Data Types to Better Support Protocol Evolution

Mike Spreitzer □ (spreitze@parc.xerox.com)
Andrew Begel □□ (abegel@cs.berkeley.edu)

□ *Xerox Palo Alto Research Center*
□ *University of California, Berkeley*

Abstract

We show an approach for adding flexibility to data types to better support the kind of decentralized protocol evolution seen in Internet-scale development.˚ Once an application is widely deployed, development continues in a decentralized manner: multiple independent entities develop extensions to the application's protocols.˚ These extensions are then incrementally deployed into the existing system, with as much interoperability as the implementer can manage.˚ One way of defining protocols is as exchange of data in some defined type system, so the type system should support independent evolution of data producers and consumers as well as possible. A commonly suggested data typing technique for enabling this incremental deployment is to make new object types that are subtypes of existing object types. However, this technique does not lead to both forward and backward compatibility □ even though other techniques, such as the use of optional and ignorable items as in HTTP and XML, can yield compatibility in both directions.˚ Using property lists works, but is painful and significantly bypasses the type system.˚ We suggest a way to make data types that are more suitable for use in an evolving system. This approach adds the following to ordinary type systems: (1) extensible record types with moded fields, where the modes reflect the categories of evolutionary change supported, (2) *coarse* record types, for which extension is compatible with subtyping, and (3) record values that can evolve.˚ We exhibit this approach in the context of a simplified type system, and discuss how to marshal and unmarshal such data in invocation protocols. We then work some examples, drawn from the history of HTTP□s evolution, using our system. The examples are worked using an imaginary future version of CORBA and its mapping into Java.

1 Introduction

Protocols deployed on a wide scale (such as the Internet) evolve in a particularly challenging way.˚ In contrast with smaller deployments, the evolution of widely deployed protocols is not managed by a single engineering organization. Rather, many independent shops develop evolutionary changes to the protocol, and then incrementally deploy these changes into the existing system.˚ This poses interoperability problems, both singly and in combination.˚ The incremental deployment of a single evolutionary change presents the challenge of not only enabling new clients to work with new servers but also (at least in the cases where that makes semantic sense) enabling old clients to work with new servers and new clients to work with old servers.˚ The kinds of data typing seen in existing typed invocation systems such as CORBA [CORBA], DCOM [DCOM], and Java RMI [JavaRMI] have technicalities that disallow interactions that would semantically make sense; an extended example appears in the next section.˚ Worse yet, for popular and important protocols (such as the World Wide Web's HTTP [HTTP/0.9, HTTP/1.1]), at any given time there are several independently developed evolutionary changes in the process of being incrementally deployed.˚ This presents a further interoperability challenge.˚ In general, a given client and sever that wish to interact each support a different set of extensions, and it is desirable for their interaction to use the intersection of those sets.˚ Using existing data

type systems to do the necessary ad hoc negotiations makes for messy application code, and possibly additional latency; the example in the next section also illustrates these problems.

Not all protocol changes are evolutionary, and there are different kinds of evolutionary changes.˚ These issues are easiest to understand in terms of a contract between producer and consumer of a given type of data.˚ An RPC protocol bundles together several pairs of such contracts, one pair (domain and range, AKA arguments and results) per procedure in the protocol, and so is subject to the combination of the issues for each contract.˚ A single producer/consumer contract has two sides: constraints on the producer, and constraints on the consumer.˚ With respect to old/new interoperability, there are four possible kinds of evolutionary change, organized as two orthogonal interoperability choices. We define them as follows:

category of change	old consumer/ new producer	new consumer/ old producer
1. □free□	yes	yes
2. □backward□	yes	no
3. □forward□	no	yes
4. □mandatory□	no	no

The two orthogonal interoperability choices are: (a) whether an old consumer can interoperate with a new producer, and (b) whether a new consumer can interoperate with an old producer.

Since the "mandatory" category involves no interoperation across a change, why call it an *evolutionary* change at all?˚ It may make sense to do so when multiple mandatory changes can be used together; we show a completely decentralized way for parties to agree to use a given combination of mandatory changes, while traditional approaches require involvement of a global engineering organization (which necessarily adds significant overhead).

An additional semantic issue arises when considering combinations of extensions: are they semantically orthogonal?˚ Independently developed extensions may make changes in semantically overlapping areas; in such a case, it may not make sense to use two such extensions simultaneously, because applying the conjunction of the two changes would leave the semantics of the complete datum either under- or over-specified.˚ Indeed, to first order we can rely on the good sense of application implementers to simply not use combinations that don't make sense: producers won't produce combinations that don't make sense, and consumers won't be written to accept combinations that don't make sense.

This paper shows an approach to creating more flexible data types for use in application-level network interface (i.e., protocol) definitions.˚ These ideas could be applied to existing network interface definition languages, such as those of CORBA and COM (Java RMI would be nice to include, but is more problematic because one of its key virtues it its alignment with the semantics of the Java language).˚ Indeed, we use a modified version of OMG IDL to illustrate our solution.˚ This particularly relevant for use in the HTTP-NG [HTTP-NG] project, which is about factoring HTTP into pieces, one of which is a generic remote invocation system like those mentioned above.

The flexible data types that we show how to create are useful in more architectural contexts than the one suggested by "network interfaces".˚ In particular, this flexibility is valuable when the producer and consumer are separated by time or access control and thus unable to negotiate (which rules out certain alternative solutions).˚ This flexibility is also valuable when there isn't a single consumer for a given datum, for which negotiation-based solutions are (at best)

problematic.˚ Common examples include event distribution systems and database systems.

One particularly intriguing area of application is data typing for XML [XML].˚ Through the use of the ignore-what-you-don□t-understand design pattern □ refined to cover all four categories of evolutionary change □ XML is suited to the transport of evolvable data for widely deployed protocols.˚ There is significant interest in the XML community in defining a data type system for use with XML [XML-Data, XSchema].˚ However, applying traditional data typing approaches to XML yields the traditional problems, obscuring XML□s inherent ability to address evolution issues.˚ By applying the ideas of this paper, it should be possible to create data types that do not discard XML's inherent flexibility.

It may also be possible to create programming languages whose type systems use the ideas presented here.˚ That exercise is beyond the scope of this paper.

This paper is organized as follows.˚ In section 2 we examine the subtyping-based technique for incremental deployment, using examples drawn from the evolution of HTTP, and note that the subtyping approach doesn't scale well.˚ In section 3 we outline a solution based on property lists, which scales better but is messy to use and largely bypasses the type system.˚ In section 4 we outline our approach to making data types more flexible.˚ In section 5 we briefly consider how this approach affects remote invocation protocols.˚ In section 6 we return to the motivating examples from section 2 and work them using our new data types; we also show how to add proxying as an extension with our mechanism; along the way we use sketch and use some aspects of an imaginary future version of CORBA and its mapping into Java. We conclude with some brief remarks on future work in section 7 and with acknowledgments in section 8.

2 Evolution by Subtyping

Let us consider what happens if we try to use subtyping to facilitate protocol evolution. Suppose Tim Berners-Lee had tried to build the World Wide Web as a CORBA application.˚ He might have written the following instead of HTTP/0.9 (supposing 1998 CORBA were available at the time).

2.1 Example Base Protocol

```
// This is file HTTP.idl
module CH { module CERN { module WWW {
  typedef sequence<octet Bytes;
  struct Response {
˚˚˚ int resultCode;
˚˚˚ Bytes entityBody;
˚˚};
˚ interface HTTP {
˚˚˚ Response GET(string uri);
˚˚˚ };
}}};
```

2.2 Adding Content Negotiation

Many features were added to the web after HTTP/0.9.˚ We'll focus on two, content negotiation and caching, to illustrate the issues with incremental deployment of independently developed extensions.˚ For the sake of brevity, we will use simplified versions of these two extensions.

The content negotiation extension says there may be multiple variants for the current value of a resource, distinguished by MIME type. The client may, but is not required to, indicate its preference(s) for these aspects.˚ Because preferences are not required, this extension allows old clients to work with new servers (which, in the absence of an explicit `accept_types` value, make a plausible assumption about the client can accept).˚ Also, there is no guarantee whether or how a server interprets any preferences it is given □ enabling new clients to work with old servers.

We add the content negotiation feature to our CORBA expression of HTTP/0.9 as follows.˚

To requests, add an analogue of the Accept header, which gives the client's preferences for MIME types.˚ To responses, add an analogue of Content-Type header, which indicates the MIME type of the returned entity.

Suppose we use subtyping for extensions.˚ That is, introducing an extension involves creating a subtype of the HTTP object type, adding a new method with the extended functionality.˚ Here is how a company named Types2Go could have added simple content negotiation in such a style.

```
// This is file HTTP_Cn.idl
#include "HTTP.idl"
module com { module Types2Go {
struct Response_Cn {
˚˚CH::CERN::WWW::Response base;
˚ string˚          contentType;
};
struct TypePrefs {..};
interface HTTP_Cn
 : CH::CERN::WWW::HTTP {
˚˚Response_Cn GET_Cn(
    string uri,
    TypePrefs accept_types);
};
}}
```

Now, consider incremental deployment issues. Old and new servers and clients might be described as follows (in C++, to be specific).

```
CH::CERN::WWW::HTTP      old_srvr;
com::Types2Go::HTTP_Cn   new_srvr;
void old_clnt(
  CH::CERN::WWW::HTTP˚    server);
void new_clnt(
  com::Types2Go::HTTP_Cn  server);
```

There are four possible ways to combine these servers and clients.

```
old_clnt(old_srvr);˚ //works
new_clnt(new_srvr);˚ //works
old_clnt(new_srvr);˚ //works too!
new_clnt(old_srvr);˚ //doesn't work!
```

The last combination doesn't work because `old_srvr` doesn't have the type required by `new_clnt`. So we have to write clients in a more complicated style.˚ Here's the new client in that style:

```
void new_client(
    CH::CERN::WWW::HTTP server)
{
if (has_type(server,
          com_Types2Go_HTTP_Cn_type))
˚˚... narrowto_com_Types2Go_HTTP_Cn(
                        server)...;
else
˚˚...server...;
}
```

In addition to the scaling problems to be seen below, this type introspection may require round trips between client and server before useful work can be done. This adds latency that some other techniques do not have.

2.3 Adding Caching

Now consider what happens if caching is introduced independently of content negotiation. For the sake of brevity, we will consider a small subset of the caching functionality of HTTP/1.1.˚ Our caching subset works as follows.˚ The validity of a cached response is determined simply by a server-provided timeout, called the expiration time.˚ A cached response depends on some subset of the arguments to the `GET` call.˚ The response indicates which subset, so that a cache can use a cached response for as many requests as appropriate.˚ There is an efficient re-validation technique, based simply on timestamps.˚ Each response includes a timestamp, called its last-modified time.˚ A `GET` call may be modified by the inclusion of an "if-modified-since" time.˚ When such a parameter is present, and the response's last-modified time would be the same as the if-modified-since time, a shorter response is returned, saying simply that the response would have been the same as before.˚ Otherwise, the normal response is returned.

Following is the IDL for the independently-developed caching extension; suppose it is from a company named FastBits.

```
// This is file HTTP_Ca.idl
#include "HTTP.idl"
module com { module FastBits {
```

```
typedef .. Date;
  // details of (Date) not important

struct Response_Ca {
˚˚˚˚CH::CERN::WWW::Response base;
˚˚˚˚Date˚˚˚˚˚˚˚˚˚˚˚˚˚˚˚˚˚ expires;
˚˚˚˚Date˚˚˚˚˚˚˚˚˚˚˚˚˚ last_modified;
˚˚˚˚sequence<string>˚˚˚˚˚˚˚ vary;
    //(vary) holds the names of the
    // relevent parameters.
˚˚˚ };

interface HTTP_Ca
  : CH::CERN::WWW::HTTP {
˚˚˚˚Response_Ca GET_Ca( string uri,
˚˚˚˚˚˚˚˚˚˚˚      Date if_mod_since);
˚˚˚ };

}}
```

2.4 Content Negotiation + Caching

Now suppose an organization named Hakerz wants to develop a server that handles both content negotiation and caching.˚ They might write something like this:

```
// This is file HakerzWeb.idl
#include "HTTP_Cn.idl"
#include "HTTP_Ca.idl"
module org { module Hakerz {

struct Response_Cn_Ca {/*all the
request fields we have seen*/};

interface HTTP_Cn_Ca
˚˚: com::Types2Go::HTTP_Cn,
    com::FastBits::HTTP_Ca
  {
˚˚˚˚Response_Cn_Ca GET_Cn_Ca(
˚˚˚˚˚˚String˚˚˚˚˚˚˚˚˚˚˚˚˚˚ uri,
˚˚˚˚˚ com::Types2Go::TypePrefs
                        accept_types,
˚˚˚˚˚˚com::FastBits::Date
                        if_mod_since);
˚˚};

}}
```

... and here's what a client cognizant of that server would have to do:

```
void client_Cn_Ca(
  CH_CERN_WWW_HTTP server)
{
if (has_type(server,
          org_Hakerz_HTTP_Cn_Ca_type))
  ...narrowto_org_Hakerz_HTTP_Cn_Ca(
                              server)...;

else if (has_type(server,
           com_Types2Go_HTTP_Cn_type))
˚ ...narrowto_com_Types2Go_HTTP_Cn(
                              server)...;

else if (has_type(server,
           com_FastBits_HTTP_Ca_type))
˚˚...narrowto_com_FastBits_HTTP_Ca(
                              server)...;

else
˚˚...server...;
}
```

In general, a client that understands N extensions has source code of size 2^N, or even greater (see next point).

Suppose that a company named BrowCo, independently of the Hakerz, decides they want to implement browsers that offer both caching and content negotiation.˚ They write down their own IDL:

```
// This is file BrowCoWeb.idl
#include "HTTP_Cn.idl"
#include "HTTP_Ca.idl"
module com { module BrowCo {

struct Response_Ca_Cn {..};

interface HTTP_Ca_Cn
˚˚: com::FastBits::HTTP_Ca,
    com::Types2Go::HTTP_Cn
˚˚{
˚˚˚˚Response_Ca_Cn GET_Ca_Cn(
  ˚   string˚˚˚˚˚˚˚˚˚˚ uri,
˚˚˚˚˚˚com::FastBits::Date
                        if_mod_since,
˚˚˚˚˚˚com::Types2Go::TypePrefs
                        accept_types);
˚ };
}}
```

BrowCo writes their client in the four-case way shown above, but use their combined object type instead of the Hakerz's.˚ When aBrowCo browser visits a Hakerz server, they do not do both content negotiation *and* caching, even though both peers understand both extensions!˚

Even if BrowCo had happened to pick the same operation, parameter, and response member names and order as the Hakerz, the two object types would still be different (e.g., have different identities in a CORBA interface repository) in the independent development scenario.˚ The way to avoid this is for both BrowCo and the Hakerz to go to a global engineering body, such as the W3C or IETF, for a single global declaration of the combination of content negotiation and caching.˚ This imposes a significant additional delay in both development and deployment projects, simply to satisfy certain technicalities of the object system.

3 Workable Solution: Property Lists

The current ad hoc solution to supporting independent protocol evolution is the property list. In its straightforward application, it supports just one of the four categories of evolutionary change: free. A property list is a mapping of˚ name-value pairs, where the name is usually a string and the value is a data value with a tagged type. Each method in the interface is defined with extra input and output property list parameters. An evolutionary change adds an optional and ignorable name-value pair into those defined for a property list. A property list producer includes the name-value pairs for the extensions the producer supports and wishes to use; the consumer extracts from the property list the pairs that it supports. In this way the producer and consumer arrange to use the combination of extensions they both support, without explosive code growth or adding unnecessary latency.

This technique has been realized in a great many ways by a great many developers. It can be done using LISP S-expressions; it can be done with string-to-string mappings; it can be done with element structure in XML; and it is done with registered-number-to-encapsulated-data mappings in object references in CORBA.

This technique can easily be generalized to support all four categories of evolutionary change. By adding an explicit "user-level bit that indicates whether a pair must be understood, we enable the consumer to tell whether it understands all that the producer requires it to. That is, we make it possible to disallow interoperation for some new producer and old consumer pairs. It is even easier to disallow interoperation for some old producer and new consumer pairs: simply let the consumer check that certain new pairs are present and raise an error if they are not.

4 New Solution

A disadvantage of the property list solution is that it has nothing to say about how one can make static declarations e.g., data types that describe the particular features supported/expected by a given client or server.˚ Another disadvantage is that it leaves the programmer to write code to convert between the property lists and the data types his programs compute with.˚ We can solve both these problems by creating a flexible data type system that can express the constraints implicit in the property list solution.

This data type system differs from traditional ones in three ways. The new system:

1. has a particular form of extensible records that corresponds with the ways property lists are used to facilitate decentralized evolution (the unique feature of these extensible records is that their fields have modes that express various categories of evolutionary changes in the producer/consumer contract);

2. includes record values that are allowed to evolve, reflecting the reality of distributed systems; and

3. includes some unusually coarse record types, to cover evolutionary changes that would not otherwise create subtypes.

We will examine this solution in detail. To concentrate on the novel features, the examination will be set in the context of a drastically simplified data and type system; the ideas here should be applicable in fully fleshed-out systems too.˚ We will use a system that has just three kinds of data and types: records, procedures, and Booleans. Think of objects as appearing in our reduced system as extensible records with procedure-typed fields. For the sake of brevity, the system also disallows almost all possibilities of mutable data; the one allowed mutation is evolution of certain record values.

We will describe data and types in a simple graph-based way: each value and each type is simply a vertex in a graph.˚ Edges, some labeled, add structural details.

The formalism used here is unusually simple.˚ That is because we don't have to explain how lambda abstraction or application expressions (or their counterparts in complex programming languages) are typed; we are only concerned with describing data (including procedure values) that simply exist.˚ It is hoped that these ideas can be incorporated into the more complex type theories that can type general expressions.

In this system we will pay attention to three transitively closed relations between types: *subtyping*, *extension* (which corresponds to evolutionary changes), and *refinement* (which is the transitive closure of the union of the first two).˚ It is convenient to let these relations also be reflexive □ that is, every type is a subtype, extension, and refinement of itself. We also pay attention to one relation between values and types, the typing relation *has-type*.˚ These relations do not relate items of different kinds (e.g., no procedure has a record type), and so will be discussed in more detail on a kind-by-kind basis.

As this type system is for use in distributed systems, we need to pay a bit of attention to the marshaled representation of our data. Specifically, the subtyping relation needs to take into consideration not only abstract substitutability, but also substitutability of marshaled representations. So our system has the following property: a type *T2* is a subtype of *T1* if and only if:

- every value that has type *T2* also has type *T1*;
- each of those values, when marshaled as a *T2*, can be successfully and consistently unmarshaled as a *T1*; and
- when unmarshaling a *T2*, if the value at hand is in *T1* but not *T2* and was marshaled as a *T1*, this condition can be detected in a controlled way.

Here is a survey of the various kinds of values and types in the system.

4.1 Booleans

We start with Booleans, just to give some simple ground on which to build.˚ There is a value vertex that corresponds to the Boolean value `TRUE`, and another for `FALSE`.˚ There is a type vertex for the type of Booleans, and both Boolean values have the Boolean type.

4.2 Moded Extensible Records

A record value represents a set of enhanced name-value associations, roughly analogous to a property list.˚ Formally, a record value is a vertex with an outbound edge for each field, plus one more descriptive outbound edge.˚ Each field edge goes from the record value to the field value, and is labeled with (a) the ID of the field and (b) an 'ignorable' bit.˚ The ID of a field consists of two things: (1) the UID of a *fine* (see below) record type □ intuitively, the one that "introduces" the field □ and (2) a simple name, such as "entityBody". We use unique identifiers (UIDs) for record types: there is a one-to-one association between UIDs and record types. For

the sake of straightforward mapping into programming languages that have an analogous restriction, we don't allow two field edges of a record value to have the same simple name component.˚ The ignorable bit is set `FALSE` in fields introduced in the base version of a record type, and can have either value in fields introduced in extensions. The ability to mark an extension field as not ignorable corresponds to the "mandatory" feature whose lack has been a thorn in the side of HTTP evolution work.

A record value□s additional descriptive edge goes to a Boolean value that indicates whether this record value is one of those allowed to evolve.˚ If so, the value may from time to time take an *evolutionary step*.˚ Informally, an evolutionary step amounts to changing a record value to a refinement of its former type. Such a step corresponds to an evolutionary change in a resource in a distributed system; think of it as changing an object's implementation to a more refined one. Precisely, an evolutionary step consists of making changes in the field edges such that the record value (a) takes on types that are refinements of its former type(s) (and, of course, also takes on all the supertypes of those types), and (b) may lose some of its former types.

For example, the following figure shows the neighborhood of a record value vertex for a request that uses the content negotiation extension introduced above (see section 2.2) and expressed in our new system below (see Figure 2 and section 6.2.1).

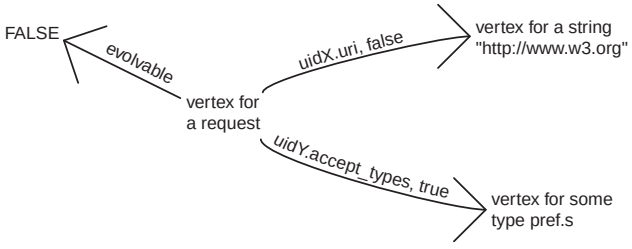

Figure 1: A Record Value

In this example we see a record value with two fields. One has ID `uidX.uri` and is ignorable. Here we suppose `uidX` is the UID of the fine record type that introduces the `uri` field; that record type is illustrated in another figure below. The other field has ID `uidY.accept_types` and is not ignorable. The record value in this example is marked as not being evolvable □ its field edges can not change.

This system has two categories of record types, which we call *coarse* and *fine*.˚ A coarse record type is a vertex with one outward edge, going to a fine record type that we call the coarse type's *leading* type.˚ A coarse record type roughly corresponds to the untagged union of its leading type and all the leading type's refinements. □Untagged□ means that there is no tag or □discriminator□ value involved: the values that have a coarse type simply *are* (except for complications due to the possibility of values evolving) the values that have any refinement(s) of the coarse type□s leading fine type. Coarse record types are particularly useful in distributed systems, where a client might have only stale information about a remote server□s current type (which could actually be a refinement of the type known to the client).

A fine record type is more like a □normal□ record type, in that it is defined not as a union of other record types but rather in terms of its fields. A fine record type is a vertex with an outward edge for each field. A fine record type also has an outward edge that indicates whether the type is *field-closed*. Each field edge goes from the record type to the type of one of the record type's fields, and is labeled with (a) the ID of the field and (b) the "mode" of the field.˚ As for record values, we disallow simple name collisions on a record type's edges.˚ Each field□s ID is among those used in the record type whose UID is in the field□s ID. That record type is said to be the one that *introduces* the field, which can also appear in other fine record types (possibly

associated with more refined field types). Modes are used to capture the aspects of the producer/consumer contract with respect to the existence and ignorability of the field.

There are nine field modes: Base, plus the cross product {Optional, NonOptional} X {DontLookIgnorable, AnyIgnorable, NonIgnorable, Ignorable}.˚ The Base mode is used in the record types of the initial version of a protocol; the other modes (called "extension modes") are used in extension fields (those added in protocol revisions).˚ The choice between Optional and NonOptional indicates whether the consumer requires the field to be present in all values of the record type.˚ The choice between DontLookIgnorable, AnyIgnorable, NonIgnorable, and Ignorable concerns the setting and reading of the "ignorable" bit, and is almost the cross product of two independent decisions: (1) what, if anything, the ignorable bit is constrained to be, and (2) whether the consumer will pay any attention to the setting of the ignorable bit.˚ When the "ignorable" bit is constrained to be `TRUE` or `FALSE`, the question of whether the consumer will pay attention to the setting is moot, so we use only four possibilities here:

1. DontLookIgnorable: the setting of the ignorable bit is unconstrained, and the consumer will not look at that bit;
2. AnyIgnorable: the setting of the ignorable bit is unconstrained, and the consumer may care about that bit;
3. NonIgnorable: the ignorable bit must be set to `FALSE`;
4. Ignorable: the ignorable bit must be set to `TRUE`.

The ability to declare that the consumer will not look at the ignorable bit looks out of place here.˚ Its usefulness appears when we consider mapping our type system into the type systems of existing programming languages, whose record values do not have "ignorable" bits; declaring an ignorable bit to be unimportant enables a faithful mapping to drop or mangle that bit.

Finally, a fine record type has a special outward edge that goes to a Boolean value that indicates whether refinements may add fields.˚ If they may not, we say the record type is "field-closed".

For example, the following figure shows the neighborhood of the vertices for the fine and coarse record types for the base and one extended request type; their IDL source appears in section 6.2.1.

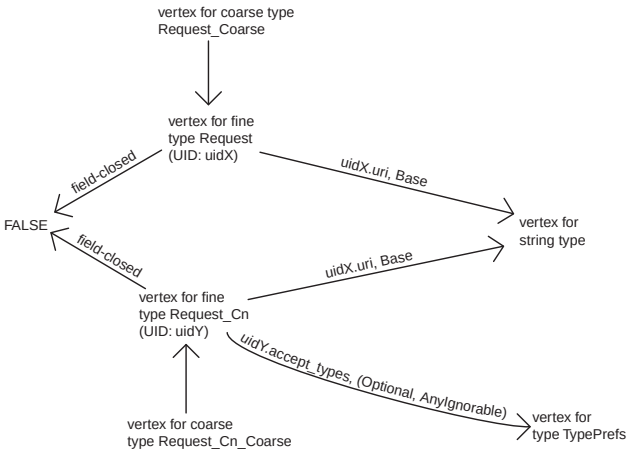

Figure 2: Some Record Types

In this figure we see two fine record types from section 6, `Request` and `Request_Cn`. We also see their corresponding coarse record types, `Request_Coarse` and `Request_Cn_Coarse`, respectively. `Request` and `Request_Cn` both have a field whose ID is `uidX.uri` and whose type is some string type. UID `uidX` is the UID of fine record type `Request`, which introduces the field ID `uidX.uri`. `Request_Cn` introduces an extension field, with ID `uidY.accept_types` and type `TypePrefs`.

In general, a record value "has" multiple record types.˚ This judgement is a bit tricky due to

evolvable record values: we cannot avoid the fact that whether a given record value has a given record type can change over time.˚ However, we can impose a certain monotonicity on those changes: once a given record value has a given record type, the value never loses that type.˚ We do this by saying that an evolvable record value never has a fine record type; the alignment between the structure of coarse record types and the allowable evolutionary changes leads to the desired monotonicity in the remaining possibilities.˚ To define the conditions under which an evolvable record value has a coarse record type □ as well as all the possibilities for a non-evolvable record values □ we use a time-dependent relation we call *conformance*.˚ At any given moment, a given record value (evolvable or not) conforms to a given fine record type if and only if the following criteria hold (for the purposes of these rules, a Base mode field is considered non-optional and non-ignorable):

1. each non-optional field of the record type is present in the record value;

2. each ignorable field of the record type is either absent from or marked ignorable in the record value;

3. each non-ignorable field of the record type is either absent from or marked not-ignorable in the record value; and

4. for each field that is in the record type and in the record value, the field value has the corresponding field type.

A record value (evolvable or not) has a coarse record type *U* if and only there exists a fine record type *T* that the value conforms to and that is a subtype of *U*.˚ A non-evolvable record value further has all the fine record types to which it conforms.

For example, the non-evolvable record value illustrated in Figure 1 conforms to both of the fine record types illustrated in Figure 2, and thus has all four of the record types illustrated in Figure 2. Furthermore, that record value also has fine record type `Request_Ca` from section 6.

Note that once an evolvable record value has a coarse record type *U* it will have *U* forever after.˚ This makes coarse types particularly useful with evolvable record values.

This definition is recursive over the structure of the data, and in general can lead to circular conditions.˚ The intended solution is the "maximum": the one with the most has-type relationships.

The subtyping relation among fine record types follows from the considerations of abstract and marshaled representation substitutability. Precisely, fine record type *T* is a subtype of fine record type *U* if and only if:

1. each field of *U* has a corresponding (same ID) field in *T*;

2. each *T* field that has a corresponding *U* field has a mode that strengthens the *U* field's and a type that is a subtype of the *U* field's;

3. if *U* is field-closed, then *T* is field-closed and all the fields of *T* have corresponding fields in *U*; and

4. every base field of *T* corresponds to a base field of *U*.

The "strengthening" relation among modes refers to "strengthening" the constraints on the producer and, conversely, relaxing the constraints on the consumer.˚ This relation is reflexive, and has additional relationships among extension modes.˚ One extension mode strengthens another exactly when corresponding components (the optionality and the ignorability) stand in a strengthening relationship.˚ The strengthening relations among extension mode components are reflexive, and also include the following relationships:

NonOptional strengthens Optional; AnyIgnorable, NonIgnorable, and Ignorable strengthen DontLookIgnorable; and NonIgnorable and Ignorable strengthen AnyIgnorable.˚ These relationships are the reflexive, transitive closure of those appearing in the following figure:

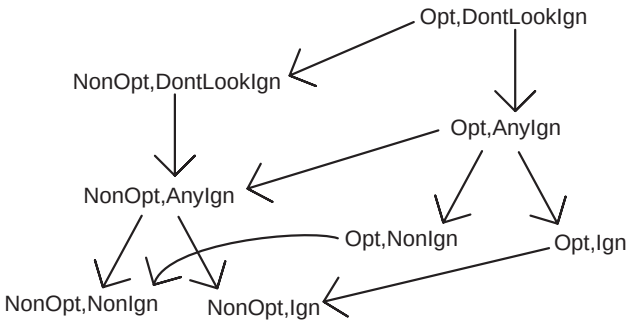

Figure 3: Field Mode Strengthening

Among fine record types, the extension relation is very similar to the subtyping relation; extension involves adding fields and extending existing fields. This relation differs from subtyping only because extension and subtyping differ for some possible field types, namely procedures. Precisely, fine record type *T* is an extension of fine record type *U* if and only if:

1. each field of *U* has a corresponding (same ID) field in *T*;
2. each *T* field that has a corresponding *U* field has a mode that strengthens the *U* field's mode and a type that extends the *U* field's type;
3. if *U* is field-closed, then *T* is field-closed and all the fields of *T* have corresponding fields in *U*; and
4. every base field of *T* corresponds to a base field of *U*.

For example, in Figure 2 the fine record type Request_Cn is both a subtype and an extension of fine record type Request.

The subtyping and extension relationships that involve coarse record types are as follows. The subtyping relationships of the coarse record types follow from their union structure. The subtypes of a coarse record type are the refinements of its leading fine type, and all their corresponding coarse types.˚ A coarse record type has no extensions (other than itself), so the refinements of a coarse record type are exactly its subtypes.˚ For example, in Figure 2 we have the following subtyping relationships involving coarse record types: Request, Request_Cn, Request_Coarse, and Request_Cn_Coarse are subtypes of Request_Coarse; Request_Cn and Request_Cn_Coarse are subtypes of Request_Cn_Coarse. The following figure illustrates a more complicated example.

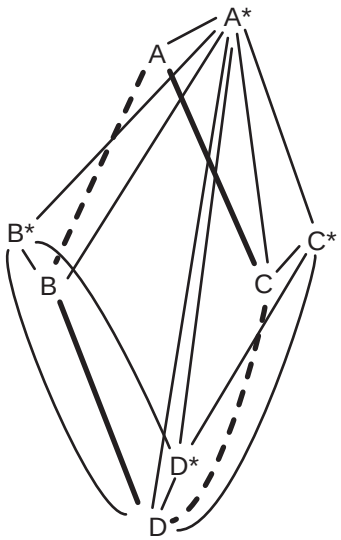

Figure 4: Example Coarse Record Type Relationships

The dashed lines show extension relationships, the continuous lines show subtyping relationships; in each case, the more refined type is below the less refined one.˚ The heavy lines show the subtyping and extension relationships among the fine record types: fine record type B is an extension, but not a subtype, of fine record type A; fine type C is a subtype, but not an extension, of A; and fine type D is a subtype of B and an extension of C.˚ Coarse

types A*, B*, C*, and D* have leading fine types A, B, C, and D, respectively.˚ The thin continuous lines show the consequent subtyping relationships between the coarse and fine types and among the coarse types.

The consumer of a record value should invoke a checking operation that tests whether the consumer has ignored any non-ignorable field in the record value. That operation takes the record value received by the consumer, plus a record type that characterizes the fields recognized and interpreted by the consumer, and tests whether the record value has any non-ignorable fields that are not mentioned in the record type.

4.3 Procedures

In this simplified data and type system, a procedure value has no internal structure; rather it is explicitly associated with a nominal procedure type.˚ That is, a procedure value vertex has exactly one outward edge, going to a procedure type.

A procedure type simply identifies a domain and a range.˚ That is, a procedure type vertex has two outward edges: one, labeled "domain", that goes to the domain type vertex, and another, labeled "range", that goes to the range type vertex.˚ Both the domain and range type vertices are fine record types, reflecting the usual structure of domain and range as a series of parameters (this restriction also avoids certain technical problems).

The subtyping and extension relations among procedure types are as follows. Procedure subtyping is defined in the usual contra-variant way: procedure type *B* is a subtype of procedure type *A* if and only if *A*'s domain is a subtype of *B*'s domain and *B*'s range is a subtype of *A*'s range. On the other hand, to serve the needs of protocol evolution, we make procedure type extension co-variant. Procedure type *B* is an extension of procedure type *A* if and only if *B*'s domain is an extension of *A*'s domain and *B*'s range is an extension of *A*'s range. It is this contrast between contra-variance and co-variance that causes the whole distinction between subtyping and extension throughout our type system.

A procedure value has the type with which it is explicitly associated, and every supertype of that type.

5 Remote Invocation Mechanism

Having introduced the data and types of our system, let us now turn our attention to how such data can be passed "on the wire" in networking protocols.

Sorting through the various ways of mixing remote and local data within a record is beyond the scope of this paper, so we add to our system the simplifying constraint that every record type is either purely data (passed by value) or purely procedures (i.e., an object passed by reference). We don't need to say much about object references. We may suppose that they contain some information about the type of the object, but that information may be outdated (i.e., the object may actually have a more refined type) by the time a client receives and acts on it.

It is thus the passing of data records that we need to address. To enable the smooth replacement of an object by a more refined object, and of a client by a more advanced (i.e., expecting a more refined server) client, we need the mechanism that passes domain and range data records to transform those records appropriately. The net effect of marshaling, transmission, and unmarshaling a given domain or range record should be to copy the record value from producer to consumer, either verifying that the record value has the type expected by the consumer or coercing (if possible, raising an exception otherwise) the value to the consumer-expected type. Following are the ways the coercion can fail.

1. The received record could lack a field required by the consumer.

2. The received record could have a field whose value does not have the type required by the consumer (because, e.g., the consumer is a subtype of the type the producer expected).

3. If the consumer expects a field-closed record type, any "excess" received fields are discarded — unless they're non-ignorable, which causes failure.

Failures of categories 1 and 3 can only involve extension fields, because neither subtyping nor extension can add or remove Base mode fields. So the main idea for how to marshal a record is this: first the Base mode fields are marshaled, without any identification or encapsulation, and then the extension mode fields are marshaled, each individually identified and encapsulated. The identification of the extension fields allows the receiver to pick out the fields it expects. The encapsulation enables excess fields to be discarded without the receiver getting lost in the stream, and also enables unexpected fields to be retained in cases (e.g., in a WWW proxy) where the receiver doesn't expect them but will pass the record on to some other receiver that might.

Here is how generated unmarshaling code can work. We start with a network interface described in terms of our type system. Each generated server-side stub is specific to a fine object (i.e., procedure-containing record) type. Consider how a domain record for a particular procedure is to be unmarshaled. Conceptually it is as follows.

1. Create a record value having all the received fields;

2. If the record value has the domain type of the procedure, proceed to invoke the procedure;

3. Otherwise, if the record value can be coerced to the domain type, proceed to invoke;

4. Otherwise, raise an exception.

This conceptual outline has the disadvantage that it might have to instantiate two records: the full received record, and the coerced record; it is possible to integrate these steps so that only one record needs to be instantiated.

In essence, we have folded some type checking functionality into the remote invocation mechanism. This is good because it lets us avoid the client-side type introspection that was shown in section 2 to be problematic. Instead, we can allow the client application to speculatively cast a stub to the server type that the client is prepared to deal with. This cast need not involve communication with the server, because the type check is deferred — it's integrated into the invocation protocol.

6 Examples

Now let us work some examples with our new system, showing how it facilitates protocol evolution without the problems outlined in section 2 but with the data typing lacking from the property list approach outlined in section 3.

For these examples we will use a hypothetical form of CORBA, created by adding the ideas from our system into CORBA 2.2 (chosen because 2.3 introduces "objects by value", which are a partial step in our direction). In particular, we suppose that in IDL, "structs" declare data-containing fine record types, and "interfaces" declare procedure-containing fine record types.

We will gradually introduce and work with some features of an example hypothetical mapping of this IDL into the Java programming language.

6.1 Base Protocol

6.1.1 The IDL for the Base Protocol

Here is how the initial protocol could be declared:

```
// This is file HTTP.idl
module CH { module CERN { module WWW {

typedef sequence<octet> Bytes;

struct Request {
    string uri;
    ...         //not field-closed
};

struct Response {
    int resultCode;
    Bytes entityBody;
    ...         //not field-closed
};

interface HTTP(_09,) {
    Response GET(Request req);
};
}}}
```

Here we've ended the structs with ... to indicate that they are not field-closed. The default mode for fields in these structs is Base; in structs that refine other structs, the default mode is (NonOptional, Ignorable). To keep this discussion short, for interfaces the default and *only* allowable field mode is (NonOptional, Ignorable).

Implicitly associated with each fine record type is a coarse record type. For two data record types we did not specify the name of the coarse record type, in which case it can default to something such as `Request_Coarse` and `Response_Coarse`. For the object type, we specified both the fine and coarse types' names with the "HTTP(_09,)" construction, which is a way of saying the fine type's name is "HTTP_09" and the coarse type's name is "HTTP".

6.1.2 Java Mapping of Base Protocol

Now let's consider how that IDL maps into Java. The fine record types `Request` and `Response` map to public Java interfaces named `Request` and `Response`, respectively, in package `CH.CERN.WWW`. We map a fine record type into a Java interface, rather than into a Java class, so that the Java mapping can enjoy some of the subtyping relationships that exist in our type system; using Java classes would unnecessarily restrict each mapped record type to having only one mapped supertype. However, our type system has many more subtyping relationships than can be mapped into Java, because Java does not have the same kinds of flexibility. In short, the rule is that subtyping relationships from our system do not map into Java; all typing operations (narrowing/widening, has-type testing, coercion, ignore-testing) of our type system become "user-level" operations in Java. There are a few defined exceptions to this rule.

To grant access to the `uri` field, the `Request` interface includes the following method:

```
java.lang.String uri();
```

We use an accessor method because Java interfaces cannot declare per-instance data slots. Similarly, the `Response` Java interface contains accessor methods for the `resultCode` and `entityBody` fields.

To support narrowing and widening, the generated code for `Request` includes a static method on an auxiliary class to narrow or widen a given record value to a `Request`:

```
public class _Request_Aux {..
static Request cast(
        org.omg.CORBA.DataRecord _r);
..}
```

This method accepts any mapped data record value; Java interface `Request`, like all the other mappings of record types, extends a Java interface named `DataRecord` that we suppose to be in package `org.omg.CORBA`. The `cast`

method tests whether the given value has the fine type `Request`, and if so returns a Java object that implements the Java interface `Request`; this object may be a copy of the given object, but because only non-evolvable record values can have the fine type `Request`, copying loses no information. If the given value `_r` does not have fine type `Request`, the `cast` method raises the standard runtime exception for casting failures.

The type testing above requires a generic way to access field values. That is done via methods of the Java interface `DataRecord`. They are:

```
Object _fieldValue(FieldID fid)
^^^^^^ throws FieldNotPresent;
boolean _fieldPresent(FieldID fid);
boolean _fieldIgnorable(FieldID fid);
```

The `_fieldValue` method returns the value of the requested field if it's present, throwing `FieldNotPresent` otherwise. Not all of CORBA's types map to Java subtypes of `java.lang.Object` – some map to Java primitive types; however, Java has defined wrapper classes to hold each of Java's primitive types, and `_fieldValue` returns those kinds of objects in those cases. The method `_fieldPresent` gives generic access to simply testing field presence, and `_fieldIgnorable` gives generic access to the ignorable bits.

How can `Request`'s `cast` method be implemented? Here's a sketch:

```
if (_r instanceof Request) return _r;
{String uri;
if (_r._fieldPresent(_uri_fid)
   uri = _r.fieldValue(_uri_fid);
else
  throw java.lang.ClassCastException;
return new _Request_Tie(uri, _r);
}
```

Here we use a generated class named `_Request_Tie`. This class implements `Request`, and has a per-instance data slot for the `uri` field. The class also has a per-instance data slot that holds the original record `_r`, and the generic access methods are delegated to `_r`.

The generated class `_Request_Aux` also has a static method for testing whether a given value has fine record type `Request`:

```
static boolean has_type(Record _r);
```

`has_type`'s job is a subset of `cast`'s, and the implementation is correspondingly simpler.

Finally, testing for ignored non-ignorable fields is a generic feature of `org.omg.CORBA` interface `DataRecord`; it is done via this method:

```
boolean _typeMissesNonIgnorable(
                                Type t);
```

This method takes as its parameter a Java object that implements a Java interface (named `org.omg.CORBA.Type`) that we've introduced for runtime representations of our types. Java interface `Type` provides user-level access to the type graph discussed earlier; the ignore-checking method tests that every non-ignorable field in `this` record value is mentioned in the given `Type`.

To make it easy for the application programmer to construct `Request` values, a convenience class is generated:

```
Public class _Request_
    implements Request
{
Public _Request_(String uri) {..}
..}
```

The convenience class `_Request_` has a constructor that takes all the field values (just `uri`, in this case) and initializes an object that implements the specific and generic accessor functions.

The mapping for `Request`'s corresponding coarse record type is a Java interface named `Request_Coarse`. It also extends `DataRecord`, has an accessor method for the `uri` field, and has type testing and type casting methods in an auxiliary class. They work very much like their counterparts for the fine record type.

The mappings for Response follow the pattern shown for Request. The mappings for HTTP and HTTP_09 show some differences, because procedure-typed fields are involved.

The mapping for fine record type HTTP_09 is a Java interface that extends org.omg.CORBA interface ProcRecord. As Java has no first-class procedures, the Java interface HTTP_09 includes method that invokes the procedure in the GET field:

```
Response GET(Request req);
```

The mapping for HTTP_09 also includes a server-side stub, as discussed in section 5.

The mapping for coarse record type HTTP is a Java interface that is named HTTP and extends org.omg.CORBA interface CoarseProcRecord. The HTTP interface also has a GET method:

```
Response_Coarse GET(Request_Coarse);
```

This method traffics in the coarsened versions of the domain and range record types, because they cover all the values allowed by all the possible refinements of the GET field.

Among the things generated for coarse record type HTTP is a client-side stub class:

```
public class _HTTP_client
  implements HTTP,
  org.omg.CORBA.ClientStub
{..
public static _HTTP_client(
              BindingInfo b) {..}
..}
```

The GET method of this class does the client side of the domain and range marshaling and unmarshaling discussed in section 5:

```
Response_Coarse GET(Request_Coarse r)
{ Response_Coarse ans;
// modulo exceptions...
  ..marshal r as a Request_Coarse..;
  ..wait for reply availability..;
  ..unmarshal a Response_Coarse
    into ans..;
  return ans;
}
```

As for data records, there is a static cast method on an auxiliary class:

```
public class _HTTP_Aux {..
public static HTTP cast(
                      ProcRecord _r);
..}
```

However, this cast method is the speculative cast method mentioned at the end of section 5. The interesting case in the speculative cast implementation is when the given _r is a client-side stub. In this case _r implements org.omg.CORBA interface ClientStub, through which _r□s binding information can be extracted and then used to instantiate an _HTTP_client.

6.1.3 Clients and Servers for HTTP/0.9

To describe clients and servers that only understand the HTTP/0.9 protocol, we can write IDL that uses field-closed record types:

```
struct Request_09 : Request { };
struct Response_09 : Response { };
interface HTTP_09_Only(_fine,)
 : HTTP_09 {
  Response_09 GET(Request_09 req);
};
```

In the declaration of fine record type Request_09, the □: Request□ means to inherit the fields from Request. The absence of □□□ from the braces makes Request_09 field-closed. Similarly for Response_09. The Java mapping of HTTP_09_Only includes this GET method:

```
Response_09 GET(String uri);
```

Because Request_09 is field-closed, none of its refinements can add fields, so the fields can be □spread□ into the arguments of GET

6.1.4 Client code for HTTP/0.9

Now, let□s look at how an HTTP/0.9 client can be written. First, the client gets binding information (e.g., a URI) for the server object from somewhere:

```
Org.omg.CORBA.BindingInfo bi = ...;
```

Then the client instantiates the coarse type of

the base protocol:

```
CH.CERN.WWW.HTTP stub
            = new _HTTP_client(bi);
```

Then the client speculatively casts the stub to the version the client is prepared to deal with:

```
CH.CERN.WWW.HTTP_09_Only stub_09
   = _HTTP_09_Only_Aux.cast(stub);
```

Or, alternatively, the client could have just instantiated a stub with the desired type to begin with:

```
CH.CERN.WWW.HTTP_09_Only
  stub_09
  = _HTTP_09_Only_client(bi);
```

Then the client can proceed to use the version-specific stub:

```
CH.CERN.WWW.Response_09 resp
  = stub_09.GET(uri);
```

6.1.5 Server code for Base Protocol

A server for the base version of the protocol can be written against the field-closed refinement:

```
class myServer implements
  CH.CERN.WWW.HTTP_09_Only_Fine
{ .. };

myServer s = new myServer(..);

..export s,
  publish its binding info..;
```

The `HTTP_09_Only_Fine`-specific server-side stub that is coupled with `s` will do the necessary domain and range conversion and checking, as discussed in section 5.

6.2 Content Negotiation Extension

Now let's look at how the content negotiation extension can be described and used.

6.2.1 The IDL

Here is the IDL for the content negotiation extension.

```
// This is file HTTP_Cn.idl
#include "HTTP.idl"
module com { module Types2Go {

struct TypePrefs {..};

struct Request_Cn
 : CH::CERN::WWW:Response {
˚˚˚˚Optional AnyIgnorable
      TypePrefs accept_types;
˚˚˚˚...
};
struct Response_Cn
  : CH::CERN::WWW::Response {
˚˚˚˚Optional AnyIgnorable
      string contentType;
˚˚˚˚...
};
interface HTTP_Cn
  : CH::CERN::WWW::HTTP {
˚˚˚˚Response_Cn GET(Request_Cn req);
};
}}
```

6.2.2 Java Mapping of Content Negotiation Extension

The Java mapping of the extension follows many of the same principles as the mapping of the base version. A notable new feature is that the Java interface `Request_Cn` extends the Java interface `Request`. This is one of the few places where a subtyping relationship in our type system is preserved in the Java mapping. This happened because `Request_Cn` explicitly inherited from `Request`, and the interface compiler (i.e., stub generator) can verify that `Request_Cn` is a subtype of `Request`.

Because the `accept_types` field of `Request_Cn` is Optional, the field-specific accessor can throw an exception due to absence of the field:

```
TypePrefs accept_types()
 throws org.omg.CORBA.FieldNotPresent;
```

Because the `accept_types` field's ignorable bit is not fixed, the Java interface `Request_Cn` provides an accessor method for the ignorable bit:

```
boolean _accept_types_ignorable();
```

The constructor for the convenience class for `Request_Cn` has extra parameters to cover the possibility of absence of the `accept_types`

field and of ignorability of the field:

```
_Response_Cn_(
    String uri,
    boolean _accept_types_present,
    boolean _accept_types_ignorable,
    TypePrefs accept_types);
```

Naturally, the `accept_types` parameter is not significant when `_accept_types_present` is false.

Suppose `HTTP_Cn_Only(_Fine,)` were defined, following the pattern set for the base protocol. The Java mapping of the coarse object type would include the following `GET` method:

```
Response_Cn_Only GET(
    String uri,
    boolean _accept_types_present,
    boolean _accept_types_ignorable,
    TypePrefs accept_types);
```

Spreading the `accept_types` field and its associated options yields three arguments, analogous to what happened to the constructor for the convenience class.

6.2.3 Client Code

Supposing `HTTP_Cn_Only` were defined following the pattern above, a client might look like this:

```
BindingInfo bi = ..;
HTTP_Cn_Only˚stub
          = _HTTP_Cn_Only_client(bi);
Response_Cn_Only resp˚= stub.GET(uri,
  true, true, accept_types);
```

This client interoperates with the server for `HTTP_09_Only`. The marshaled form of the domain record contains: the `uri` field, with no explicit identification or encapsulation, followed by the `accept_types` field, marked as ignorable and explicitly identified and encapsulated. When the `HTTP_09_Only_Fine` server-side stub unmarshals that record value, it will discard the `accept_types` field, and invoke the `HTTP_09_Only_Fine` implementation. Conversely, for the range record, the client-side stub will take the marshaled form of a `Response_09` and produce a `Response_Cn_Only` in which the `contentType` field is absent.

6.2.4 Server Code

The server code is similarly direct:

```
class myCnServer
  implements HTTP_Cn_Only_Fine
{..};

myCnServer s = new myCnServer(..);

..export s,
  publish its binding info..;
```

This server interoperates with the `HTTP_09_Only` client. The relevent optional fields are ignored or taken to be absent, as in the case of an `HTTP_Cn_Only` server interoperating with an `HTTP_09_Only` client (but with the forward/backward directions switched).

6.3 Basic Caching Extension

Now let□s look at how caching is introduced as an independent extension. Here is the IDL for the caching extension.

```
// This is file HTTP_Ca.idl
#include "HTTP.idl"
module com { module FastBits {
typedef .. Date;
struct Request_Ca
  : CH.CERN.WWW.Request {
˚˚˚˚Optional AnyIgnorable
      Date˚˚˚˚˚˚˚˚˚˚˚ if_mod_since;
};
struct Response_Ca
  : CH.CERN.WWW.Response {
˚˚˚˚Optional AnyIgnorable
      Date˚˚˚˚˚˚˚˚˚˚˚ expires;
˚˚˚˚Optional AnyIgnorable
      Date˚˚˚˚˚˚˚˚˚˚˚ last_modified;
˚˚˚˚Optional AnyIgnorable
      sequence<string> vary;
};
interface HTTP_Ca
    : CH.CERN.WWW.HTTP_09 {
˚˚˚˚˚˚Response_Ca GET(Request_Ca req);
};
}}
```

The Java mapping and client and server code follow the patterns seen for the content negotiation example. Note that these clients and servers interoperate with the unextended and the content-negotiating clients and servers, because both extensions added fields that are optional and ignorable; the mechanisms we've discussed implement the optionality and ignorability efficiently behind the scenes.

6.4 Combinations

Now let's look at how to create clients and servers that support a combination of independently developed extensions.

6.4.1 Hakerz

First, let's look at the interface and code developed by the Hakerz.

```
// This is file HakerzWeb.idl
#include "HTTP_Cn.idl"
#include "HTTP_Ca.idl"
module org { module Hakerz {
struct Request_Cn_Ca
  : com::Types2Go::Request_Cn,
    com::FastBits::Request_Ca
{...};
struct Response_Cn_Ca
  : com::Types2Go::Response_Cn,
    com::FastBits::Response_Ca
{...};
interface HTTP_Cn_Ca
: com::Types2Go::HTTP_Cn,
  com::FastBits::HTTP_Ca
{
    Response_Cn_Ca GET(
                  Request_Cn_Ca req);
  };
}}
```

The Hakerz IDL constructs `Request_Cn_Ca` by inheriting fields from both `Request_Cn` and `Request_Ca`; `Request_Cn_Ca` is both a subtype and a refinement of those other fine record types. Similarly for `Response_Cn_Ca`. The clients and servers of this IDL interoperate with all the previous clients and servers, for the same reasons as seen in the previous cases.

6.4.2 BrowCo

Now we turn our attention to an independently-devloped combination, from BrowCo.

```
// This is file BrowCoWeb.idl
#include "HTTP_Cn.idl"
#include "HTTP_Ca.idl"
module com { module BrowCo {
struct Request_Ca_Cn
  : com::FastBits::Request_Ca,
    com::Types2Go::Request_Cn
{...};
struct Response_Ca_Cn
  : com::FastBits::Response_Ca,
    com::Types2Go::Response_Cn
{...};
interface HTTP_Ca_Cn
 : com::FastBits::HTTP_Ca,
   com::Types2Go::HTTP_Cn
{
    Response_Ca_Cn GET(
                  Request_Ca_Cn req);
  };
}}
```

The clients and servers of this IDL interoperate with all the others seen earlier, again through the behind-the-scenes implementations of optional and ignroable fields. While BrowCo's `Request_Ca_Cn` type is not the same vertex as the Hakerz's `Request_Cn_Ca` type, the two types are subtypes of each other – that is, every value of the one type is also a value of the other. This means that BrowCo clients interoperate with Hakerz servers, and vice versa.

6.5 Proxying

Using the techniques introduced here, we can add proxying as an extension to HTTP/0.9:

```
struct Request_Py : Request {
    Optional NonIgnorable HTTP orig;
};

interface HTTP_Py : HTTP_09 {
    Response GET(Request_Py req);
};
```

An `HTTP_Py` server is willing to either serve its own content, when passed an unextended `Request`, or delegate to another server, `orig`, when supplied in a `Request_Py`. By marking

the extension field as non-ignorable, we ensure that the client will see a claim of success only if the proximate server understood that it was being asked to act as a proxy.

7 Future Work

Supporting concurrent development and incremental deployment of multiple independent evolutionary changes to a protocol that is based on explicit interfaces with typed data is an important problem, and using subtyping does not solve it as well as we would like. Using property lists avoids the problems of subtyping, at the cost of largely bypassing the type system. We have shown an approach to making type systems flexible enough to describe the relevant constraints of the property list solution. We have worked some examples that validate our approach.

Several things remain to be done. One item of high importance is to fully work this approach out in a fully fleshed-out type system. In particular, it would be good to actually integrate it into the type system(s) of CORBA, DCOM, and/or HTTP-NG.

Another important task is to incorporate ordering information among the fields in an extensible record. In so doing, we make this approach usable not only for sets of orthogonal extensions, but also for sets that semantically interfere to the degree that can be untangled by simply specifying an ordering for the extension usages. Experience with HTTP shows that this would be a useful enhancement.

8 Acknowledgments

This work has benefitted from discussions within the HTTP-NG project and with colleages at Xerox PARC. In particular, we would like to thank Henrik Frystyk Nielsen, Jim Gettys, Bill Janssen, Dan Larner, John Lamping, Larry Masinter, and Dan Swinehart.

9 References

[CORBA] Object Management Group. Common Object Request Broker Architecture. http://www.omg.org.

[DCOM] Microsoft Co. Distributed Component Object Model Protocol (DCOM/1.0). http://premium.microsoft.com/msdn/library/techart/msdn_dcomprot.htm. January 1998.

[JavaRMI] Sun Microsystems Co. Java Remote Method Invocation Specification. http://java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html.1996.

[HTTP/0.9] Berners Lee, T. HTTP/0.9 Specification. http://www.w3.org/Protocols/HTTP/AsImplemented.html. 1991.

[HTTP/1.1] IETF HTTP Working Group. HTTP/1.1 Specification. http://www.w3.org/Protocols/rfc2068/rfc2068. January 1997.

[HTTP-NG] World Wide Web Consortium. HTTP-NG Project home page. http://www.w3.org/Protocols/HTTP-NG/.1998.

[XML] World Wide Web Consortium. Extensible Markup Language. http://www.w3.org/XML/. 1998.

[XML-Data] Andrew Layman et. al. XML-Data Specification. http://www.w3.org/TR/1998/NOTE-XML-data/. January 1998.

[XSchema] XML-DEV mailing list. XSchema. http://www.simonstl.com/xschema/. 1998.