

Internet Draft  
expires in six months

Bill Janssen, Xerox PARC  
Henrik Frystyk Nielsen, W3C  
Mike Spreitzer, Xerox PARC  
1 August 1998

## HTTP-ng Architectural Model

[<draft-frystyk-httpng-arch-00.txt>](#)

### Status of this Document

\*\*\*\*\*

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

To view the entire list of current Internet-Drafts, please check the "1id-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories on ftp.is.co.za (Africa), ftp.nordu.net (Northern Europe), ftp.nis.garr.it (Southern Europe), munnari.oz.au (Pacific Rim), ftp.ietf.org (US East Coast), or ftp.isi.edu (US West Coast).

This document has been produced as part of the W3C HTTP-ng Activity (for current status, see "<http://www.w3.org/Protocols/HTTP-NG/Activity>"). This is work in progress and does not imply endorsement by, or the consensus of, either W3C or members of the HTTP-ng Protocol Design Working Group. We expect the document to evolve considerably as the project continues.

Distribution of this document is unlimited. Please send comments to the HTTP-NG mailing list at [<www-http-ng-comments@w3.org>](mailto:www-http-ng-comments@w3.org). Discussions are archived at "<http://www.w3.org/Protocols/HTTP-NG/>".

Please read the "HTTP-NG Short- and Longterm Goals" [[HTTP-ng-goals](#)] for a discussion of goals and requirements of a potential new generation of the HTTP protocol and how we intend to evaluate these goals.

### Abstract

\*\*\*\*\*

This document defines the architectural model for a new HTTP framework called HTTP-ng, along with a set of terms for referring to parts of it.

### Table Of Contents

\*\*\*\*\*

- [1. Introduction](#)
- [2. Overview of](#)

- [3. Architecture of an HTTP-ng-based Web](#)
- [4. The HTTP-ng Type System](#)
  - [4.1. Type IDs](#)
  - [4.2. Identifiers](#)
  - [4.3. The Boolean Type](#)
  - [4.4. Enumerated Types](#)
  - [4.5. Numeric Types](#)
    - [4.5.1. Fixed-point Types](#)
    - [4.5.2. Floating-point Types](#)
  - [4.6. String Types](#)
  - [4.7. Sequence Types](#)
  - [4.8. Array Types](#)
  - [4.9. Record Types](#)
  - [4.10. Union Types](#)
  - [4.11. The Pickle Type](#)
  - [4.12. Reference Types](#)
  - [4.13. Object Types](#)
    - [4.13.1. Supertypes and Inheritance](#)
    - [4.13.2. Methods](#)
    - [4.13.3. State](#)
    - [4.13.4. The HTTP-ng.RemoteObjectBase Type](#)
    - [4.13.5. Distributed Garbage Collection of Objects](#)
  - [4.14. The HTTP-ng.TypeReference Type](#)
- [5. Application Program Architectures](#)
- [6. References](#)
- [7. Address of Authors](#)

## [1. Introduction](#)

\*\*\*\*\*

This document describes a new architecture for HTTP, and the part of the World Wide Web that is built on top of the HTTP infrastructure. This work has been motivated by some observations about the current HTTP 1.x infrastructure.

HTTP began as a generic request-response protocol, designed to accommodate a variety of applications ranging from document exchange and management to searching and forms processing. As HTTP has developed, though, the request for extensions and new features has exploded; such extensions range from caching, distributed authoring and content negotiation to various remote procedure call mechanisms. By not having a modularized architecture, the price of new features has been an overly complex and incomprehensible protocol with the following drawbacks:

- \* Complexity
 

Even though HTTP/1.1 provides a number of interesting features it does not provide a clean framework for defining their interaction. This has resulted in the large and complex HTTP specification.
- \* Poor Extensibility
 

Acknowledging a large number of proposed extensions as part of the core protocol has stretched HTTP/1.1. The interactions between the extensions are complex at best, often unspecified or even broken.
- \* No Application Deployment Model
 

Many applications and services are being deployed on the web that

are not, in fact, retrieval of documents. Some applications are tunnelled through HTTP but would be more appropriately deployed as applications of a distributed object systems. Tunneling itself is being done in a variety of different ways, which causes problems for firewalls and other filtering intermediaries. The proliferation of application deployment schemes has increased developer confusion over how to use the Web.

\* Alternative Technologies

Alternative distributed-application deployment technologies, such as DCOM, CORBA, and Java RMI, have proposed different models of application development which don't integrate well with the current HTTP. As a result, these protocols have come to use HTTP and the Web either as a 'initialization service', delivering some startup code and/or application which then uses a non-Web technology, or as an expensive, albeit widespread, reliable datagram delivery service. These very similar distributed object systems tend to find different solutions to the same problems, which again increases overhead and decreases interoperability.

\* Poor Scalability

At the time it was designed, the HTTP/1.0 protocol still represented a very low fraction of today's Internet traffic. Caching and connection management has improved HTTP/1.1 but recent measurements show that the protocol overhead can still be much improved in terms of CPU and network efficiency.

These concerns have driven the design of the HTTP-ng architecture. The basic approach is to define a layered HTTP framework which provides both a Web distributed-object system, and transport layers which address the interaction with other Internet protocols. The layering allows modularization of the implementation framework, so that the effect of changes in one part of the system have minimal impact on other parts of the system. The distributed-object system allows application-specific interfaces to be defined, and provides an application deployment model; it keeps different applications from interfering with each other. The distributed-object system itself provides the core functionality of the distributed-object systems defined by DCOM, CORBA, and Java RMI; the intent is to make it directly usable by those technologies so that tunnelling becomes unnecessary. Elements of this architecture have been drawn from HTTP 1.1 [[RFC 2068](#)], ILU [[ILU](#)], PEP [[PEP](#)], DCOM [[DCOM](#)], CORBA [[CORBA](#)], and Java RMI [[Java RMI](#)].

## [2. Overview of Distributed Object Systems](#)

\*\*\*\*\*

Remote or distributed applications typically have a defined interface; this consists of a set of operations which participants in the application invoke on other participants. These operations typically are defined in terms of input parameters and output results; sometimes exceptional results, or exceptions, are also specified for the operation. In standard Internet usage, an application such as Telnet or FTP is described in a document which specifies an abstract model of operation, and enumerates the operations by describing the form of a message which conveys that operation when marshalled onto an appropriate transport substrate. Some Internet standards define an

application framework, which is a set of general information that applies to a family of applications, related by all using that framework. The [RFC 1831](#) for RPC is such a framework specification. Finally, some specifications, such as HTTP, define both a framework, and a particular application (the Web), which uses that framework. It does this by defining an extensible system of messages in a request/response context, then defines several specific messages ('GET', 'HEAD', 'POST', etc.) and the context of a distributed application that uses them.

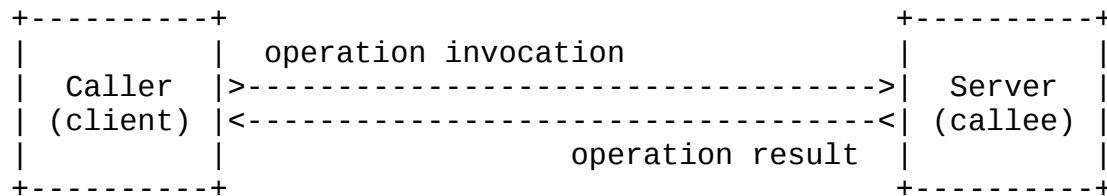


Figure 2a. Elements of a Distributed Application

Though there may be many participants or threads of control in the application, they conventionally engage in two-party interactions. These two parties can be categorized as the caller or client, which is the participant which invokes an operation, and the callee, or server, which is the participant implementing or responding to the operation. When there are many participants in the applications, the role each plays may shift continuously. These participants are said to reside in different compatibility domains; the distributed application serves as a bridge between these domains. There is some essential element which separates the domains; it is typically the fact that the participants are on different computers, but it may also be other factors such as that the two participants are operating under the aegis of different security principals, or are implemented in different programming languages.

Distributed object systems are application frameworks that try to simplify and optimize the definition and use of an application. They typically have an interface definition language (often called IDL), which make it easier to talk about the components of an interface. They often have standards for marshalling the parts of a message into binary forms, or for discovering resources, or for recovering from certain types of errors, or for exporting the interface defined in their IDL to a particular programming language for use in a larger program. This separation of concerns allows each part of the application framework to be small and cohesive, but allows the development of large applications. They allow the different participants to interact with each other having only limited, but precise, understanding of the other participants' capabilities.

Some distributed object systems use the procedure call as a metaphor of operation; these are called RPC systems. Others use a messaging model, and are typically called messaging systems. Some modern distributed object systems have support for both RPC and messaging.

In a typical distributed-object system, an interface definition consists of a set of inter-related types, exceptions, and methods. Methods are packaged in object types, developed with a partitioning of them according to object-oriented methodologies. A callee is

represented by an instance of an object type. Operations are invoked on the callee by calling one of its methods. If the caller of a method and the callee of the method exist in different compatibility domains, they are connected with a connection, which carries messages back and forth between the compatibility domains, and has an associated protocol and transport stack. The message is the basic unit of communication; the connection's protocol specifies the set of messages which can be sent, and their syntax; the connection's transport stack, which typically consists a series of individual transport elements, specifies how the messages are carried between the two domains. Most protocols define two distinguished messages, a request message, which invokes a service in the callee, and a response message, which provides a response about an invocation back to the caller, along with some other control messages used to synchronize state between the two domains. We typically distinguish between transport elements called transport filters, which modify the bits of message passing through them (encryption, compression, etc.) and other elements transport endpoints, which actually convey the message to a different compatibility domain (TCP/IP, UDP/IP, etc.).

An implementation of an object type is called a class. An implementation of one or more of the object types defined in an interface, along with whatever additional code is necessary to perform any ancillary tasks such as creating initial instances of a class, or registering instances with a name service, is called a module. A program typically consists of several modules, each of contains one or more classes which implement object types, either by dispatching messages to a module exported from another compatibility domain (classes which do this are called surrogate classes), or by directly performing the functionality of the object type (classes which do this are called true classes). Programs which tend to have many true classes and relatively few surrogate classes are typically called servers, and programs which have few true classes and relatively many surrogate classes are typically called clients; these terms should be used with caution, since they are only appropriate in a specific context. A server is typically made available by a publisher, which is the principal responsible for the services encapsulated in the true instances of object types provided by the module. The principal responsible for accessing the services provided by the publisher is sometimes known as the user.

### 3. Architecture of an HTTP-ng-based Web \*\*\*\*\*

In the HTTP-ng architecture, the Web is defined as an application on top of the HTTP-ng framework. A typical HTTP-ng application would be layered as described in [Section 2](#). The lowest layer, the transport layer, would typically (but not necessarily always) consist of a series of transport elements. A simple layering might consist of a MUX [[HTTP-ng-webmux](#)] transport filter over a TCP/IP transport endpoint; a more complicated transport layering might introduce layers concerned with connection management, security, compression, or other issues.

The MUX layer seems particularly useful; it is used to provide multiple connections over a single TCP/IP connection, bi-directional use of the TCP/IP connection for callbacks, and message-marking for the higher layers. We call the MUX connection a session to distinguish it

from TCP/IP connections. Similarly, we call the MUX port (the thing to which MUX connections are made, like a TCP/IP port), a channel, to similarly distinguish it. The set of MUX channels which can be connected to over a TCP/IP connection to a particular IP host machine is called a "MUX endpoint". Typically, this endpoint identifies a particular address space or process on an IP host. Note that a single MUX endpoint (and all the MUX channels at that endpoint) may be available via a number of different TCP ports. This means that the TCP port used in the transport stack does not explicitly identify a set of MUX channels; rather, the set of MUX channels are identified by the MUX endpoint.

The next layer, the operation invocation layer, would consist of the object-oriented HTTP-ng messaging protocol, which maximizes use of protocol features found to be successful in existing Internet protocols. This layer is a generic messaging layer - it does not provide any application-specific services like security or caching, or any other application layer functionality. It does provide an extensible and regular mechanism for forming and handling messages described by application-level interfaces without knowledge of semantic information that is application-specific. It provides a standard type system for defining application interfaces. It provides a distributed garbage-collection facility which can be used for automatic management of application objects. This layer also has associated machinery which allows automated uses of interface descriptions written in an interface description language, such as client-stub-code and implementation-skeleton generation. This layer may use a number of different, but equivalent, protocols; we expect the major one will be the efficient binary wire protocol [[HTTP-ng-wire](#)] defined in this suite of documents, but any protocol that properly supports the HTTP-ng type system can be used.

Given this protocol framework, the Web application is defined at the third and highest layer, the application layer. This layer is "application-specific", meaning that it varies from application to application. For example, "The Classic Web Application" (TCWA) would have a different layer here than the WebDAV application (though they might share some common part). The HTTP-ng architecture allows multiple applications to co-exist at this level, and provides a mechanism to add new applications easily without disturbing existing applications. The Web application is defined both statically, in terms of the type system at the second layer, and dynamically, in terms of the transport elements of the first layer. An associated document provides an initial sketch of what the interface definition for the Web application might look like [[HTTP-ng-interfaces](#)].

#### 4. The HTTP-ng Type System \*\*\*\*\*

Interfaces for applications are defined with a "type system". This section describes the proposed type system for HTTP-ng. It consists of two kinds of types, "primitive types" and "constructed types". Only two primitive types, or pre-defined types, are included, boolean and pickle. All other types are constructed from parameters and other types using "constructors", like sequence types and record types. Note that this type system provides support for both statically typed interfaces, and, with the pickle type, dynamically typed interfaces.



Note: This section currently uses the interface specification language ILU ISL [[ISL](#)]; it is not meant to be prescriptive. Any other interface language which captures this type system could be used instead. We expect that current widely used interface definition languages, such as DCOM MIDL or OMG IDL, will be adapted for use with HTTP-ng.

#### [4.1. Type IDs](#)

=====

All types have a single associated universally and globally unique identifier, called the "type ID", which can be expressed as a URI. Type IDs from different UUID spaces may be mixed. An implementation of the type system should allow explicit identification of type IDs with types, but should also provide a default type ID for every type in a consistent and predictable fashion. [ We need to define the algorithm here. ]

#### [4.2. Identifiers](#)

=====

The rules governing the syntax of identifiers are the same as for Standard C; that is, uppercase and lowercase alphabetic characters and digits from the ASCII character set, along with the underscore character, may be used, and the first character of the identifier must be an alphabetic character. Case is significant.

#### [4.3. Boolean Types](#)

=====

The single boolean type, the primitive type boolean, has exactly two values, 'True' and 'False'.

#### [4.4. Enumerated Types](#)

=====

An enumerated type is an abstract type the values of which are explicitly specified. It is specified with one parameter, a set of values, specified as identifiers. Enumerated types are not numeric types, and the values of an enumerated type do not have intrinsic numeric values associated with them; however, some programming languages may use numeric types to represent enumerated types.

#### [4.5. Numeric Types](#)

=====

The type system includes two different kinds of numeric types, "fixed-point" and "floating-point". All numeric types are constructed; that is, there are no pre-defined "primitive" integer or floating-point types.

##### [4.5.1. Fixed-point Types](#)

-----

A fixed-point type is defined with three parameters: a denominator, a maximum numerator value, and a minimum numerator value. These define a

series of rational values which make up the allowable values of that fixed-point type. The numerator and denominator are integer values; the denominator is either a positive integer value greater than zero, or the reciprocal of a positive integer value greater than zero. Each value of a fixed-point type abstractly exists as a rational number with a numerator in the range specified for numerators, and a denominator of the specified denominator value. For example, one could define a fixed-point type which would cover the 16-bit unsigned integer space with a denominator of one (all integer types have denominators of one), a maximum numerator of 65535 and a minimum numerator of zero. One could define a fixed-point type 'dollars' for business purposes with a denominator of 100 (two decimal places for 'cents'), a maximum numerator of 100000000 (1 million dollars) and a minimum numerator of -100000000 (1 million dollars). There are no limits on the sizes of denominators, maximum numerators, or minimum numerators.

We use this approach, instead of specifying a procrustean flock of predefined integer types as in DCOM or CORBA, to simplify the underlying system in several ways.

1. Small applications can handle all fixed-point values of a particular type as 'bignum' value (the numerator), and not have to have any special-case code for any primitive integer types. However, any primitive integer types they care about can be special-cased and handled efficiently.
2. This approach also supports the CORBA 'fixed' data type, but does so more effectively by additionally providing for non-decimal fixed-point fractional types: types such as sixteenths can also be defined directly for stock market applications by using a denominator of 16, while egg producers can also deal in dozens by specifying a denominator of 1/12.
3. Programming language mappings of this type system can special-case those integer types they support internally directly, and handle all other fixed-point types in a generic fashion. This eliminates the need to 'shoehorn' some numeric types into language types not quite right for them, such as the representation of CORBA 'unsigned' types in the Java programming language.

#### 4.5.2. Floating-point Types

-----

Floating-point types are specified with eight parameters:

- \* the size in bits of the significand,
- \* the base of the exponent,
- \* the maximum exponent value,
- \* the minimum exponent value,
- \* whether they support a distinguished value for 'Not-A-Number',
- \* whether they support a distinguished value for 'Infinity',
- \* whether denormalized values are allowed, and



\* whether the zero value is signed (whether they can have both +0 and -0). For instance, the floating point type usually described as IEEE 32-bit floating-point has the parameters significand-size=24, exponent-base=2, maximum-exponent-value=127, minimum-exponent-value=-126, has-Not-A-Number=TRUE, has-Infinity=TRUE, denormalized-values-allowed=TRUE, and has-signed-zero=TRUE; the floating point type usually described as IEEE 64-bit floating-point has the parameters significand-size=53, exponent-base=2, maximum-exponent-value=1023, minimum-exponent-value=-1022, has-Not-A-Number=TRUE, has-Infinity=TRUE, denormalized-values-allowed=TRUE, and has-signed-zero=TRUE. We expect that interface description languages for the HTTP-ng type system will provide shorthand notation for certain floating-point type patterns, such as those corresponding to IEEE single and double precision floating point.

We use this approach because CORBA and similar systems have a problem in that they have no way to represent the variety of floating point numbers available, particularly the different variants of IEEE 'extended'. In addition, this system allows representation of older floating-point types still in wide circulation, such as IBM, VAX, and CRAY floating-point, in an simple fashion.

#### 4.6. String Types

=====

"String" types are defined with two parameters: the maximum length in bytes of the string values, and the language [[RFC2277](#)] used in the string. If no language is specified, the language defaults to the Internet default language "i-default". The maximum length must be less than or equal to 0x7FFFFFFE, or it may also be omitted, in which case a maximum length of 0x7FFFFFFE ( $2^{31}-2$ ) is used. The character set (or sets) used in the string is that identified by the string's language. The codeset used in representations of the string is not specified at this level. This type system does not have any representation for individual characters or character codes; integer types should be used for that purpose.

[ Note that there is no such thing as a "NULL string", as occurs in C or C++. However, a similar type can be constructed with the optional type constructor, using a string type as the base type. ]

#### 4.7. Sequence Types

=====

Sequence types are variable-length one-dimensional arrays of some other type. They are defined with two parameters: the "base" type of the sequence, and optionally a maximum number of values in the sequence, which must be a value less than or equal to 0x7FFFFFFE. If no maximum length is specified, a default maximum length of 0x7FFFFFFE is assumed.

#### 4.8. Array Types

=====

Array types are fixed-length multi-dimensional arrays of some other type. They are defined with two parameters: the "base" type of the

sequence, and a sequence of dimensions, each of which is an integer value less than or equal to `0x7FFFFFFE`. Arrays are in row-major order.

#### 4.9. Record Types

=====

A record type is defined with a single parameter, a sequence of fields. Each field is specified with an identifier, unique within the scope of the record type, and a type.

#### 4.10. Union Types

=====

Union types are defined with a single parameter, a set of fields. Each field is specified with an identifier, unique within the scope of the union type, and a type for the field. The same type may be used multiple times for different fields of the union. The value of a union type with N fields consists of a single value of the type of one of the fields, and an unsigned integer value in the range [0..N-1] identifying which field is represented.

#### 4.11. Pickle Type

=====

A special type that can hold a value of any other type is known as the pickle type. This type essentially provides an escape mechanism from the static typing of the HTTP-ng type system; every pickle value is dynamically typed. A value stored in a pickle has a standard representation as a sequence of octet values; this representation of the value of the pickle may be accessed as used as an externalized form of the value. Pickles are opaque, by which we mean that the value stored in a pickle cannot be manipulated directly, but only through a procedural interface.

[ This is similar to the `any` type in CORBA, but also provides a standard externalization form for values. The opacity of the pickle also provides important marshalling efficiency gains over the `any`. ]

#### 4.12. Reference Types

=====

Reference types are specifically designed to support recursion through the type system, and must be mapped to a pointer type of some type in those programming languages which distinguish between value types and pointer types. Each use of a reference type constructor adds another level of indirection. Reference types are defined with a parameter, another type, called the base type of the reference type, and two optional modifiers optional and aliased. Thus it's possible to have non-aliased non-optional reference types, optional non-aliased reference types, non-optional aliased reference types, and optional aliased reference types.

If the reference type is declared optional, a value of the reference type may be either NIL or a value of its base type. This type allows the description of data structures like binary trees.

If the reference type is declared aliased, the reference type is distinguished in that multiple references to the same value of an aliased type in a single 'scope' will be passed between compatibility domains as a single value. This avoids the problem in some type systems of converting graph data structures to trees when transferring them between compatibility domains.

The scope of an aliased type varies depending on its use. When a value of an aliased type is being sent as an input or output parameter in a method call, the scope of the type is the whole message in which the value is being sent; that is, the invocation or result of the method call. When it is being pickled, the scope of the value type is the pickle.

[ These types effectively cover the practical uses of reference types in the CORBA objects-by-value spec, the Java RMI spec, and the DCE RPC / DCOM system. ]

#### 4.13. Object Types

=====

In the HTTP-ng type system, operations are modeled as method calls on an instance of an object type which supports that method. There are two kinds of object types, "local" and "remote". Instances of remote object types have a global identity, and are always passed between compatibility domains by reference; the original object value is called the "true instance", and references to that true instance are called "surrogate instances". Passed instances of any remote object type are always surrogate instances, except in the compatibility domain of the true instance, where it will be the true instance. Instances of local object types have no global identity, and are always passed between compatibility domains by copying the "state" of the local instance to the new compatibility domain, where a new instance of that object type (or a subtype of the object type) is created from the state. A local instance in one compatibility domain and copy of that local instance in a different compatibility domain have no explicit association after the copy has been made. The behavior of a local object type must be provided locally, but it may be provided statically at link time, or dynamically at runtime via dynamic loading of class code, as is done in Java RMI. The HTTP-ng system does not define any particular system for dynamic loading of behavior.

Both local and remote object types are defined by specifying three parameters: a sequence of supertypes, a sequence of methods, and the state of the object type. Note that the ordering of the elements of each of these parameters is significant. Any of these parameters may be empty.

[ Note that there is no such thing as a "NIL object", as occurs in the CORBA system. However, an equivalent construct may be obtained where necessary by using the optional type constructor, using an object type as the base type. ]

##### 4.13.1. Supertypes and Inheritance

-----

Each object type may have one or more object types as supertypes.

An object type with supertypes is said to "inherit" the methods and state of its supertypes, which means that it provides all of the methods provided by any of its supertypes, and any of their supertypes, and so forth. An object type may occur more than once in the supertype inheritance graph of an object type. Note that the inheritance provided in this model is that of interface inheritance. In particular, it does not guarantee or prohibit polymorphic dispatching of methods.

Object types may be "sealed". A sealed object type may not be used as a supertype for another object type.

#### 4.13.2. Methods

-----

A "method" is a way of invoking an operation on an instance of an object type. Each HTTP-ng method has a three required parameters: a synchronization attribute which may either be "synchronous" or "asynchronous"; an identifier, unique within the scope of the object type, called the "method name"; and an "invocation", which is specified as a sequence of named and typed input parameters. Synchronous methods may also have additional parameters: a "normal result", which is specified as a sequence of named and typed output parameters, and zero or more "exceptional results", each specified as a sequence of named and typed output parameters. Exceptional results should only be used to describe results that occur infrequently, as the overhead of handling exceptional results in most programming languages that support them is higher than the overhead of handling a normal result. Input parameters and output parameters are each specified with an identifier, unique within the scope of the method, and a type. [ Note additional restriction on the names of exception parameters. ]

Asynchronous methods transfer the invocation parameters to the compatibility domain of the true instance, and invoke the operation. The caller-side control flow for an asynchronous method returns to the caller of an asynchronous method after the message has been handed to a reliable transport mechanism for transfer to the compatibility domain of the true instance; the remote method itself may not be executed for an arbitrary length of time after that return. Asynchronous methods may be thought of as "messages".

Synchronous methods transfer the invocation parameters of the method to the compatibility domain of the true instance, where the method is invoked as a local method on that instance. The result from that invocation, normal or exceptional, is transferred back to the caller's compatibility domain, if different, and returned as the result of the method.

[ Do we need some set of 'standard' exceptions? If so, what are they? A small starting list might be:

- \* communications failure - underlying failure in the message transport subsystem
- \* no such object - server can't identify instance on which the method was invoked
- \* bad type - object doesn't have type specified in operation

identifier

- \* no such type - type specified in operation identifier unknown at server
- \* internal local before - something in the middleware system failed in this compatibility domain, before the invocation was attempted
- \* internal local after - something in the middleware system failed in this compatibility domain, after a result was received
- \* internal remote before - something in the middleware system failed in the remote compatibility domain, before the invocation was attempted
- \* internal remote after - something in the middleware system failed in the remote compatibility domain, after the invocation of the true method had begun ]

#### 4.13.3. State

-----

Each object type may have state associated with it. The state of an object type is specified as a sequence of attributes, where each attribute has an identifier unique within the scope of the object type, and an associated type. When an instance of this object type is passed between compatibility domains, the values of these attributes are passed. This is the normal way of passing the state of a local object type. [ TBD - for remote object types, the state may act as the initial values for surrogate instance caching of remote object state, or as a way of passing immutable metadata about the instance with the object reference.]

Attributes may also be marked as "public" or "private". This has no effect on their handling at the lower levels of the system, but may influence the way in which mappings to programming languages deal with them. For example, a programming language object type might provide public accessors for public attributes, but not for private attributes.

#### 4.13.4. The HTTP-ng.RemoteObjectBase Type

-----

All remote object types must inherit directly or indirectly from the object type HTTP-ng.RemoteObjectBase.

```
INTERFACE HTTP-ng BRAND "http-ng.w3.org";
...
TYPE UUIDString = STRING LANGUAGE "i-default"
  TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/UUIDString";

TYPE TypeIDTreeNode = RECORD
  type-id : UUIDString,
  "supertypes" : InheritanceHierarchy
END TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/TypeIDTreeNode";

TYPE TypeIDTreeNodeRef = ALIASED REFERENCE TypeIDTreeNode
  TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/TypeIDTreeNodeRef";
```

```

TYPE InheritanceHierarchy = SEQUENCE OF TypeIDTreeNodeRef
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/InheritanceHierarchy";

TYPE RemoteObjectBase = OBJECT
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/RemoteObjectBase"
    METHODS
        GetTypeHierarchy () : TypeIDTreeNodeRef
            "Returns the type ID hierarchy for the object, rooted
             at the most-derived type of the object"
    END;
...

```

#### 4.13.5. Distributed Garbage Collection of Objects

-----

A simple form of garbage collection is defined for HTTP-ng objects. If an object type inherits from HTTP-ng.GCCollectibleObjectBase, a module that implements objects of that type expects clients holding surrogate instances to register with it, passing an instance of a callback object. When a client finishes with the surrogate, the client unregisters itself. Thus the server may maintain a list of clients that hold surrogate instances. If no client is registered for an object, and the object has been dormant (had no methods called on it by a client from a different compatibility domain) for a period of time T1, the module may feel free to garbage collect the true instance. T1 is determined both by human concerns and network performance: T1 is set long enough to allow useful debugging of a client, and longer than the typical transmission delay between all participants in a program.

To deal with possible failure of a client process, we introduce another time-out parameter. If an instance with registered clients has been dormant for a period of time T2, the server uses the callback instance associated with each client to see if the client still exists. If the client cannot be contacted for the callback, the server may remove it from the list of registered clients for that instance.

Object types which participate in distributed garbage collection must inherit from HTTP-ng.GCCollectibleObjectBase.

```

INTERFACE HTTP-ng BRAND "http-ng.w3.org";
...
TYPE Seconds = FIXED-POINT DENOMINATOR=1
    MIN-NUMERATOR=0 MAX-NUMERATOR=0xFFFFFFFF
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/Seconds";

TYPE GCCallBackObject = OBJECT SUPERTYPES RemoteObjectBase END
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/GCCallBackObject"
    METHODS
        StillInterested (obj : GCCollectibleObjectBase) : BOOLEAN
            "Should return TRUE if the callback object is still interested in
             using the remote object 'obj', FALSE otherwise. An error return
             is counted as a FALSE return."
    END;

TYPE GCCollectibleObjectBase = OBJECT SUPERTYPES RemoteObjectBase END
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/GCCollectibleObjectBase"
    METHODS

```



```

RegisterGCInterest (user : GCCallBackObject)
    "Indicates that the caller, indicated by the 'user' parameter, is
    interested in using the object, and should be considered a reference
    for the purposes of distributed garbage collection. This may be
    called multiple times with the same 'user' parameter, but only one
    reference per distinct 'user' parameter will be registered.",
UnregisterGCInterest (user : GCCallBackObject)
    "Indicates that the caller, indicated by the 'user' parameter, is
    no longer interested in using the object, and should no longer be
    considered a reference for the purposes of distributed garbage
    collection. It is not an error to call this for objects which have
    not previously registered interest.",
GetTimeoutParameters(OUT t1 : Seconds, OUT t2 : Seconds)
    "Returns the T1 and T2 garbage collection parameters used by the
    server.",
Ping()
    "Can be used by surrogate users of the instance to refresh the T1
    timeout of the instance, and prevent server-side outcalls to the
    callback object."
END;
...

```

#### 4.14. The `HTTP-ng.TypeReference' Type =====

It's often useful to have a standard data representation for the description of a type. CORBA, for example, defines the pseudo-type `Typecode', which is defined to be a type which makes various attributes of the description of another type available to the application program. Rather than defining a pseudo-type, the HTTP-ng interface simply defines a representation of a type description in the type system itself. Many other possible ways of representing the type description are also possible; the type `HTTP-ng.TypeReference' is defined as a convenience.

```

INTERFACE HTTP-ng BRAND "http-ng.w3.org";
...
TYPE Identifier = STRING LANGUAGE "i-default"
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/Identifier";

TYPE PrimitiveTypeDescription = ENUMERATION "boolean", "pickle" END
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/PrimitiveTypeDescription";

TYPE EnumeratedTypeDescription = SEQUENCE OF Identifier
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/EnumeratedTypeDescription";

TYPE IntegerLiteral = FIXEDPOINT DENOMINATOR 1
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/IntegerLiteral";
TYPE NonNegativeNonZeroIntegerLiteral = FIXEDPOINT MIN-NUMERATOR 1 DENOMINATOR 1
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/NonNegativeNonZeroIntegerLiteral";
TYPE OptionalNonNegativeNonZeroIntegerLiteral = OPTIONAL NonNegativeNonZeroIntegerLiteral
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/OptionalNonNegativeNonZeroIntegerLiteral";

TYPE FixedPointTypeDescription = RECORD
    "denominator" : NonNegativeNonZeroIntegerLiteral,
    denominator-reciprocal : BOOLEAN,
    "min-numerator" : OptionalNonNegativeNonZeroIntegerLiteral,
    "max-numerator" : OptionalNonNegativeNonZeroIntegerLiteral

```

```

END TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/FixedPointTypeDescription";

TYPE FloatingPointTypeDescription = RECORD
    significand-size : NonNegativeNonZeroIntegerLiteral,
    exponent-base : NonNegativeNonZeroIntegerLiteral,
    min-exponent : IntegerLiteral,
    max-exponent : IntegerLiteral,
    has-Not-A-Number : BOOLEAN,
    has-Infinity : BOOLEAN,
    denormalized-values-allowed : BOOLEAN,
    has-signed-zero : BOOLEAN
END TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/FloatingPointTypeDescription";

TYPE LimitInteger = FIXEDPOINT MIN-NUMERATOR 1 MAX-NUMERATOR 0x7FFFFFFF DE
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/LimitInteger";
TYPE StringLanguage = STRING LANGUAGE "i-default"
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/StringLanguage";

TYPE StringTypeDescription = RECORD
    "limit" : LimitInteger,
    "language" : StringLanguage
END TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/StringTypeDescription";

TYPE SequenceTypeDescription = RECORD
    "limit" : LimitInteger,
    base-type : TypeReference
END TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/SequenceTypeDescription";

TYPE ArrayDimensions = SEQUENCE OF LimitInteger
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/ArrayDimensions";

TYPE ArrayTypeDescription = RECORD
    dimensions : ArrayDimensions,
    base-type : TypeReference
END TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/ArrayTypeDescription";

TYPE RecordOrUnionField = RECORD
    name : Identifier,
    "type" : TypeReference
END TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/RecordOrUnionField";

TYPE RecordTypeDescription = SEQUENCE OF RecordOrUnionField
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/RecordTypeDescription/";

TYPE UnionTypeDescription = SEQUENCE OF RecordOrUnionField
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/UnionTypeDescription";

TYPE ReferenceTypeDescription = RECORD
    "optional" : BOOLEAN,
    "aliased" : BOOLEAN,
    base-type : TypeReference
END
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/ReferenceTypeDescription";

TYPE ObjectSupertypeSequence = SEQUENCE OF TypeReference
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/ObjectSupertypeSequence";
TYPE ObjectStateField = RECORD

```

```

    name : Identifier,
    "type" : TypeReference,
    "private" : BOOLEAN
END TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/ObjectStateField";
TYPE ObjectState = SEQUENCE OF ObjectStateField
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/ObjectState";

TYPE MethodSynchronization = ENUMERATION "synchronous", "asynchronous" END
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/MethodSynchronization";

TYPE ParameterDescription = RECORD
    name : Identifier,
    "type" : TypeReference
END TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/ParameterDescription";

TYPE MethodInvocationParameterDescriptions = SEQUENCE OF ParameterDescription
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/MethodInvocationParameterDescripti";

TYPE MethodResultParameterDescriptions = SEQUENCE OF ParameterDescription
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/MethodResultParameterDescripti";

TYPE MethodResultDescriptions = SEQUENCE OF MethodResultParameterDescriptions
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/MethodResultDescriptions";

TYPE MethodOptionalResultDescriptions = OPTIONAL REFERENCE MethodResultDescriptions
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/MethodOptionalResultDescriptions";

TYPE MethodDescription = RECORD
    name : Identifier,
    synchronization : MethodSynchronization,
    invocation : MethodInvocationParameterDescriptions,
    results : MethodOptionalResultDescriptions
END TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/MethodDescription";

TYPE ObjectMethodSequence = SEQUENCE OF MethodDescription
    TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/ObjectMethodSequence";

TYPE ObjectTypeDescription = RECORD
    "local" : BOOLEAN,
    "sealed" : BOOLEAN,
    "supertypes" : ObjectSupertypeSequence,
    "state" : ObjectState,
    "methods" : ObjectMethodSequence
END TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/ObjectTypeDescription";

TYPE ConstructedTypeDescription = UNION
    "enumerated" : EnumeratedTypeDescription,
    "fixedpoint" : FixedPointTypeDescription,
    "floatingpoint" : FloatingPointTypeDescription,
    "string" : StringTypeDescription,
    "sequence" : SequenceTypeDescription,
    "array" : ArrayTypeDescription,
    "record" : RecordTypeDescription,
    "union" : UnionTypeDescription,
    "reference" : ReferenceTypeDescription,
    "object" : ObjectTypeDescription
END TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/ConstructedTypeDescription";

```

```

TYPE TypeDescription = UNION
  primitive : PrimitiveTypeDescription,
  constructed : ConstructedTypeDescription
END TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/TypeDescription";

TYPE UnaliasedTypeReference = RECORD
  uuid : UUIDString,
  description : TypeDescription
END TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/UnaliasedTypeReference";

TYPE TypeReference = ALIASED REFERENCE UnaliasedTypeReference
  TYPEID "http-ng-typeid://http-ng.w3.org/HTTP-ng/TypeReference";
...

```

## 5. Program Architectures

\*\*\*\*\*

Figure 5a illustrates the general arrangement of layers an HTTP-ng-enabled program. Each application would contain application-specific code to use the classes of interfaces it imports, and to provide the functionality of classes it exports. Any number of different applications can be supported on a common base. Note that multiple protocols (message formats) can co-exist simultaneously; note that several different transport chains can share a single TCP/IP port or connection by using the MUX protocol; note that additional processing of messages, such as compression, can be performed by using an appropriate transport chain in the transport layer.

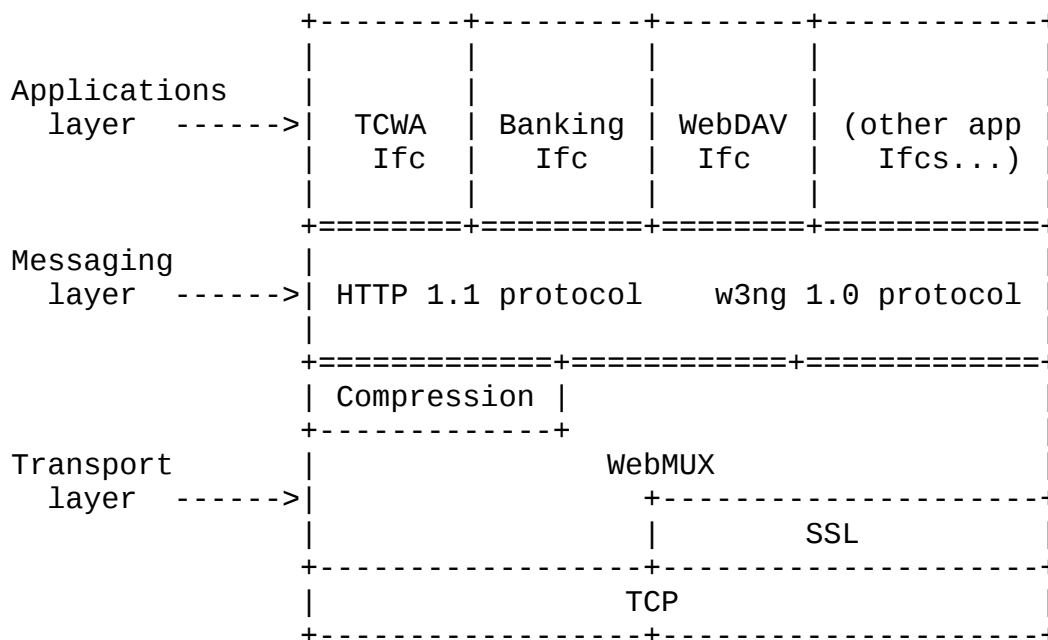


Figure 5a. Layers of a Sample Program

In an application like the Web, there would be at least two programs involved in an interaction: a web browsers, and one or more web servers and/or proxies. At the gross level, the programs would appear similar, as each has this layered architecture. With a more detailed inspection, differences corresponding to the essential client-like behavior of the browser and server-like behavior of the web server

appear. For instance, the browser tends to have more surrogate classes for the object types defined in the Web interface, and the server tends to have more true classes. The server may also have more different kinds of protocols and transport elements, in order to cope with a wider variety of possible clients.

For example, a server might speak both HTTP and w3ng; the browser might speak only w3ng. The server might support HTTP-ng messages over transports other than TCP/IP, while the browser might expect every server to have both webmux and TCP capability. Each might be capable of handling digitally signed documents. Both client and server might be able to use a compression transport filter on their message streams. Many other possible combinations of transport filters and protocols are possible on both sides. Some programs, such as proxy servers, might combine all the attributes of clients and servers, and add others, such as special interface extensions for caching.

## 6. References \*\*\*\*\*

[CORBA] CORBA/IIOP 2.2 Specification; Object Management Group, 1998. (See ``http://www.omg.org/corba/corbiiop.htm'.`)

[DCOM] Distributed Component Object Model; Microsoft, 1998. (See ``http://www.microsoft.com/com/dcom.htm'.`)

[HTTP-ng-goals]: HTTP-ng Short- and Long-term Goals. (See ``http://www.w3.org/TR/WD-http-ng-goals'.`)

[HTTP-ng-interfaces] HTTP-ng Web Interfaces. (See ``http://www.w3.org/TR/WD-HTTP-NG-interfaces'.`)

[HTTP-ng-webmux] HTTP-ng WEBMUX Protocol Specification. (See ``http://www.w3.org/TR/WD-mux'.`)

[HTTP-ng-wire] w3ng: Binary Wire Protocol for HTTP-ng. (See ``http://www.w3.org/TR/WD-HTTP-NG-wire'.`)

[ILU] ILU Reference Manual, W. Janssen, M. Spreitzer, 1998. (See ``ftp://ftp.parc.xerox.com/pub/ilu/ilu.html'.`)

[ISL] ILU Reference Manual, Chapter 2: "The ISL Interface Specification Language", W. Janssen, M. Spreitzer, 1998. (See ``ftp://ftp.parc.xerox.com/pub/ilu/2.0a12/manual-html/manual_2.html'.`)

[Java RMI] RMI - Remote Method Invocation; Sun Microsystems, 1997. (See ``http://java.sun.com/products/jdk/1.1/docs/guide/rmi/index.html'.`)

[RFC 2277] [RFC 2277](#), IETF Policy on Character Sets and Languages; H. Alvestrand, January 1998. (See ``http://info.internet.isi.edu:80/in-notes/rfc/files/rfc2277.txt'.`)

[RFC 2068] [RFC 2068](#), Hypertext Transfer Protocol - HTTP/1.1; R. Fielding et. al., January 1997. (See ``http://www.w3.org/Protocols/rfc2068/rfc2068'.`)

[PEP] PEP - An Extension Mechanism for HTTP; W3C, 1998. (See

`http://www.w3.org/Protocols/PEP/'.)

## 7. Address of Authors

\*\*\*\*\*

Bill Janssen

Mail: Xerox Palo Alto Research Center  
3333 Coyote Hill Rd  
Palo Alto, CA 94304  
USA

Phone: (650) 812-4763

FAX: (650) 812-4777

Email: janssen@parc.xerox.com

HTTP: `http://www.parc.xerox.com/istl/members/janssen/'

Henrik Frystyk Nielsen

Mail: World Wide Web Consortium  
MIT/LCS NE43-348  
545 Technology Square  
Cambridge, MA 02139  
USA

Phone: + 1.617.258.8143

FAX: + 1.617.258.5999

Email: frystyk@w3.org

HTTP: `http://www.w3.org/People/Frystyk/'

Mike Spreitzer

Mail: Xerox Palo Alto Research Center  
3333 Coyote Hill Rd  
Palo Alto, CA 94304  
USA

Phone: (650) 812-4833

FAX: (650) 812-4471

Email: mike-spreitzer@acm.org

HTTP: `http://www.parc.xerox.com/csl/members/spreitze/'