Abstract

We report on the results of the Protocol Design Group of the W3C's HTTP ``Next Generation'' Activity. The group produced and measured a prototype that shows it is possible, largely using familiar engineering principles, to make simultaneous improvements in the following problem areas of HTTP/1.1 [Fie99]: (1) the layering of other application protocols over HTTP, (2) modularity and extensibility, (3) networking performance and fairness, (4) the rigid binding between identifiers and protocol stacks, and (5) the opacity of layered traffic to firewalls. The prototype also suggests that these can be done in a way that may lead to unifying the web with related middleware systems such as COM [Bro95, Msf99a, Kin97], CORBA [OMG99], and Java RMI [Sun99].

Keywords: HTTP-NG, type system, Web applications, distributed objects, RPC

1. Introduction

In mid-1997 the W3C chartered an Activity (currently only post-conclusion web pages [W3C99a, W3C99b] are available) on HTTP-NG. The Activity consisted of two parts, one devoted to characterizing the web and one devoted to prototyping a major revision to HTTP. The second part, known as the Protocol Design Group (PDG), ran through late 1998 and produced and measured a prototype to study the feasibility of moving the web onto an application-independent distributed object system. The prototype showed that it is indeed possible to make improvements in a number of problem areas of HTTP/1.1, and that this can be done in a way that could lead to the unification of the web with the related middleware systems COM, CORBA, and Java RMI.

The group addressed the following problem areas.

1.1. Modularity & Extensibility

Although HTTP's initial success was fueled by simplicity, HTTP is no longer simple. The HTTP/1.1 specification [Fie99] is 175 pages long. Over the years many features have been added, each for a reason. Simplicity encompasses many things, only some of which remain within the realm of possibility for HTTP. Because of the many strong demands placed on it, HTTP cannot return to being a small protocol. However, HTTP could be made much more modular.

HTTP currently addresses concerns over a wide range of levels of abstraction. These include low-level transport issues, such as persistent connections and the delimiting of messages. These also include mid-level issues, such as regular patterns for identifying methods and passing parameters, as you find in RPC
HTTP is now being used for applications other than TCWA, which causes additional problems. These other applications include those that are closely related (e.g., WebDAV [Gol99], DASL [Dsl99], and DELTAV [Dtv99]), those that are less closely related (e.g., IPP [Her99], SWAP [Red99]), those that are based on partial clones of HTTP (e.g., SIP [Han99]), and those that are completely unrelated and based on the layering over HTTP of middleware systems of independent origin (e.g., COM, Java RMI, and some CORBA implementations) and of web-conscious origin (e.g., XML-RPC [ULS99], SOAP [Box99]). These other applications do not benefit from the parts of HTTP specific to TCWA, and because those parts are not well separated there are resulting inefficiencies and confusions. The parameter passing technique in HTTP is based on mail "headers" [Cro82] (excepting the one distinguished, optional parameter that is the MIME-typed "body"). This parameter passing technique does not address structuring of recursive data, and requires a level of quoting/encoding to pass arbitrary application data. The use of HTTP for other applications invites confusions [Moo98] between the other application and HTTP's document application. The following questions, which must be answered in the course of developing applications layered on top of HTTP, are symptoms of this confusion.

- Should the application use a different port than the HTTP default of 80?
- Should the application use traditional HTTP methods (GET, POST, etc.) or should it define new methods?
- Should the application use http: URLs or define its own?
- Should the application define its own MIME-types, or use something that already exists?

Answering questions like this has been a real problem. For example, this was a contentious issue in the development of the Internet Printing Protocol. The 1.0 version was taken off the Standards Track at the last minute by the Internet Engineering Steering Group, due partly to their disagreeing with the Working Group on choice of URI scheme [IPP99, Her99: IESG Note].

The well known paradigm of explicit interfaces, as used in modern programming languages and RPC systems, would help manage the co-existence and evolution of the multiplicity of applications of HTTP, but HTTP does not support that paradigm. The closest it comes is the OPTIONS method, which only reveals "communication options" that manifest themselves through response headers.

HTTP offers limited support for decentralized evolution. Decentralized evolution is what happens when no one organization is in engineering control of a widely deployed distributed system (such as the web). Evolutionary changes are independently developed by multiple independent organizations. At any given point in time, multiple such changes are in the process of being incrementally rolled out into the deployed system. In general, any given client and server have some extensions in common, and each also has extensions not supported by the other. When such a client and server interact, it is desirable for them to automatically employ the extensions they both understand, without a lot of latency or code complexity due to negotiation. HTTP supports this, with its rule that extension headers are optional and ignorable. However, it is also desirable for "mandatory" extensions to be possible. When one peer employs a mandatory extension, the other must either support that extension or signal an error. Again, it should be possible to employ multiple mandatory extensions without a lot of latency or code complexity due to negotiation. Mandatory extensions are always easy to support on the receiving side (it can raise an error on its own whenever desired). HTTP does not give senders much support for mandatory extensions. The only available technique is to use an entirely new method --- which loses all the benefits of formally being an incremental
change, notably including the automatic combination of multiple extensions.

1.2. Networking Performance and Fairness

HTTP has networking performance and fairness problems. These include inefficient use of the network, poor latency delivered to the users, and a tendency to abuse the network with multiple parallel TCP streams.

The use of verbose text-based representations for "headers" and other protocol-level data, and the confusion between that layer and human-readable documents and mail messages, leads to unnecessarily high numbers of bytes used for protocol overheads and method parameters.

HTTP's use of TCP suffers from high latencies for three reasons. One is simply that it takes time to transmit the unnecessarily large number of bytes. The next reason follows from the use multiple parallel TCP streams. Commonly deployed TCP implementations do not share congestion-avoidance state between parallel streams, leading to less efficient reactions to network conditions. In particular, the streams may ramp up to full speed more slowly than possible, and may provoke more congestion and consequent back-off than necessary; both increase the latency suffered by users. Finally, opening a TCP connection involves three messages over one and a half round trips; there have been proposals [Bra94] for how to reduce this to just one message prepended to the data stream.

There is a tendency for browsers to open multiple parallel HTTP/TCP streams, for two reasons. One is that with the currently deployed router policies, this tends to deliver an unfairly large share of the available network bandwidth. No changes in HTTP can fix this; the needed fix is to router policies. But even if and when the unfairness is fixed, there is another reason for browsers to use parallel streams (with the undesirable consequences mentioned above as well as the current unfairness). When first fetching a new web page with inlined images, it is advantageous to be able to quickly fetch the first few bytes of each inlined image because those bytes tend to contain metadata critical to computing the page's overall layout. Some browser vendors (e.g., Netscape) have chosen to do this by invoking GETs in multiple parallel connections. While it might appear that there is a viable alternative in using HTTP/1.1's Range header, there are some problems with that approach: (1) it is relatively recently standardized, so uniform server support cannot be assumed; and (2) the Range header's interactions with other features of HTTP are unclear or bad (e.g., HTTP/1.1 offers integrity checking on messages, not documents [Mog99]).

1.3. URIs bound to protocol stacks

HTTP can, in principle, be used over a great variety of "transport" substacks, but the "http" URI scheme is bound specifically to TCP. There is another scheme ("https") bound to TLS/SSL over TCP. One could imagine other transport substacks (e.g., a wireless version of TCP). In the current architecture, each particular choice of transport substack requires a new URI scheme. New URI schemes are painful to deploy, because, among other things, each one forms its own name space. Further, it's inconceivable for a URI to have multiple schemes (e.g., to offer multiple alternative transport substacks). Without the possibility of multiple stacks, it is not possible to incrementally move the current web onto a new transport substack --- nor offer multiple alternatives for any other reason --- except by using mechanisms (such as HTTP redirections or UPGRADE) that cost additional round trips.

1.4. Tunnelled traffic vs. firewalls

The practice of tunnelling general applications through HTTP makes the job of a firewall harder. We must be clear on the job of a firewall. A firewall, if it passes any traffic between them at all, cannot prevent collusion between an attacker inside the firewall and an attacker outside the firewall. Nor does a firewall benefit a trusted and trustworthy insider who fully secures his machines (this involves high administrative and
operational burdens, and the use of security-bug-free software). A firewall’s job is to make it easier for a trusted insider to be trustworthy, by enforcing certain limits on traffic between the inside and outside. The great variety and obscurity of ways of tunnelling general applications through HTTP makes it hard for a firewall to do anything with HTTP traffic.

1.5. Unifying the web with COM, CORBA, and Java RMI

The prototype solution to the above problem areas is based on factoring HTTP into three layers: (1) transport of opaque byte or message streams, (2) application-independent remote method invocation, and (3) the document fetching/storing application. The lower two layers suffice to serve the needs of the other applications currently being layered over all of HTTP, and provide a more robust platform on which to deploy and evolve a large collection of applications. There is a significant overlap among the problems addressed by this platform and the problems addressed by COM, CORBA, and Java RMI. This suggests an intriguing possibility. It starts with making a single wire protocol able to carry COM, CORBA, and Java RMI as well as web traffic. This provides interoperability, at least in the areas where the features of those systems overlap. Additionally, this could spur further convergence between those systems.

In the next section we present an overview of the prototype design, focussing on the lessons learned in solving the above problems. In section 3 we present the experimental results, which show that it is indeed possible to improve performance even while using a design and implementation that are more modular. In section 4 we briefly consider future and related work.

2. The Prototype Design

The prototype design shows a feasible way to make simultaneous improvements in all the problem areas above. This is largely done by straightforward application of well-known engineering principles. The first principle applied is divide-and-conquer. The major application of this principle is the division of HTTP’s functionality into three conventional layers; this yields a significant dose of simplicity (i.e., modularity), and is key to realizing a deep unification of the web with related middleware systems. The lowest layer addresses transport of opaque messages or streams, in a way that allows composition of "transport filters"; included is a design of a particular multiplexing filter that addresses some shortcomings of current TCP and provides a service abstraction that can insulate higher layers from certain desirable and expected changes in the lower levels. The middle layer addresses application-independent RPC, including typed messages as a degenerate case. The highest layer expresses the web as an application of the lower two layers.

We consider these layers in turn.

2.1. The Transport Layer

The transport layer addresses problems with modularity and extensibility, networking performance and fairness, transport flexibility, and even makes a contribution to unifying middleware systems. All this is easily done using familiar ideas, mainly: (1) a system of filters (as, e.g., in UNIX shell commands), and (2) multiplexing.

The transport layer, which is inspired by and very similar to the corresponding layer in ILU [ILU99], addresses reliable ordered bidirectional transport of opaque byte or message streams. An HTTP-NG connection can employ a stack of transport filters. A transport filter implements reliable ordered bidirectional transport of opaque byte streams or of opaque messages. The filter may do this either by directly using the services of protocols outside the scope of this design (e.g., TCP) or indirectly by using the services of the
next lower filter in the stack. A filter can have explicit parameters, such as the TCP port number to use.

This transport layer is modular and offers controlled evolution; its flexibility is part of the solution to the problem of the strong linkage between URIs and protocol stacks. The remainder of the solution appears in the middle layer, where the choice of stack is communicated in a way that is independent of object identifiers.

The prototype design details two particular transport filters. One implements byte streams by directly using TCP. This filter has two parameters, a host name or address and a port number. The other filter is a multiplexer, described in the next section.

2.2. The MUX Filter

The MUX filter addresses network performance and fairness problems, and contributes to middleware system unification. It does these by adding to the functionality delivered by its underlying stack in three ways. The underlying stack delivers reliable ordered bidirectional transport of opaque byte streams (e.g., by TCP). The MUX filter adds: (1) the delimiting of messages; (2) the multiplexing of parallel message streams over a single underlying byte stream; (3) the ability of the accepting (server) side of the byte stream connection to initiate message stream connections in the reverse direction.

The multiplexing of parallel message streams into a single byte stream addresses some of the network performance problems of HTTP. By moving the multiplexing up a level, the problems associated with lack of sharing of congestion-avoidance state between parallel TCP streams are avoided. The MUX filter can open a new message stream over an existing byte stream with the sending of only one message, at the start of the message stream; this saves a round trip compared to the cost of opening a new parallel TCP stream. The higher layers can open parallel message streams without paying the penalties of parallel TCP streams.

The ability of a byte stream connection receiver to initiate message stream connections in the reverse direction may be useful both (1) to solve the same performance problems mentioned above and (2) to enable callbacks from servers to clients behind firewalls. This contributes to unifying COM with other middleware systems. COM is unique in that a method parameter may be a callback function. Of course, a callback function can be considered just a special case of an object --- except for the interaction with firewalls. Calling an object normally requires the caller to open a connection to the object's server. For an object modelling a callback function, this server is the client of the outer call --- and may well be behind a firewall. Firewalls typically do not pass TCP connections initiated externally. Carving out this exception is reasonable, because it only allows traffic that the protected party has specifically requested and will interpret. Enabling message streams to be initiated in the reverse direction thus eliminates a problem with using objects to deliver the functionality of COM's callback functions.

Introducing this new layer of multiplexing introduces new possibilities of interference between independent connections. This can be partly solved by adopting TCP's flow control design, but that doesn't eliminate all the possible bad interactions.

The MUX protocol applies flow control independently to each message stream. Thus, a stall in one message stream does not block other message streams multiplexed with it. Flow is controlled by limiting the sender to a window, in terms of the total number of message payload bytes sent, given by the receiver. This is essentially the same as TCP's flow control. TCP's congestion avoidance ("slow start") does not need to be repeated at the MUX layer, as its presence in the underlying stack (e.g., in TCP) is sufficient.

There remains one potentially significant bad interaction between parallel message streams. If the underlying byte stream (e.g., TCP) encounters a delay due to an internal re-transmission (e.g., due to a lost IP packet), then all of the message streams multiplexed over it suffer that delay --- even if the lost packet...
The IETF has chartered a Working Group on "Endpoint Congestion Management" [Ecm99], which is intended to develop a way for parallel connections (both TCP and other kinds) to share congestion-avoidance state. By solving the state-sharing problem where it arises, this would eliminate the need for another layer of multiplexing above TCP. It would not eliminate the need for the other two things the MUX filter does. The flexibility of HTTP-NG's transport layer would allow a switch from MUX over TCP to a MUX--over TCP++ without significant disruption of the other layers.

Introducing a new layer of multiplexing requires a solution to the problems of identifying the streams multiplexed together, and of identifying which of potentially several targets a stream is connected to. The obvious and simple technique of using numeric IDs suffices. The HTTP-NG prototype design goes a bit further, exploring opportunities that come with associating further semantics with those numbers.

Among those multiplexed together, a message stream connection is identified by a number called a "session ID". The space of session IDs is divided in two: one half to be allocated by the byte stream initiator, the other by the byte stream acceptor.

The MUX filter supports the possibility of multiple message stream acceptors on each side of the byte stream, through a technique that is inspired by and extends TCP's notion of port numbers for passive sockets. Each message stream acceptor is identified by a number; the initiator sends this number as part of opening a new message stream connection. The extension is that these numbers are known as "Protocol IDs", and may be used to support a simple form of negotiation for the protocol stack above the message stream. The space of protocol IDs is divided into four parts: (1) one with an ID for each possible TCP port number; (2) one with an ID for each possible UDP port number; (3) one where the IDs are allocated by the server at its discretion; and (4) one where the IDs are associated by the initiator with a URI that identifies the protocol stack above. The value of the first two parts is that they provide a standard way to use the MUX protocol to tunnel TCP and UDP over TCP, fixing the performance problems above in a way that is transparent to applications. The third part of the Protocol ID space functions analogously to the accepting side's port number in TCP: the acceptor uses one that either is allocated by an external (to the MUX protocol) process or is purely ephemeral, and the way that the initiator learns of that number is outside the scope of the MUX protocol. The fourth part of the Protocol ID space is intended to enable a message stream acceptor to have dynamic flexibility in protocol stacks. The idea is that rather than allocate a number for every supported combination and parameterization of transport filters and higher protocols (with the ability to combine and parameterize filters, the number of supported superstacks could get quite high), the client simply states the desired superior stack at the start of a message stream. We have not yet explored this much further, but it costs little to carve out this piece of Protocol ID space.

An issue not fully explored is how to identify "endpoints". For the ability to initiate message stream connections in the reverse direction to be useful, it is helpful to allow the byte stream initiator to give the acceptor some identification of what services can be reached at the initiator side. To this end, the MUX protocol allows one peer to send the other a message listing endpoints that are available. The MUX protocol says about endpoints only that they are URIs, with hierarchy --- if any is defined by the URI's scheme --- respected. That is, if one side wishes to open a message stream connection to endpoint "sch:X/Y" and has in hand a byte stream connection initiated by a peer that advertises endpoint "sch:X/", that byte stream connection may be used for the new message stream connection. Plausible things that an endpoint might actually identify include: (1) a particular host; (2) a particular process (address space); and (3) a particular software module instance in a particular process.

2.3. The Remote Invocation Layer
The middle layer of the prototype HTTP-NG design is an application-independent remote method invocation layer. It provides object oriented method calls, similarly to COM, CORBA, and Java RMI; HTTP's document fetching/storing application becomes a definition of a particular network interface for that application (the third layer, discussed later). The middle layer addresses problems of modularity and extensibility, network performance, the binding of object identifiers to protocol stacks, the opacity of tunnelled traffic to firewalls, and the unification of the web with related middleware systems.

One lesson to draw from this exercise is that familiar ideas about how to design RPC systems can easily be applied to the web, and that doing so yields the benefits we describe. The very idea of RPC --- particularly when done with explicit interfaces --- makes a major improvement in the modularity and extensibility of the web. The interfaces are explicitly identified on the wire, and this makes it easier for firewalls to do more discriminating filtering. Further details on this first lesson appear in the sections below.

Another lesson to draw is that it is not difficult to take great strides toward a deep unification of the web with COM, CORBA, and Java RMI. The unification of which we speak is not at the level of wire protocols --- where all three systems are hopelessly far apart now but their owners have shown considerable flexibility for the future --- but rather the abstractions with which application developers work. By a "deep" unification we mean the deployment of a system whose semantics essentially encompass those of the all unified systems; this may be contrasted to the more shallow unification achieved, for example, by having all those systems exchange their data encoded in XML over HTTP but without agreeing on common kinds of data or common encodings. Unification is discussed further in the section on the type system, where this issue chiefly appears.

The middle layer's design may be organized into three parts: (1) a type system for the data (including object values and references) passed in the calls, (2) a way of encoding those data in serial bytes, and (3) the call framing and other conventions needed to implement remote invocations in terms of opaque messages. We consider each of those parts in turn.

2.3.1. The Data Type System

The problems for the data type system to solve are: (1) be sufficient for good expression of TCWA as well as other applications being layered on HTTP, and (2) unify the type systems of COM, CORBA, and Java RMI. As this was only a feasibility study, the PDG was willing to overlook relatively obscure features of related middleware systems, and to explore advanced concepts that seemed to be on the upswing, on the grounds that the other systems might make such changes in the future. The HTTP-NG prototype's type system is described in [HFN98].

One lesson we drew from this design exercise is that a fairly conventional type system would indeed support the web application and other applications being layered on HTTP, with one important caveat. This conclusion is based on exploratory work on expressing the web application using the type system described here; this exploration is discussed more in section 2.4. The important caveat concerns support for decentralized evolution, which is important for Internet-scale applications like the web's. Supporting decentralized evolution requires considerable flexibility in the types used to characterize applications' expectations of data, because decentralized evolution introduces relatively complex patterns of changes to the data being exchanged. The best way to get such flexibility in a network interface expressed in a conventional type system is to use some form of property list as the type for a datum that may evolve in a decentralized way. Although this leaves the type system saying relatively little about the data in the property lists, it is a viable approach. For example, you see it in the application-level view of IPP. For another example, consider the popularity of dealing with XML through the DOM or SAX interfaces --- both of which present (all at once or serially, respectively) the XML document as essentially a fancy tree-structured property list. However, it is possible for a type system to be more involved in managing decentralized
evolution. The work on this was pursued somewhat independently and later, because it was not on the critical path to producing and measuring a prototype that captured the essence of the web application as it existed at one point in time. See [Spr99] for one example of how a type system might better support decentralized evolution, using a particular combination of ideas that are each familiar and in the spirit of strong object-oriented typing; that paper also briefly reviews what goes wrong when you try to use object subtyping to support decentralized evolution.

As mentioned earlier, another lesson we drew is that it is not difficult to make great strides toward unifying the type systems of COM, CORBA, and Java RMI. These type systems have significant overlap, and if you look carefully you can view their differences as mainly a matter of features being present in some of those systems but not others. Thus it is straightforward to unify them: offer a type system that includes every feature found in any of the systems to be unified. Because those systems were similar to begin with, this need not produce a terribly bloated result. However, it is important to avoid duplicating functionality that is packaged in different ways in those systems. This was done by unbundling features, taking a relatively orthogonalized view of those type systems and producing a relatively orthogonalized result. This is just another application of the principle of modularity. The PDG also ignored relatively obscure features that could conceivably be dropped in future revisions of the systems to be unified, as well as some that were simply not important to the application being prototyped. The experience reported suggests that taking an orthogonalized view could make it relatively easy to add any of these other features that are also desired. Following are some details of how the unification was achieved.

The biggest area in which the type systems to be unified contain differing packages of features is in types for "objects". Indeed, to proceed fruitfully we must even be careful about terminology. In CORBA, an object reference type is commonly called an "interface", and a "value type" describes objects passed by value. COM also has a concept of an "interface", and it is semantically close to a CORBA "interface". In COM there is also recognition of a larger unit of software organization, one of which generally has multiple "interfaces". Both the terms "object" and "component" are variously used for these larger units; a type for these units is known as a "coclass". In the Java programming language there are two kinds of types for objects: "interfaces" and "classes". In Java RMI, every type for objects passed by reference is an "interface"; both "interfaces" and "classes" may be used as types for objects to be passed by value. In this report, the term "object" is used consistently for the level of software organization known as an "object" in CORBA and Java RMI, and the term "object type" for a type for objects (both when typing objects passed by reference and objects passed by value).

One of the biggest areas of difference among the type systems to be unified is in the question of multiple inheritance for object reference types. In CORBA and Java RMI, object reference types can directly inherit from multiple supertypes. In COM, an "interface" can inherit directly from at most one other --- and user-defined ones tend to inherit from exactly one base, $\text{IUnknown}$. COM's $\text{IUnknown}$ "interface" addresses two areas of functionality: (1) reference counting of objects (COM's technique for memory management), and (2) navigation --- via the $\text{QueryInterface}$ method --- among the "interfaces" of a COM "component". Memory management is implicitly present for all object reference types in CORBA and Java RMI, and it is handled in a similar way (reference counting) for the programming language mappings in common with COM, so there is no real difference there. The HTTP-NG prototype allows multiple inheritance among object reference types, viewing COM's "interfaces" and "coclasses" as a limited usage pattern. The idea is that a COM "component" is taken to be an object passed by reference, and both COM "interfaces" and "coclasses" are types for such references, where multiple inheritance is used in just one way (to construct a "coclass" from multiple "interfaces").

Another area of difference is the support for passing objects by value instead of by reference. Java RMI supports this, and CORBA does too --- with an added feature, the ability to declare that a passed object may be truncated to certain of its supertypes (which might happen when the receiver has access to the code for
only some of the object's supertypes). COM does not directly support passing objects by value, although an application can use COM's custom marshalling feature to get that effect. The solution taken is to allow objects passed by value. For the sake of simplicity, the prototype design makes objects passed by value truncatable at every inheritance link. The prototype design does not address custom marshalling, as it was not needed for the prototyped application.

A further area of difference concerns whether pointer equality is preserved. Java RMI preserves equality of all pointers within a "serialization context", which for RPCs amounts to a call or reply message. CORBA preserves pointer equality among objects passed by value within a message, and nowhere else. In COM, preservation of pointer equality applies to some kinds of pointer types ("full" ones) and not others ("unique" and "ref" pointers). As the preservation of pointer equality is linked to the type of the data, these systems are easily unified by allowing explicit declaration of whether pointer equality is preserved for a given type of data. For the sake of type system modularity, this is broken out as part of a separate "reference" constructor: a pointer-equality-preserving type can be constructed from any "base" type; non-reference types don't promise any pointer equality preservation. In COM the scope over which equality of full pointers is preserved is larger --- an entire call. This is reported to be problematic enough that usage of full pointers is avoided. The prototype solution is to scope pointer equality as in CORBA and Java RMI, in hopes that COM may later adopt this preferred semantic.

A similar pattern of difference and solution occurs with regard to the issue of "optional" data. In CORBA and Java RMI, every object type is implicitly "optional", meaning that the null value is included in the type. However, in Java RMI arrays and strings are also implicitly optional, while in CORBA they're implicitly non-optional. In COM, some pointer types ("unique" and "full" ones) include null and others ("ref" pointers) don't. The solution is to make every type implicitly non-optional, except that the reference constructor can make optional types.

There is a similar, but different, story for network garbage collection. In COM and Java RMI, objects are implicitly subject to network-wide garbage collection; in CORBA, no network garbage collection is available. Again the solution is based on explicit declaration, but it is directly attached to the object types. Any object type that inherits from a certain designated type (HTTP-ng.GCCollectibleObjectBase) is subject to network garbage collection; others are not.

In addition to differing over the preservation of pointer equality, the type systems to be unified differ on the question of whether objects passed by reference have meaningful identities. In Java RMI and in the web, there is a strong notion of object identity. In COM and CORBA this is not emphasized --- but identities do appear in the implementations. While some workers have argued that there is no useful generic definition of object identity for distributed systems [Wat97], the problems raised in that argument can be viewed as saying only that a generic definition of object identity has limited --- but non-zero --- utility. For this reason, objects passed by reference have identities in the prototype design.

Another area of difference is in the handling of "charsets" [Alv98] for strings. In COM and CORBA, the charset of a string type is unspecified, and the charset of a string value is determined at runtime by "locale" and negotiated and possibly converted in the course of remote messaging. In Java, the "charset" of string types and values is fixed as the UTF-32 encoding of Unicode. The solution taken is to let the charset be unspecified in HTTP-NG string types and negotiated and possibly converted in HTTP-NG messages, letting Java peers be particularly hard-nosed negotiators.

One more area of difference concerns degenerate methods. COM offers "message" methods; these have no results, they raise no exceptions, and the caller does not normally wait for completion but proceeds immediately after issuing the call. Message methods are tied into Microsoft Message Queuing, and several controls over the message queuing are available. CORBA offers returnless exceptionless "ONEWAY" methods, whose distinction is that the call might not be reliably delivered; the issue of whether the caller may
proceed immediately is not explicitly addressed in the specification, but private conversations with CORBA developers reveal an appreciation for this concept. Java RMI offers neither. The HTTP-NG prototype allows returnless exceptionless methods to be "asynchronous", meaning only that the caller does not wait for a return.

A minor difference easily removed is the fact that only COM offers "callback" function parameters. These were simply omitted in the HTTP-NG prototype, on the grounds that callback functions can be treated as a special case of object references.

We now turn our attention to the problem of encoding data for network transmission.

2.3.2. The Data Encoding

The data encoding defines how data of types described above are encoded into bytes for transport, and does so in a way that addresses network performance problems and the rigid binding between object identities and transport stacks. Familiar techniques are entirely adequate to achieve these objectives. A binary data encoding was chosen, for the sake of efficiency in bytes on the wire and in processing time. The encoding is described in [Jan98], and can be considered an extension of XDR [Sri95]. The encoding of an object reference has a structure similar to that in CORBA; this structure includes separate places for the object's: (1) identity, (2) type information, and (3) contact information. Also like CORBA, the contact information is structured as a set of alternatives.

2.3.3. The RPC Protocol

The RPC protocol defines the formats of the messages exchanged to effect RPCs, and does so in a way that addresses modularity/extensibility and network performance. Again, only familiar techniques are needed to improve on HTTP and help unify the web with related middleware systems. The RPC protocol is also described in [Jan98].

The message formats facilitate modularity/extensibility by having an application-independent section of protocol extensions, analogous to "service contexts" in CORBA's GIOP.

The RPC headers in a request message improve network performance by using sender-managed tables to reduce the bytes needed to transmit common values. These apply both to the operation identifier and to the identifier of the object whose operation is being invoked. The sender may associate an identifier with a table index, and then use that table index in place of the identifier in multiple messages.

2.4. The Web Application Layer

The highest layer of the prototype design expresses TCWA as an application of the lower two layers. The group did not produce a full design here, but did prototype an indicative subset of HTTP's functionality [Lar98a]. The purpose of that prototype was twofold: (1) test the lower two layers against the needs of TCWA, and (2) enable measurement of an actual prototype. The lower two layers did indeed prove adequate, with the proviso noted earlier about the quality of the support for decentralized evolution. The prototype interfaces attempt to use subtyping to organize some of HTTP's extensions beyond its basic functionality; for this reason there are a number of interfaces of increasing sophistication. At the base we can start with a very simple version of the application, which could be rendered in OMG IDL as follows:

```idl
module TCWA {
    interface Resource {
        status_code GET (out Entity resp_ent);
    }
}
```
More sophisticated versions are rendered as subtypes with cloned operations with added parameters relevant to the various advanced features of HTTP, such as caching, content negotiation, and so on.

One particular development explored in the prototype interfaces shows that although one might not expect this from the RPC paradigm, it is possible to write interfaces that pass documents by streaming rather than all at once. When a document is to be returned by streaming, the client passes a callback object that is the consumer of the stream. The server repeatedly calls an asynchronous method of the callback, passing chunk after chunk of the document body. The use of asynchronous methods means each call simply amounts to a message passed from server to client. That message stream differs from a direct byte stream of document content mainly in the addition of message framing and a little other per-chunk overhead, all of which is small. The first call back passes a control object reference to the client, who can call on it to make the server pause or abort the message stream --- without having to close any connections --- or back up for error recovery.

3. Experimental Results

Although modularity and flexibility can have a negative impact on performance, the PDG produced and measured a prototype that shows it is possible to simultaneously improve performance and make the other reported improvements in HTTP. The comparison was between the prototype HTTP-NG protocols and both HTTP/1.0 and HTTP/1.1.

The main prototypes were built using a distributed object toolkit called ILU [ILU99]. This toolkit is implemented in ANSI C, and runs on top of either a POSIX UNIX API or a Win32/WinSock API, and supports the system of transport stacks present in the lowest layer of the HTTP-NG design. ILU also supports alternatives at the RPC layer, and the main comparisons were done using ILU implementations of the middle layer of HTTP-NG as well as the corresponding parts of HTTP/1.0 and HTTP/1.1. For the tests reported here, a network interface (NgRendering.NgRenderable [Lar98a]) that supports the caching and content negotiation features of HTTP was used, although those features were not specifically measured. The operation (GetRendering) that returns the result all at once was used.

To exercise these interfaces on the most basic use of the Web, fetching a document, the PDG created two ANSI C programs, "nggetbot" and "ngwebservice". "ngwebservice" acts as a very simple Web server, managing a file base. It exports the TCWA interface via a number of different wire protocols, including HTTP 1.0, HTTP 1.1, HTTP-NG, CORBA's IIOP 1.0, and Sun RPC. The program "nggetbot" acts as a testing client. It reads lists of URLs, and fetchs them. An optional delay can be specified for each URL, which specifies how long the client will wait before it fetches the next URL. The client can be directed to spread its fetchs across multiple concurrent threads; ten threads were used in the ILU-based tests reported here.

The tests were run on a small network consisting of two Sun Ultra-30 Model 250 computers, each having a 248 MHz UltraSPARC-II processor, 128 megabytes of memory, and running Solaris 2.6; two Compaq Deskpro 6000 computers, each having a 300 MHz Pentium II processor, 64 MB of memory, and running Windows NT (Service Pack 3); and a Xylan OmniSwitch switched-packet Ethernet router, using the ESM-100C-12W-2C fast Ethernet module. This network was connected to the regular PARC network, but the Xylan switch removed traffic not directed to the machines on the test network. The tests were mainly run on the two Sun machines, using one as a server and the other for the client programs. Some of the tests were
also duplicated using the Compaq machines, to check for problems having to do with endian-ness, and between the Sun and Compaq machines. The results reported here were measured on the Sun machines using the 100Mbit switched Ethernet.

This network is unflattering for the HTTP-NG protocols, which address concerns that are non-issues on this network. Because the network amounts to a direct connection between the client and server, there is no congestion to avoid, so it does not matter that the MUX filter improves the handling of congestion information. The network is very fast, so the number of bytes on the wire is not much of an issue, and round trips take very little time --- thus HTTP-NG's conservation of these things matters very little.

The primary test was the fetch of a single web page. This page had been developed as part of earlier performance testing work with HTTP 1.1 [HFN97]. Called the "Microscape site", it is a combination of the home pages of the Microsoft and Netscape web sites. It consists of a 43KB HTML file, which references 41 embedded GIF images (actually 42 embedded GIF images, but in our tests the image "enter.gif" was not included, as libwww's webbot did not fetch it as a result of fetching the main page). Thus a complete fetch of the site consists of 42 HTTP GET requests. Our base performance measures were the time it took to fetch this page, and the number of bytes actually transferred. We repeated this test with HTTP 1.0, HTTP 1.1, and HTTP-NG.

There are inherent differences in the protocol stacks tested. Some of particular note are in the use or non-use of batching, pipelining, and the MUX filter. Pipelining is the practice of sending a series of requests without waiting for intervening replies. HTTP/1.0 lacks this feature, but it was added in 1.1. It allows a single TCP connection to serve some of the needs that prompt use of multiple parallel connections in HTTP/1.0. With batching, transport buffers are not flushed until an overflow, timeout, or specific prompt forces them out, which means that several pipelined or multiplexed call or reply messages can go in one network packet.

<table>
<thead>
<tr>
<th>Test</th>
<th>Notes</th>
<th>time, seconds (std. dev.)</th>
<th>overhead bytes</th>
<th>TCP connections used</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP 1.1 (ILU)</td>
<td>+B, +P, -M</td>
<td>0.408 (0.012)</td>
<td>14,817</td>
<td>1</td>
</tr>
<tr>
<td>HTTP NG (ILU)</td>
<td>+B, -P, +M</td>
<td>0.306 (0.026)</td>
<td>7,935</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1

Table 1 compares the HTTP/1.1 and HTTP-NG protocols in the ILU-based test harness. The `Notes' column indicates whether batching, pipelining, and/or the MUX filter were used. The `time' column gives elapsed time. It is the average of 10 runs done in rapid succession. The bytes transferred and the number of connections used remained constant from run to run. The `overhead bytes' gives the result of subtracting from the total number of bytes transferred, in both the request and the reply, the constant size in bytes of the HTML document and the associated GIF images (a total of 171049 bytes); the remaining bytes consist of the data headers transferred with the requests and responses. The HTTP-NG protocols make a considerable improvement in both elapsed time and in overhead bytes.

<table>
<thead>
<tr>
<th>Test</th>
<th>Notes</th>
<th>time, seconds (std. dev.)</th>
<th>overhead bytes</th>
<th>TCP connections used</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP 1.0 (ILU)</td>
<td>-B, -P, -M</td>
<td>0.181 (0.003)</td>
<td>14,817</td>
<td>42</td>
</tr>
<tr>
<td>HTTP NG (ILU)</td>
<td>-B, -P, +M</td>
<td>0.194 (0.011)</td>
<td>7,935</td>
<td>1</td>
</tr>
</tbody>
</table>
Table 2

Table 2 shows better results for both HTTP/1 and for HTTP-NG. At least two effects probably contribute to the speed improvements. One is that ILU's batching functionality lacks a critical piece: there is no way for the server side to indicate it is done producing replies, and thus buffers occasionally languish until a timeout (the timeout used was 0.05 seconds). Another is that pipelining involves exclusive access to the connection on the client side at a higher level in the software than does multiplexing in the MUX filter or in the kernel's IP implementation, and we can expect less concurrency the higher the level at which access is exclusive. Discarding the problems associated with ILU's batching speeds up both HTTP/1 and HTTP-NG. The times for the two protocol approaches end up in the same ballpark; it's not yet known whether the small advantage of HTTP/1.0 is due to the use of many parallel connections and/or other factors. HTTP-NG could be run over parallel connections if desired. Recall that the network environment of these tests does not exercise some of the strengths of the HTTP-NG protocols. What Table 2 shows is that even in this unflattering network environment, HTTP-NG improves significantly on bytes on the wire and does not have a significant speed disadvantage.

<table>
<thead>
<tr>
<th>Test</th>
<th>Notes</th>
<th>time, seconds (std. dev.)</th>
<th>overhead bytes</th>
<th>TCP connections used</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP 1.1 (libwww/Apache)</td>
<td>+B, +P, -M</td>
<td>0.231 (0.003)</td>
<td>17,918</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3

To check that nothing was grossly wrong in the ILU-based test programs, the PDG also tested the "webbot" program from the libwww-5.1m release, together with the 1.2 release of the Apache web server. This gave comparison figures, reported in Table 3, for reasonably good HTTP 1.1 client and server implementations. Compared to the best result for HTTP-NG, this HTTP/1.1 result shows a larger elapsed time and larger overhead in bytes. However, there are some important differences between the two code bases, so the comparison is very rough. The webbot parses the fetched HTML and this in turn causes it to fetch the inline GIFs; the nggetbot is instead driven immediately from a list of URIs, and so has fewer delays built in. The webbot also produces more diagnostic output, which naturally costs elapsed time (even though it was directed to a file instead of a terminal). In the other direction, the ILU-based code is not particularly tuned for performance; this can be seen from the significant difference between the ILU-based 1.1 performance and the webbot/Apache 1.1 performance. While these factors affect the comparison of elapsed time, they do not affect the overhead bytes --- for which the HTTP-NG protocols show a clear improvement.

A more complete description of the tests, along with the data collected for the results, is available [Lar98b]. In this file, the HTTP 1.1 (ILU) test case is "Pipelined-HTTP", the HTTP NG +B test case is "Base-NG", the HTTP 1.0 test case is referred to as "Multi-connection-HTTP", the HTTP NG -B test case is "Batchless-NG", and the HTTP 1.1 (libwww/Apache) test case is "libwww-webbot". The code used for the tests is available as part of the ILU distribution [ILU99], in the directory ILUSRC/examples/ngtest/.
Finally, we have also talked about HTTP becoming a substrate for the carriage of popular distributed-object systems such as CORBA, DCOM, and Java RMI. To get an idea of how well that would work, we took the "test1" test case from the ILU system, and ran the ANSI C client from that test against the ANSI C server from that test, using both the CORBA IIOP 1.0 protocol and the HTTP-NG protocol. Again, the application code and application interface used in that test were identical; only the wire protocol and transport stacks were changed. This test is an artificial application designed mainly to utilize a number of features such as distributed garbage collection, floating point numbers, strings, object references, union types, arrays, and asynchronous method calls. Note that HTTP-NG supports all of those concepts directly, without any need for the `tunnelling' used, for example, in RMI over HTTP. We ran the test1 test 100 times in sequence, for each protocol stack, running the server on one of our Sun machines and the client on the other. This is a total of 1500 synchronous requests, with replies, and 100 asynchronous requests, without replies. Table 4 shows the results:

<table>
<thead>
<tr>
<th>Test</th>
<th>msec/call (std. dev.)</th>
<th>total bytes transmitted</th>
<th>TCP connections used</th>
</tr>
</thead>
<tbody>
<tr>
<td>test1 - IIOP 1.0</td>
<td>1.718 (0.598)</td>
<td>566876</td>
<td>3</td>
</tr>
<tr>
<td>test1 - HTTP-NG</td>
<td>1.631 (0.405)</td>
<td>170040</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4

The timings were roughly the same, and with a fairly high variance, but IIOP used significantly more bytes than HTTP-NG to make the same calls. These bytes in this case include both the data payloads of the calls and the request and reply header `overhead bytes'. Again, this should be regarded as indicative rather than definitive, but it's worth noting that the HTTP-NG protocol was carrying the same information as the IIOP protocol while using less than one-third the bandwidth. Much of this reduction is due to HTTP-NG's more efficient marshalling of object references. The reduction in TCP connections used is due to the ability in HTTP-NG of a server to `call back' to a client through the same TCP connection that the client used to talk to the server.

4. Future and Related Work

Further work can, and is, being done both on solving HTTP problems and on unifying the web with other middleware systems.

The Protocol Design Group attempted only to produce a prototype for use in studying feasibility. There are several areas in which the prototype design is only a preliminary guesstimate and could use improvements. This includes further work on simplifying and modularizing the middle layer (the designs for strings and exceptions are particularly preliminary). This also includes further work on making type systems better support decentralized evolution, and on integrating that work with the rest of the middle layer design. The unification of the web with other middleware systems would profit from further work on the middle layer of HTTP-NG as well as on the other systems, to help them all converge. A complete proposal for network interfaces for TCWA is also needed, as is the formulation of WebDAV as an HTTP-NG application.

Further testing to more fully explore the performance relations is warranted.
Other workers have taken on parts of the problem space addressed by the HTTP-NG PDG. The IETF's "Endpoint Congestion Management" Working Group [Ecm99] has already been noted. Others have undertaken to address part of the transport flexibility problem by adding a facility, known as "TCP Filters" [Bel99a, Bel99b], for dynamically negotiating transport stacks. This is limited to transport stacks built on TCP, and does not address the issue of decoupling stacks from resource identifiers. There has been a long history of trying to establish a standard for mandatory extensions to HTTP; the latest is known as the "HTTP Extension Framework" [HFN99].

The idea of unifying the web with other middleware systems has captured a lot of attention. For example, there is a series of related designs --- each more elaborate and specific than the preceeding --- for doing RPC via XML over HTTP: XML-RPC, SOAP, and Windows DNA 2000 [Msf99b]. However, these fail to address the underlying problems with HTTP, and make only a limited contribution to unifying COM with the other popular middleware systems (the biggest limit is that no attention is paid to object types from the other systems).

5. Conclusion

The Protocol Design Group produced a prototype that showed it is not difficult to make progress in all the addressed problem areas simultaneously, and that suggests unification of the web with COM, CORBA, and Java RMI is possible. Further work can, and is, being done on both solving HTTP problems and on unifying the web with other middleware systems.

Acknowledgements

The work reported above was a collaboration of many people. The other major contributors were Jim Gettys, Dan Larner, and Henrik Frystyk Nielsen. Daniel Veillard did the integration of the HTTP-NG prototype with Apache. Andy Begel worked on flexible typing issues. Paul Bennett, Larry Masinter, and Paula Newman were also valuable collaborators. Thanks are also due to Doug Terry for reviewing drafts of this paper.

References


Vitae

Mike Spreitzer works at the Xerox Palo Alto Research Center, in the areas of distributed systems, security, and programming languages. He is the current project lead for the PARC HTTP-NG project.

William C. Janssen, Jr., is a member of the research staff of the Information Systems and Technologies Laboratory at Xerox's Palo Alto Research Center. He is a principal of the PARC Inter-Language Unification project, which investigates different approaches to language-independent object-oriented program component systems, and former project
lead for the PARC HTTP-NG project. His research interests include object interface systems, distributed object systems, programming language design, and group coordination systems.