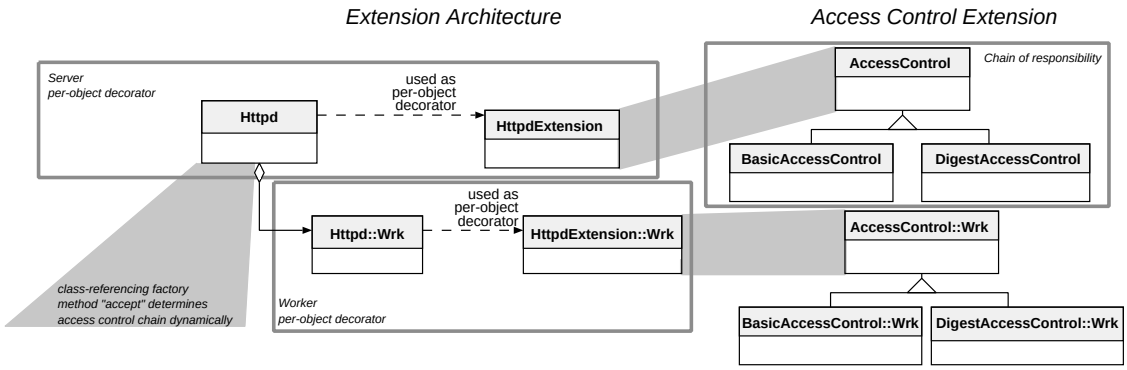


Figure 11: Access Control Architecture

Note that access control (and other extensions of the server) are completely configurable. The basic server has absolutely no knowledge about the extensions, the code is not cluttered by any if-then-else or similar constructs that check the existence of extensions. This improves readability, locality and performance. Figure 11 shows the general extension architecture for HTTP place/worker and how access control is attached.

## 5 HTTP PLACE

This section describes the architecture of the HTTP place of ActiWeb and how it uses the xoComm infrastructure. A central property of places is that on every port on a host no or exactly one place is located in order to give places a unique identification through host and port number. In each ActiWeb process exactly one place is running. This constraint is ensured by a specializable singleton pattern [Gamma *et al.* 1994]. This pattern asserts that from a class one or no instance is derived. It also provides a central point to access this instance. In XOTcl the singleton pattern is language supported.

```
Singleton Place -superclass Invoker
```

A place has also the role of a whole in a whole-part pattern [Buschmann *et al.* 1996]. The place aggregates objects for agent management, RDF creation, central registration, error management, script creation, and persistence. Furthermore the place aggregates an HTTP server, which gets a nested class `Place::HttpdWrk` as its associated HTTP Worker. This way the place can intercept all incoming HTTP messages and redirect them as object-oriented calls. In order to make object identification unique in the web, we use the web standard URL to identify objects, which are encoded/decoded using URL encoding, described in RFC 1738 [Berners-Lee *et al.* 1994], by a `callCoder` object which is part of the place as well.

### 5.1 Invoker

The worker must transform HTTP messages into invocations of object-oriented calls. Normally it delegates to the place singleton in order to let the place do the actual invocation. The place inherits from the abstract interface `AbstractInvoker` through its super-class `Invoker`:

```
Class AbstractInvoker
AbstractInvoker abstract instproc invoke {obj method arguments}
AbstractInvoker abstract instproc eval {obj method arguments}
AbstractInvoker abstract instproc callError {msg obj args}
Class Invoker -superclass AbstractInvoker -parameters {{place [self]}}
```

The `Invoker` defines a default invocation behavior in `invoke` (which especially checks if the object is exported object of the place and if the object allows HTTP messages to call the particular method). If the invocation is valid, it is executed by `eval`, otherwise an error is created, using the aggregated error manager. Again the central point of dispatch for invocations allows us to change invocation processes easily.

Since places are singletons new places can not be instantiated just to provide a new invocation variant. But often several variants have to exist in parallel. E.g. various user interfaces require a proxy [Gamma *et al.* 1994] that delegates to the proper user interface, as explained in [Zdun 1999]. Another example is a secure invoker, which has only to be used for calls from foreign networks. In such cases, a direct instance of the special invoker class may be instantiated and the `place` parameter (which is a self-reference in the default case, that the `Invoker` is used as the place's super-class) can be set to the place singleton. Then this new invoker object can be used for special invocations parallely to the place singleton. The additional invoker can also be attached using the same extension scheme as for access control (see Figure 11).

### 5.2 HTTP Worker of the Place

The special HTTP worker of the place is a nested class, defined inside of the place class. Its superclass is the worker class defined inside of the `Httpd` class, because the worker should act as a normal HTTP worker, except that it intercepts all calls to the GET and the PUT method.

```
Class Place::HttpdWrk -superclass Httpd::Wrk
Place::HttpdWrk instproc parseParams {o m a call} {...}
Place::HttpdWrk instproc respond-GET {} {...}
Place::HttpdWrk instproc respond-PUT {} {...}
```

In the overloaded `respond-GET` method an RPC mechanism is implemented on top of the HTTP GET method. `parseParams` determines if the call is a valid object-oriented call and extracts the called object, method, and arguments. The place singleton's method `invoke` is called with the extracted information and

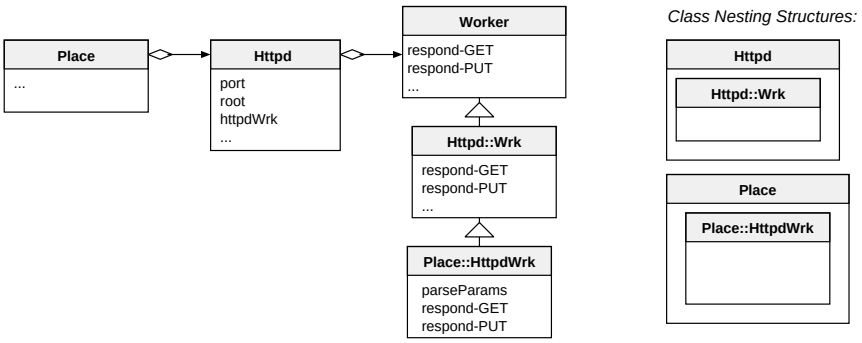


Figure 12: Connection to the Place

the invocations result is sent back. The method `respond-PUT` functions quite similar, but the HTTP PUT method is used to implement an RP mechanism. Agents can be sent as PUT-data (encoded in RDF) in order to migrate and clone themselves to other places. The resulting architecture is presented in Figure 12.

## 6 OVERALL FRAMEWORK ARCHITECTURE

In the preceding sections we discussed a flexible, adaptive, and dynamic design and architecture of xoComm. We have described (in many details) crucial part of the design and have illustrated our particular solutions using the concepts and techniques discussed in Section 2. In this section we try to bring the results together. Figure 13 combines the various presented (base) classes to an static, architectural overview.

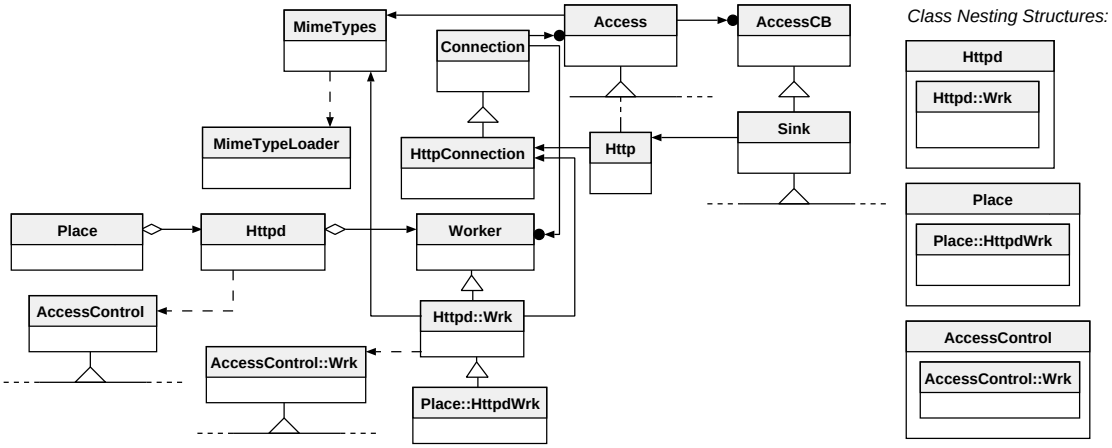


Figure 13: Architecture Overview

xoComm is an object-oriented framework and was presented as a case study. We used many design patterns and pattern variants (summarized in Table 2 in order of appearance) to explain our design decisions. Since xoComm is in use and under development for a long time it was subject to considerable evolution. New

requirements, like changing web standards or the usage in ActiWeb, are constantly appearing. The design and the architecture have to cope with these changes. Finding central places in the architecture, where hooks for extensions may be flexibly and transparently placed, is a central aspect of designing software systems. The aim is to cope with changes even before one can foresee them. One way to reach that aim is to centralize parts of the architecture into hook methods or objects, that are known from the experience as constantly changing, like creation processes, event processing, or accesses to shared resources.

Unfortunately one can not predict all changing parts, therefore, a division into stable and unstable parts of the architecture can always be only preliminary. In order to even *cope with unexpected changes*, reflection, dynamics, and powerful means of adaptation (like filters or per-object mixins) have proved to provide great benefits. Another reason for adaptation facilities is *software evolution transparency*, e.g. through automatically redirecting calls for implementations of new client requirements to new components.

*Clients should not have to interfere with used (third-party) components.* Often when a new requirement, that is a pure addition, is introduced, conventional designs would place the addition in the extended class itself or would delegate to an object handling the addition. In both solutions the extended class would have to be changed, in order to bring in the new functionality. High-level language construct and concepts for adaptations, let an object be (transparently) enhanced with extensions, without interference with (third-party) components. If a client is even not capable of changing a component to the requirements, client-side adaptation is even more useful for *easy component combination*.

Usage of open standards means to *cope with the evolution of open standards*. An open standard lets software systems cooperate with other systems compliant to the standard. Open systems using a standard have to implement changes in the standard as soon as possible. ActiWeb bases on open web standards, like HTTP as communication protocol, URL encoding, URLs for distributed object identification, XML/RDF for knowledge exchange. A central reason for using web standards is to make applications heterogeneous. E.g. ActiWeb can communicate with any other system directly, that is written in a language in which an HTTP access and/or server is implemented.

Management of several versions of components in a software system (or a whole product-line architecture) is a complex and difficult task. *Avoiding superfluous versions* eases software maintenance, i.e. through placing an addition in a dynamically addable/removable component rather than producing one version with and one without the addition.

Our design was founded on the assumption that the described high-level constructs/concepts are provided by the targeted programming language. We have shown that the concepts have a considerable impact on the design/architecture, but always a similar conventional implementation was at hand, which could be extracted from the design (eventually even by tool support). Nevertheless, the programming language and

the design method should be of comparable expression power and both should use the most expressive means (instead of reducing programming language and design method to their common denominator).

## 7 PERFORMANCE

Efficiency was for long time an argument against languages with a dynamic type system. Nowadays CPUs are fast enough to execute even complex applications with sufficient speed [Ousterhout 1998]. We can confirm these results with several applications, including our xoComm-HTTP server. We compared in detail the xoComm-HTTP server (OO scripting language, design pattern based implementation, single process) with Apache 1.3.6 (implemented in C, architecture based on multiple worker processes). In our benchmark configuration we tested the simultanoues usage of the server by 1 to 20 client sessions. Each client session was running either on a separate Intel PPro machine with 200MHz and a 10MBit Ethernet, or on the same machine as the server (local client configuration). For the server hardware we used machines with 100MHz Ethernet cards (one single processor Intel Celeron PC with 466MHz, and one dual PPro machine with 200MHz). The operating system of all machines is Linux with a 2.2.5 kernel. Each client session consists of 16 HTTP/1.0 GET requests ($6 \times 0.5KB$, $7 \times 5KB$, $1 \times 50KB$, $1 \times 500KB$, $1 \times 5MB$) and 60 HTTP/1.1 GET requests ($10 \times 50KB$, $50 \times 5KB$), which totals to about 6.5 MB of data (verified by the client).

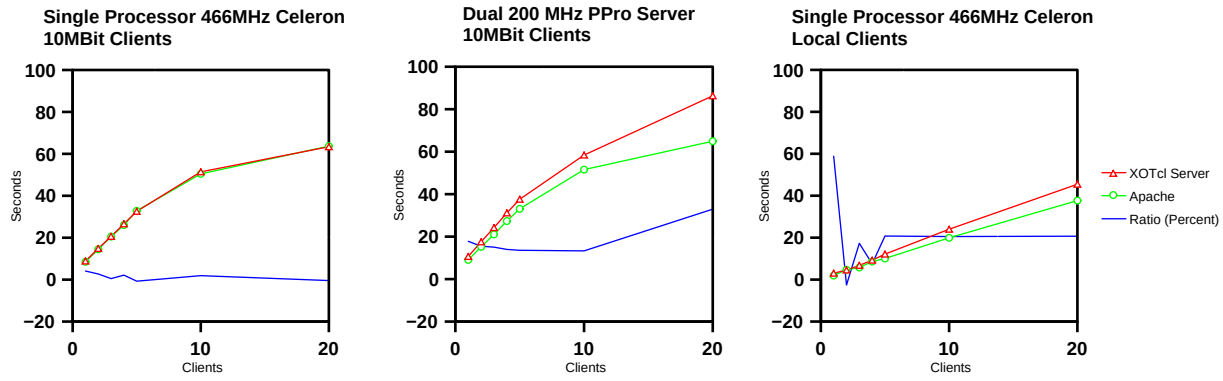


Figure 14: Comparison of the xoComm-HTTP Server vs. Apache 1.3.6

Our results (Figure 14) show that both the XOTcl web server and Apache can easy saturate an 10MB local network on an off-the-shelf PC (466MHz Celeron). The bottle-neck in this configuration is the 10Mb Ethernet (see left graphic), the 20 clients transfer a total of $\sim$ 130MB in $\sim$ 63 seconds ($20 \times 76 = 1520$ requests). The lines of both servers are practically identical.[1)]

The middle chart shows that on CPU of the slower 200MHz PPro cannot fully saturate the 10MB

[1)] We verified these results with a different setup, where the 5MB request was removed and each client sessions transfers 75 requests instead (a total of $\sim$ 1.5MB of data). The performance difference in this setup is again on average less the 4%. In total $\sim$ 30MB are transfered by the 20 clients in total in $\sim$ 16 seconds ($20 \times 76 = 1520$ requests, 95 requests/sec).

network, and that the architecture of Apache with multiple worker processes running on different CPUs leads to an performance advantage of up to 30%. Note, that the total time of the 20 clients is similar ($\sim 65$ seconds) to the single CPU machine with the faster processor (bounded by the network bandwidth).

The right chart shows the server comparison running in a non-networked configuration, where clients and servers are running on the same machine. Here we get the maximum inter-process communication bandwidth, and we obtain more information about the speed penalty of the implementation in the scripting language. We see in this benchmark that typically the performance difference between Apache and the xoComm-HTTP server is in the range of 5 to 20 percent.

## 8 RELATED WORK

WebBroker [Tigue and Lavinder 1998] also uses the web standard HTTP for communication. It is, in contrast to our approach, interface-based. Similar to CORBA, it needs proxy (stub) and skeleton object in order to communicate (solely) through RPC, with all its disadvantages (see Section 1) in comparison with RP. The benefits of the approach are the unique addressing through URLs, the object-orientation (with XML encoding), the simplicity in contrast to approaches like CORBA, and the platform independence. All these advantages are also valid for our approach.

WIDL [Wales 1999] is another approach using web standards for RPC communication. WIDL objects use XML for data representation and can be accessed via URLs from any platform with an HTTP implementation. Benefits, like efficiency, language-/platform-independability, and the ability to exchange arbitrarily complex data structures, are valid for this (and our) approach. It can not exploit the benefits of RP.

ZOPE [Latteier 1999] is an object-oriented development environment for web sites, based on the scripting language Python. It also contains a web server and an object-oriented data-base, like ActiWeb. All mentioned approaches have several similarities to ActiWeb, but they lack components, like a mobile code system, a registry, or a unique data representation. In xoComm we provided many hooks to enable flexible adding of such components. The overall goal is to provide an extensible infrastructure for *one* general framework for the development of distributed, information-oriented applications (as in [Kotz and Gray 1999]).

Telescript [White 1995] is an object-oriented programming language, which pioneers in the area of mobile code. Several concepts, like agents and places are very similarly used in our approach. D'Agent (formerly Agent Tcl) [Gray 1996] is a mobile agent system, which supports several programming languages, like Tcl, Java, and Scheme. These, and several other mobile code approaches, are not founded on open web standards (and are therefore not an ideal architecture for web applications).

In [Dömel 1996] agents produce HTML output through a document generator and are therefore enabled for web access. This approach only solves the partial problem of web representations of agents. But interestingly, with Telescript mixins a micro-architectural approach, similar to per-object mixins, is used to bring in the additional functionality. In [Lingnau *et al.* 1995] an HTTP-based infrastructure for mobile agents is described. It also consists of a specialized HTTP server and language-specific extensions (and therefore also enables heterogeneity of the programming language). It only concentrates on this partial problem of a general communication infrastructure, but in this part the results are similar to our work.

The reference architecture, described in [Ciancarini *et al.* 1998] also gains web access through special agents, translating application results to a web representation. The main difference to our approach is that is uses a coordination technology, here Pagespace, to manage interaction of (distributed) agents. Tuples are written into a coordination environment (tuple-space). This tuple-space can be read by all participants. The tuple-space approach has the advantage of heterogeneity as well, since it enables coordination independent of the programming language. We tried to build an architecture that does not impose the coordination technologies, is open for several different technologies, and achieves heterogeneity through web standards.

## 9 CONCLUSION

In this paper we have presented a general communication architecture as a case study for an extensible object-oriented framework. We have discussed various excerpts of the architecture which we consider as crucial spots for extensibility in detail and explained the design decisions on basis of the concrete domain problem (an HTTP access/server). Our architecture is founded on usage of open standards, on constructs and concepts, like design patterns, extensibility hooks, components, etc., but also on the assumption that a dynamic, reflective, and adaptive environment is available in design and implementation. The design has used constructs and concepts of the language XOTcl, though constructs of equal expression power could have been used as well. For implementation in another language, the design can be refined to a more complex design in a second step. On the example of the ActiWeb places we have shown how to connect other components, that have requirements intruding into xoComm's scope, to xoComm. Since xoComm has not changed in order to cope with ActiWeb's additional requirements, we consider the ease of the extensions, of connecting components, and of adaptations as a proof for our opinion that the above described constructs and concepts are good companions on the way to more understandable and more extensible designs and architectures. XOTcl and xoComm are available from: `http://nestroy.wi-inf.uni-essen.de/xotcl`.

**References**


Berners-Lee, T., L. Masinter, and M. McCahill (1994), "Uniform Resource Locators (URL)," RFC 1738.

Bosch, J. (1998), "Design Patterns as Language Constructs," *Journal of Object Oriented Programming 11*, 2, 18–32.

Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal (1996), *Pattern-oriented Software Architecture - A System of Patterns*, J. Wiley and Sons Ltd.

Ciancarini, P., R. Tolksdorf, F. Vitali, D. Rossi, and A. Knoche (1998), "Coordination Multiagent Applications on the WWW: A Reference Architecture," *IEEE Transactions on Software Engineering 24*, 5.

Dömel, P. (1996), "Mobile Telescript Agents and the Web," In *Proceedings of COMPCON'96*, Santa Clara, California, USA.

Fielding, R., J. Gettys, J. Mogul, H. Frysyk, L. Masinter, P. Leach, and T. Berners-Lee (1999), "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616.

Franks, J., P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart (1999), "HTTP Authentication: Basic and Digest Access Authentication," RFC 2617.

Fuggetta, A., G. P. Picco, and G. Vigna (1998), "Understanding Code Mobility," *IEEE Transactions on Software Engineering 24*, 5.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Gray, R. S. (1996), "Agent Tcl: A flexible and secure mobile-agent system," In *Proceedings of the 4th Tcl/Tk Workshop*, Monterey, California, USA.

Hatton, L. (1998), "Does OO Sync with How We Think?" *IEEE Software 15*, 3, 46–54.

Kiczales, G., J. des Rivieres, and D. Bobrow (1991), *The Art of the Metaobject Protocol*, MIT Press.

Köppen, E., G. Neumann, and S. Nusser (1997), "Cineast – An Extensible Web Browser," In *Proceedings of the WebNet 1997 World Conference on WWW, Internet and Intranet*, Toronto, Canada.

Kotz, D. and R. S. Gray (1999), "Mobile agents and the Future of the Internet," *ACM Operating Systems Review 33*, 3.

Kristensen, B. B. and K. Østerbye (1996), "Roles: Conceptual Abstraction Theory & Practical Language Issues," *Theory and Practice of Object Systems 2*, 143–160.

Latteier, A. (1999), "The Insider's Guide to Zope: An Open Source, Object-Based Web Application Platform. Learning the Language," *Web Review 3*, 5.

Lieberman, H. (1986), "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems," In *Proceedings of OOPSLA '86*, Portland, USA.

Lingnau, A., O. Drobnik, and P. Dömel (1995), "An HTTP-based Infrastructure for Mobile Agents," In *Proceedings of 4th World-Wide Web Conference.*

Neumann, G. and U. Zdun (1999a), "Enhancing Object-Based System Composition through Per-Object Mixins," In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC)*, Takamatsu, Japan.

Neumann, G. and U. Zdun (1999b), "Filters as a Language Support for Design Patterns in Object-Oriented Scripting Languages," In *Proceedings of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems*, San Diego, California, USA.

Neumann, G. and U. Zdun (1999c), "Implementing Object-Specific Design Patterns Using Per-Object Mixins," In *Proceedings of NOSA'99, Second Nordic Workshop on Software Architecture*, Ronneby, Sweden.

Neumann, G. and U. Zdun (2000a), "Towards the Usage of Dynamic Object Aggregation as a Foundation for Composition," In *Proceedings of Symposium of Applied Computing (SAC'00)*, Como, Italy.

Neumann, G. and U. Zdun (2000b), "XOTcl, an Object-Oriented Scripting Language," In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA.

Ousterhout, J. K. (1990), "Tcl: An embeddable Command Language," In *Proceedings of the 1990 Winter USENIX Conference*, Washington, D.C., USA.

Ousterhout, J. K. (1998), "Scripting: Higher Level Programming for the 21st Century," *IEEE Computer 31*, 3, 23–30.

Schmidt, D. C. (1999), "Wrapper Facade: A Structural Pattern for Encapsulating Functions within Classes," *C++ Report, SIGS 11*, 2.

Schmidt, D. C., M. Stal, H. Rohnert, and F. Buschmann (2000), *Patterns for Concurrent and Distributed Objects*, Pattern-Oriented Software Architecture, J. Wiley and Sons Ltd.

Soukup, J. (1995), "Implementing Patterns," In *Pattern Languages of Program Design*, J.O.Coplien and D. Schmidt, Eds., Addison-Wesley, pp. 395–412.

Tigue, J. and J. Lavinder (1998), "WebBroker: Distributed Object Communication on the Web," http://www.w3.org/TR/1998/NOTE-webbroker.

Wales, M. G. (1999), "WIDL: Interface Definition for the Web," *IEEE Internet Computing 3*, 1, 55–59.

Wetherall, D. and C. J. Lindblad (1995), "Extending Tcl for Dynamic Object-Oriented Programming," In *Proceedings of the Tcl/Tk Workshop '95*, Toronto.

White, J. (1995), "Mobile Agents White Paper," http://www.genmagic.com/technology/techwhitepaper.html, General Magic, Inc.

Zdun, U. (1999), "Entwurf und Entwicklung eines mobilen Objekt-Systems für Anwendungen im Internet," Diplomarbeit (diploma thesis), Universität Gesamthochschule Essen.

| *Pattern* | *Pattern Usage in* xoComm/ActiWeb |
|---|---|
| Wrapper Facade [Schmidt 1999] | Legacy integration. |
| | Object wrapper for Tcl socket/event handling. |
| Layers [Buschmann *et al.* 1996] | Basic architectural structure. |
| Factory Method [Gamma *et al.* 1994] | Class-object-specific variant for request creation. |
| | Class-referencing variant for HTTP worker creation. |
| Flyweight [Gamma *et al.* 1994] | Sharing cached requests. |
| | Request joining. |
| | Persistent connections. |
| Observer [Gamma *et al.* 1994] | Callback notification. |
| | Connection observation. |
| Reactor [Schmidt *et al.* 2000] | Dispatches client side request events. |
| | Dispatches server side request events. |
| Acceptor/Connector [Schmidt *et al.* 2000] | Connection establishment. |
| Template Method [Gamma *et al.* 1994] | GET, PUT,... as templates for event processing invocation. |
| Per-Object Decorator [Neumann and Zdun 1999c] | Addition of a time sink. |
| | Mime type loading. |
| | Decoration with Access Control. |
| Proxy [Gamma *et al.* 1994] | Shadow file sink. |
| | Additional invokers. |
| Whole-Part [Buschmann *et al.* 1996] | Access objects aggregate sinks. |
| | Place aggregates HTTP server, agent management, ... |
| | Server aggregates workers. |
| Chain of Responsibility [Gamma *et al.* 1994] | Access Controller Chain. |
| Singleton [Gamma *et al.* 1994] | Place as specializable singleton. |

Table 2: Overview of Used Patterns