Using Event-Based Translation to Support Dynamic Protocol Evolution

Nathan D. Ryan and Alexander L. Wolf
Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430 USA
{ryannd,alw}@cs.colorado.edu

Abstract

All systems built from distributed components involve the use of one or more protocols for inter-component communication. Whether these protocols are based on a broadly used "standard" or are specially designed for a particular application, they are likely to evolve. The goal of the work described here is to contribute techniques that can support protocol evolution*. We are concerned not with how or why a protocol might evolve, or even whether that evolution is in some sense correct. Rather, our concern is with making it possible for applications to accommodate protocol changes dynamically. Our approach is based on a method for isolating the syntactic details of a protocol from the semantic concepts manipulated within components. Protocol syntax is formally specified in terms of tokens, message structures, and message sequences. Event-based translation techniques are used in a novel way to present to the application the semantic concepts embodied by these syntactic elements. We illustrate our approach by showing how it would support an HTTP 1.1 client interacting with an HTTP 1.0 server.*

1. Introduction

The overarching goal of this work is to enable dynamic communication protocols, which is to say, to allow a distributed application to continue functioning even when the communication protocols used by its distributed components have changed. For our purposes, a protocol for distributed communication is considered to be a form of application-level message passing. In our current work we are not concerned with network-level protocol issues, such as routing and forwarding.

Traditionally, an application uses a protocol by having close ties to the *syntactic details* of the protocol. Here, syntax refers to the structure of messages and the rules governing the coordination of messages passed among multiple components. Thus, a rule stating that a "reply" message must follow a "get" message is considered as much a part of the protocol syntax as is the textual format of each "reply" and "get" message.

The embedding of the protocol's syntactic details into the code of an application means that changes to the protocol might force alterations to the application in order to accommodate those changes. Even small changes or additions introduced into a protocol—such as the redefinition of a token, an appended message structure, a new timeout specification, or a supplemental acknowledgement message for coordination—can have drastic implications for an application, possibly forcing the application to be redesigned, modified, and rebuilt according to the new protocol specification.

Instead, we would like the application to be concerned with the *semantic concepts* encapsulated by the protocol, rather than its syntactical details. A semantic concept (or simply "concept") represents an element of the data or behavioral logic of a component interaction. A common example is the concept of "date", which can have many syntactic manifestations. Another example is the concept of "request" in an HTTP protocol interaction; the fact that different version of the protocol represent this concept in different syntactic forms should be largely irrelevant to an HTTP client or server component.

Typically, however, there is nothing available to decouple semantic concepts from syntactic details on behalf of an application, so the application either must itself extract the concepts from the syntax or must rely solely on the syntax as a representation of concepts. Either way, the behavior of the application is tightly coupled with the syntactic details of the protocol. Therefore, if the protocol is altered—whether or not the syntactic change reflects a true concept change—the application cannot continue.

Clearly, this issue of syntactic and semantic separation has been long recognized in various communities, and a variety of technologies have been proposed to address the problem. For example, the database community has explored the problem of multi-database integration, and has

Proceedings of the 26th International Conference on Software Engineering (ICSE'04)
0270-5257/04 $20.00 © 2004 **IEEE**

offered the technology of what are called *mediators* as a solution [21]. Mediators serve to perform automatic translation of data and data requests, either to and from a common universal schema, or on a database-to-database pairwise basis. In the software architecture community, the latest attempt at solving the problem comes in the form of the *connector* [8], which is an architectural element whose purpose is to encapsulate inter-component communication. The idea is that replacing one connector by another to effect a protocol change should be possible, even at run time.

The distinction between syntactic detail and semantic concept is not always easy to discern. To some extent, it is a matter of exercising the proper discipline in the design of the application. Our goal is to provide tools and techniques that encourage the designer to make this distinction explicit. The incentive that we offer is the ability to develop components that can transparently accommodate certain kinds of protocol changes.

The approach we take is a novel use of *event-based translation*. The idea behind event-based translation is to generate "events" as syntactic structures are parsed. This technique is demonstrated to some degree by the XML SAX parser [15], which generates events as the syntax of an XML document is parsed. Each event captures some concept represented by the XML syntax of the document, and presents it to the application in the form of a method call belonging to a handler object. Inspired by the XML SAX parser, we apply this basic idea to achieve the separation of protocol syntax from protocol concepts, as suggested in Figure 1.

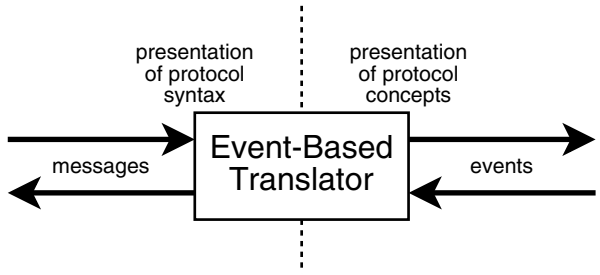


Figure 1. Separating Syntax from Concepts.

An event or sequence of events can embody an abstraction of one or more syntactic elements, specifically the semantic concept or concepts associated with those elements. This means that concepts of the protocol are captured by the events produced by the parser and given to a component, isolating the component from the syntax of the protocol. The event-based translation technique can be generalized and formalized so that an event-based translator can be constructed on demand. If the protocol specification changes, a new event-based translator can be constructed to replace the old one, and the application can continue to receive the events in which it is interested.

We stress that there are limits to the nature of changes that can be accommodated by our approach; those that involve deep semantic changes will obviously require changes to the components themselves. Moreover, there is some development-time and run-time cost to adopting the approach. As to the first point, our initial experience indicates that there is a sufficiently rich set of changes that can be handled by our approach and, perhaps more importantly, that these changes can be found in the real-world evolution of popular protocols. For example, HTTP 1.1 introduced a new message, the "status 100" continue-to-wait message, a change to the protocol that can, through our technique, be made transparent to an HTTP 1.0 client. As to the second point, we are faced with a classic tradeoff of performance (both programmer and program) against flexibility. We have interposed a level of indirection in the form of an event-based translator, which requires some amount of programmer effort to develop and some amount of computer effort to execute. At this stage in the research, we have only early indications of the intellectual and computational overheads involved in this tradeoff.

In the next section we discuss related work. Following that we present the details of our approach. We then illustrate the approach through an example, that of an HTTP 1.1 client attempting to communicate with an HTTP 1.0 server.

We have implemented a prototype of the event-based translation system and applied it to the HTTP example. This prototype represents a first vertical slice through the whole problem and has served us well in guiding the refinement of the approach over the past six months.

2. Related Work

The ideas presented here can be seen as a particular approach to developing a dynamic connector among components, where the interface between a component and a connector is modeled as events. In that sense it is related to work in the area of software architecture and the treatment of connectors as "first-class objects". Garlan points out that "allowing complex connectors provides a single home where one can talk about the semantics", also noting that "[one] could attach a single description of the protocol of interaction to the complex connector" [8, page 113]. The same viewpoint is adopted here. We perceive the protocol to be the semantics of an arbitrary connector that is capable of being dynamically swapped with another connector. Figure 2 shows this viewpoint, where the black boxes attached to each component indicate the coupling of the connector with the component.

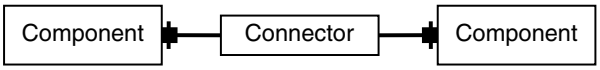


Figure 2. Connector Coupled to Components.

The mechanics of realizing such a vision, however, are not straightforward. Garlan notes that "even simple interactions, such as message sending or procedure calls, become complex in their own right in most distributed settings" [8, page 113]. To make the separation between component and connector possible, the connector must be able to present some sort of coupling interface to its attendant components.

Proposals for such couplers already exist. Jones, Romanovsky, and Welch [11] developed an approach based on the notion of *integrators*. Integrators are used as the coupling between connector and component, and are attached to each component with "glue code", thus providing a known interface to clients. Unfortunately, this approach requires a separate integrator for each interface/protocol pair, similar to early program translation methods, which required translation networks that included a unique translation between each pair of programming languages. For programming languages, this problem was simplified by the introduction of abstract languages and algebras [2, 5]. However, for the method proposed by Jones, Romanovsky, and Welch, which focuses on a fault tolerance mechanism with multi-layered exception handling, each integrator (essentially the translator) must be specifically designed with a particular protocol and interface in mind; no unified intermediary is proposed.

An important approach is taken in the architecture description language Wright [1]. Wright connectors use *roles* to define the behaviors involved in an interaction, where the connectors themselves describe the behavior of the connection. The roles are coupled with *ports* associated with the components. The language used to describe the behavior of the interactions is a variant of CSP [10], a powerful (largely symbolic) protocol description language that is accompanied by a wide variety of validation tools.

Although the Wright model is general enough to be used in almost any situation, it still relies on the notion of a protocol-specific coupling between components and connectors, as embodied in roles, ports, and the "glue" that binds them. If a component cannot match the specific image of the protocol that is presented to it through the definitions of the various roles and ports, the connector cannot be coupled with the component even if the roles and ports conceptually present the same information to which the component is accustomed. Figure 3 depicts the potential for such a mismatch.

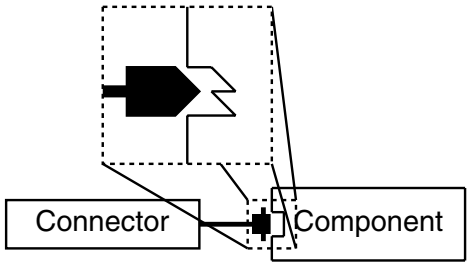


Figure 3. Mismatched Coupling.

The approach described here uses a mechanism that loosens the requirements for the coupling between a component and a connector, in a sense providing a "universal plug" for a connector to be coupled with a component. Conceivably, any connector can then be coupled to any component, regardless of the semantics associated with the connector. Therein lies the primary difference between most existing techniques and the one described here. Current realizations of the component/connector approach define mechanisms that rely on a strong specification of the interface provided by the connector and require that the component be strictly matched with the interface, whereas our approach provides a technique that generalizes the coupling. Of course, this flexibility comes at a price, and that price is the ability to predict *a priori* the compatibility of the connector and component simply by examining the interface. We must appeal to some outside mechanism to guarantee, or at least check, on compatibility.

This seeming conflict in approaches can be resolved if we treat a mechanism such as that found in Wright as a *design* aid and treat our approach as an *implementation* aid. There is no reason why a strict model of the component/connector interface could not be used in conjunction with a faithful, albeit looser implementation of that interface.

The C2 architectural style and its supporting infrastructure take a similar approach to strictness and flexibility [19]. Strictness is achieved through an external mechanism that is part of the C2 development environment. At run time, flexibility is achieved through a common event bus to which arbitrary components can connect, disconnect, and reconnect. C2 differs from our approach in that it has adopted a universal protocol for all component interactions, which happens to be based on events. In contrast, we are trying to capture a variety of specific protocols, and use events at a different conceptual level, namely the boundary between the protocol and the components.

Given that our focus is on communication protocols for distributed systems, it is important to briefly explain more broadly the relationship of our work to that of the

many tools and techniques available to perform protocol specification, such as the general-purpose specification languages CSP [10], LOTOS [20], and Esterel [4, 6], as well as the architecture description languages Darwin [14, 17], Rapide [12, 13], and Wright [1]. As we explain in the next section, our approach relies heavily on the formal description of a protocol. Theoretically, we could use any of the popular protocol specification languages. The choice among them reduces to convenience, such as availability of tools and simplicity of conception, more than any other factor. For example, Esterel is a full-fledged programming language for describing signal processing in reactive systems. As such, it appears to be too heavy weight for our purpose. We simply need a means to specify the format of messages and the coordination rules among messages. For the former, traditional grammar specification languages suffice. For the latter, some sort of state machine specification language is suitable.

We turn now to event-based translation. For that we have looked most closely at the XML SAX parser [15] in its three versions. However, it is a parser for a document structure rather than a communication protocol, and it is specific to the XML syntax (rather than to the syntax of documents specified by a DTD or schema), which uses a restricted grammar. To date the XML SAX parser has served as a good model, but eventually we will have to create a more general form of event-based parsing.

Another reason to look beyond the XML SAX parser is that we need a tool that can also perform the complement of parsing, namely composition. Composition provides the channel from a component into the protocol, which explains our use of double-headed arrows in Figure 1. It also explains our use of the more general term "translation", rather than just "parsing". XML serves as a model here as well. Various tool sets, including the DOM parser, allow one to generate an XML document, but only after the elements supplied to construct the abstract syntax tree of the document can be validated against the relevant DTD or schema. These validation methodologies are the basis for a technique suitable for our purpose here.

With both parsing and composing in mind, we must also be able to automatically generate the translator. Compiler compilers have existed for decades, and their theory is well known. Again, however, this is primarily for what one might consider a document (usually a program text) and not a protocol. Such systems must be adapted to support the full range of features necessary for generating protocol translators. Fortunately, such systems do exist. The protocol compiler developed by Castelluccia, Dabbous, and O'Malley [7] is an example of a compiler that produces automata from an abstract protocol specification written in Esterel. This work is promising, not only because it demonstrates the feasibility of such a technique, but also because it indicates that automatically generated translators can be efficient as well. In our prototype we use JavaCC (available at http://javacc.dev.java.net) to automatically generate translators particular to a specified protocol.

3. Technical Approach

The general approach we have developed to support dynamic protocol evolution consists of two main elements: a three-part specification of protocol syntax, and an event-based technology for translating from the specified syntax into semantic concepts (and vice versa). The three-part specification serves as a means to conveniently modularize the different aspects of a protocol that might evolve, while the event-based translation system, whose behavior is driven by the syntax specification, presents semantic concepts as abstract events. The relationship between syntactic elements and semantic concepts is currently established through a simple mapping from the elements to a set of concept names, although one could easily imagine using instead some arbitrarily sophisticated ontologic scheme.

3.1. Specification of Protocol Syntax

We treat a protocol generically as a form of application-level message passing. Most well-known protocols for distributed communication have a precise definition for the structure of a "message", and at least an informal description of what constitutes "passing". If we assume that messages can be viewed structurally as documents (a reasonable assumption), then we can use document description techniques to specify the structure of messages as tokens, as well as to specify compositions of tokens, that is, the format of a message.

However, this is clearly not the complete definition of a protocol. Message structure specifications say nothing with regard to the rules of coordination among messages, and so additional information is necessary. We refer to these rules as an *interaction specification*. Note that we require several specifications for each interaction of a protocol, one specification for each role a component may assume while using the protocol. Further, these specifications must include information that affects the way in which messages are delivered, such as transport bindings, reconstruction of partial messages, special timeouts, and the like. These last aspects of a protocol are in a sense orthogonal to the main goal of our work and are not currently addressed, but their influence on a protocol must be acknowledged.

Thus, we have a three-part syntax specification. Figure 4 depicts the "uses" relationship (solid arrows) among these elements. Also shown is that the syntax elements refer to concept names (dashed arrows).

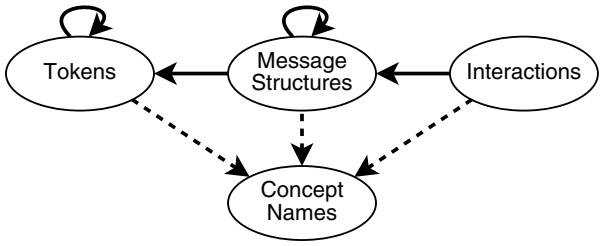


Figure 4. Protocol Specification.

Given this specification framework, a description language must be chosen (or developed) to satisfy each of the three parts. A suitable, although not necessarily distinct, description language should be assigned to each part, where we take "suitable" to imply at least the following four criteria: (1) sufficient expressiveness; (2) compact representation; (3) easy (visual) manipulation; and (4) isolation of changes.

A balance must be kept among these criteria when assigning description languages to each of the three parts. Clearly, some important conflicts exist among the criteria. Additionally, conflicts may arise among the description languages chosen for each part of the specification framework, due to the dependencies among the parts (noted in Figure 4). For example, since the specification of message structures depends on the specification of tokens, the presentation of tokens must be compatible with their use in the specification of message structures.

In our current work, we have adopted, not surprisingly, basic grammar languages for tokens and message structures, since the expressive power required by token specifications is generally regular, while the expressive power required by message structure specifications is generally context free. Message structures do require some amount of context-sensitive specification such as, for example, the common situation that arises when one message structure element is used to indicate how many of another message structure element appears in the message (e.g., the value of the "Content-Length" HTTP header indicates the length of the included entity body). Thus, for message structures, we use description languages inspired by those found in JavaCC.

For the interaction specifications we currently use a simple language of state machines with guarded transitions. Note, however, that the choice of language features to capture information describing the way in which messages are delivered is problematic, since issues such as the recomposition of messages from possibly out-of-order communications and transport bindings do not immediately lend themselves to a simple state-machine language. Thus, in the future, we may add special language features for this aspect of the specification.

3.2. Event-Based Translation

The primary elements of our approach consist of the three-part *syntax specification*, described above, and an event-based *translation system* that, in turn, provides a *negotiator* responsible for negotiating protocols and for generating/managing event-based *translators* to operate on specific versions of the evolving protocol. A particular *distributed component* interacts with the translation system through *protocol events*, which are either derived from or are composed into *protocol messages*. Figure 5 shows these elements in relation, illustrating how the interposition technique allows a component to be isolated from the syntactic details of the protocol.

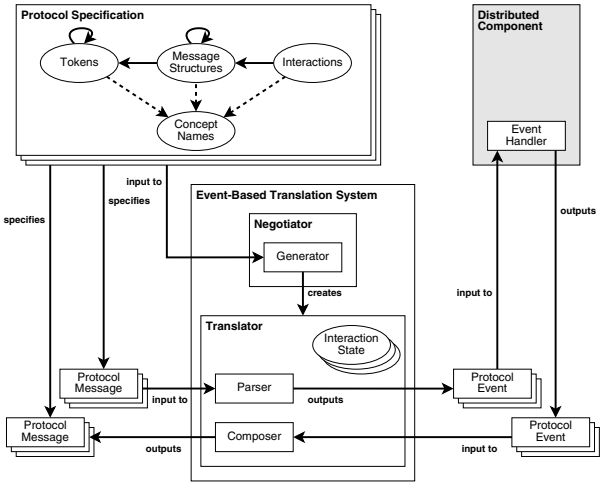


Figure 5. Event-Based Translation.

The negotiator element of the event-based translation system typically collaborates with the corresponding negotiator for another component when the two distributed components first initiate communication. It is then the responsibility of the negotiators to determine which protocol will be used, given the preferences and capabilities of the relevant applications. If the negotiator determines that a protocol will be used for which a translator does not exist, it will direct the generator element to create one, obtaining the protocol specification from the other negotiator, if necessary.

The protocol specification is used as input to the generator element of the event-based translation system. The generator creates a translator that recognizes the specified protocol. The translator is responsible for deconstructing protocol messages via the parser, constructing protocol mes-

sages via the composer, and maintaining state for each protocol interaction.

Protocol messages are input to the parser from some source component engaged in the communication. The parser deconstructs the message and outputs the corresponding semantic concepts as a series of events. As the events are generated, they are delivered to an *event handler* in the distributed component. Once delivered, it is up to the component to decide how to interpret the events.

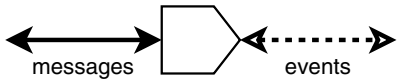
For the return path, the component gives a set of concepts to the composer, which uses those concepts to construct a legal message (or legal messages) of the protocol. That message is then sent to some target component, which in general may be different from the source of the protocol message originally passed to the parser.

Notice that the application must cooperate with the translator, in the sense that it incorporates an event handler capable of processing the events generated by the parser, as well as being able to give appropriately encapsulated protocol concepts to the composer.

3.3. Alternative Configurations

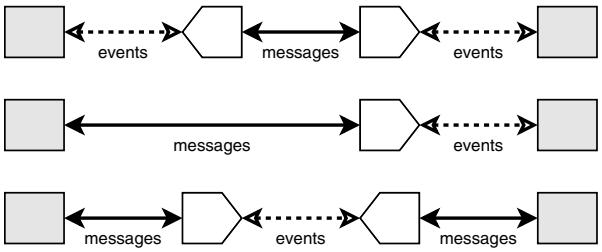
As described, it would appear that our technique requires a component to cooperate directly with the event-based translation system. In fact, that is just one way to make use of our approach. We are also able to apply the technique without requiring modification to pre-existing application components. In such cases, the components will communicate using the full protocol, and the role of the event-based translation system is reversed.

An event-based translator can be thought of architecturally as a bi-directional pipe, with messages of the protocol coming in and going out one side, and protocol concepts (as events) coming in and going out the other, as shown here.



The solid line indicates message communication, and the dotted line indicates event communication. Note that a translator is specific to a single protocol.

Such a translator can then be connected to an application component in two different ways, depending on whether the component is "aware" of the event-based translation system or not. Given any two distributed components that are acting as the end points of a communication, there are then four distinct configurations possible, effectively falling into three categories: both communicating components are aware, only one component is aware, or neither component is aware. We depict the categories as follows.



The first case is the ideal situation, since the protocol being used is not fixed by either distributed component. All message parsing, message composition, and interaction-state tracking is performed by the translators, so each component can be dynamically connected to different translators, and each message of the protocol is composed only once and parsed only once. Note that the translators must both be specific to the same protocol. The respective negotiators (not shown) will determine the protocol to use and direct the corresponding generators (also not shown) to dynamically create the translators if necessary.

In the second case, the left-hand component is dependent on the syntax of the protocol, but the right-hand component is not, and can therefore be dynamically connected to different translators. Again, each message of the protocol (communicated in either direction) is composed only once and parsed only once. The translator must be specific to the same protocol on which the left-hand component depends; no negotiation takes place, and the left-hand component is connected directly to the translator as if it was connected to the right-hand component. (An analogous situation occurs when the left-hand component is aware and the right-hand component is not, and thus we have not bothered to depict this case.)

The third case deals with the situation in which neither component is aware of the event-based translation system. In this case, the aware "component" to which each translator is connected is, in fact, another translator. The protocol that each component uses is fixed; the advantage this situation has over direct communication between the components is that the protocols used by each component can be different, yet the components can still communicate. The disadvantage is that, since potentially two different protocols are being used, each communication requires composition and parsing in both the protocol used by the right-hand component and the protocol used by the left-hand component, effectively doubling the amount of translation work required. In practice, both translators would exist on one side of the distributed application, so that event communication is local. Again, no negotiation takes place.

In the next section we present an example that embodies the first configuration.

4. Example

In this section we present an example application of our approach to dynamic protocol evolution. The scenario, implemented using a prototype of the event-based translation system, involves an encounter between an HTTP 1.1 client and an HTTP 1.0 server, the goal being to show how the client and server can, via our prototype, communicate using their respective versions of the protocol. For details of the HTTP 1.0 and HTTP 1.1 protocols used for this example, see RFC 1945 [3] and RFC 2068 [9], respectively.

Normally, an HTTP server that understands only HTTP 1.0 will have difficulty handling an HTTP 1.1 request. This is because servers are currently designed to manage such a request in a way that is heavily tied to the syntax of the protocol. Thus, despite the fact that the two versions of HTTP are remarkably similar, even small changes in syntax have a serious impact on compatibility. On the possibility that the syntax would be misunderstood, HTTP 1.0 servers typically were programmed to deny all requests that did not specify exactly an HTTP 1.0 version number, irrespective of the later clarification of HTTP version numbers in RFC 2145 [16]. In theory, however, many (if not most) HTTP 1.1 requests should be accommodated by an HTTP 1.0 server, since the syntactic differences are minor, and are almost exclusively additions to HTTP rather than modifications or deprecations.

The syntactic differences between HTTP 1.0 and HTTP 1.1 span the three parts of our syntax specification, including interaction specifications. For example, a difference in the tokens is the length of a target URL, limited to 100 characters in HTTP 1.0 but unlimited in HTTP 1.1. A difference in the message structure is the introduction of the "Host" header in HTTP 1.1, required for all requests but not even defined in HTTP 1.0. And, even though both versions of HTTP are thought of as single-request/single-response protocols, a difference in the interactions is that HTTP 1.1 contains the addition of "status 100" (i.e., continue) responses, an arbitrary number of which can precede the actual response to the request. All three of these differences are considered syntactic under our approach.

Of course, there are many other differences between HTTP 1.0 and HTTP 1.1, some of which are truly semantic ("concept") changes requiring modifications to HTTP 1.0 servers. Our aim here is not to address HTTP evolution specifically, but rather to illustrate our approach through a well-known protocol and, thus, without having to describe the details of some unknown protocol.

Our example assumes that the client and server are in the first configuration described in Section 3.3. This means that the client and server are both written to operate on protocol concept events rather than on raw HTTP messages. The client communicates through an event-based translator using HTTP 1.1 concepts, while the server communicates, initially, through a translator using HTTP 1.0 concepts. In our implementation of the example, the client and server are simulated, in the sense that they each only communicate with their corresponding translator via a set of events limited to the minimum required for the most basic HTTP messages, performing no real HTTP-like processing, such as returning a page of HTML text in response to a `GET` request. The translators are generated from specifications of HTTP 1.1 and HTTP 1.0. These specifications, which are not shown here but are available elsewhere [18], use the description languages discussed in Section 3.1.

The essence of the scenario, depicted and numbered in Figure 6, and elaborated more fully below, is as follows: The client sends (1) events to its translator, which requests (2) the address of the server's HTTP 1.1 translator from the client's negotiator. The client's negotiator requests (3) this information in turn from the server's negotiator. The server's negotiator, finding that no HTTP 1.1 translator exists, requests that one be generated, using the specification given by the client's negotiator. (The specification conceivably can be obtained from elsewhere, such as at a given URI, and while an interesting aspect of the approach, it is not central to the current discussion.) An HTTP 1.1 translator is generated (4) for the server, and its address is returned (5,6) to the client's translator via the client's negotiator, which caches the address. The client's translator composes an HTTP 1.1 `GET` request using the information given to it by the client, and the request is sent (7) to the server's HTTP 1.1 translator for parsing. While the message is parsed, events are forwarded (8) to the server, indicating the concepts encapsulated by the message in the context of the interaction. Although not shown in the figure, an analogous process is used to send an HTTP 1.1 "status 200" response message to the client. No negotiation is required in this case, since the path is already established.

For this HTTP interaction (and, in fact, most HTTP interactions), only three states are required: one state prior to the request being communicated, one state after the request is communicated and prior to the response being communicated, and one state after the response is communicated. Since we have two roles (client and server), we must have two specifications for the interaction. Figure 7 shows the graphical representation of the interaction specification for both the client role (top) and the server role (bottom), respectively. Although in this case the two specifications are quite similar, this will not be true in general, especially where there are multi-way interactions among distributed components.

We now provide a bit more detail about the actions taken during the scenario execution. Interpreting the specification

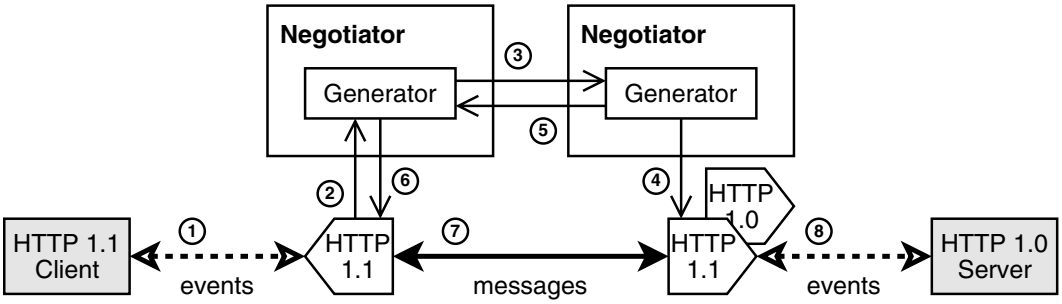


Figure 6. An HTTP Application Scenario.

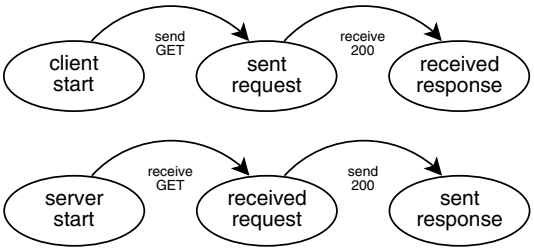


Figure 7. Portion of Interaction Specification.

of HTTP 1.1, we find that the minimal `GET` request will require the following events from the client:

```
[01] target {
       HOST = "research.cs.colorado.edu"
       PORT = 80 }
[02] token METHOD_GET
[03] token URI = "/"
[04] done
```

The events are numbered according to the order in which they are given to the translator, although only the position of the last is important. Event 1 gives the host and port number of the server's negotiator. Notice that, since we are using the first of the configurations described in Section 3.3, the well-known port number for HTTP, port 80, is assigned to the server's negotiator, while the translators are assigned dynamically determined port numbers. This allows the server to service two versions of HTTP at the same well-known port. In contrast, the other two configurations would require two different port numbers, one for each version of HTTP.

Event 2 indicates to the translator that it will compose a `GET` request; a parameterized value is not necessary, since it is implicit. Event 3 indicates the target URI of the request, and event 4 indicates that the client has no more information for the translator.

The translator requests from the negotiator the address (host and port) for a corresponding HTTP 1.1 translator on the server. Since the negotiator does not have such information, it contacts the server's negotiator using the given host and port. The server's negotiator also does not have the information, since no server-side HTTP 1.1 translator exists. The server's generator is therefore instructed to create one, using JavaCC. If the generator does not have the specification for HTTP 1.1, then it can request the specification from the client's HTTP 1.1 translator, via the negotiators. Regardless, the specification is used to generate a new translator, and the address at which the translator is listening for incoming messages is communicated back to the client's HTTP 1.1 translator.

The client's HTTP 1.1 translator then composes the following HTTP 1.1 `GET` request, using both information given by the client and defaults determined from the protocol specification:

```
GET / HTTP/1.1
Host: shield.cs.colorado.edu
```

The message is sent to the server's HTTP 1.1 translator. The client's translator gives the client an event, "`interaction request_sent`", noting the state change in the data structure used to represent the interaction state from the client's perspective. As the server's HTTP 1.1 translator parses the protocol message, the following events are generated and given to the server:

```
[01] token METHOD_GET = "GET"
[02] token URI = "/"
[03] token VER_HTTP = "HTTP"
[04] token VER_MAJOR = "1"
[05] token VER_MINOR = "1"
[06] token VER = { VER_HTTP, VER_MAJOR,
       VER_MINOR }
[07] structure GET_LINE
[08] token HEADER_HOST_NAME = "Host"
[09] token HEADER_HOST_VALUE
       = "sidhe.cs.colorado.edu"
```

```
[10] token HEADER_HOST = {
       HEADER_HOST_NAME,
       HEADER_HOST_VALUE }
[11] structure GET_HEADER
[12] structure GET_HEADER_LIST
[13] structure GET_REQUEST
[14] interaction request_received
```

The events are numbered according to the order in which they are generated, and labeled by the part of the protocol from which they result. For example, event 1 is a token event that indicates the recognition of the `GET` method keyword. Note that some token events contain references to nested token events, to indicate the composite nature of some tokens. Event 7 is a message structure event that indicates the recognition of a line in a `GET` request. The six prior events indicate the recognition of tokens that constitute the line. Event 14 is an interaction event that indicates a change in the interaction state from the server's perspective.

The server will likely not understand events 8 through 10, since they are concepts outside the server's context of HTTP. (Recall that the server is specific to HTTP 1.0 and has not been modified to accommodate HTTP 1.1 concepts.) Regardless, all events will be sent to the server's event handler, and the server may choose to ignore them or process them as desired; in the case of our simple server, the unrecognized events are ignored, and thus the concepts associated with the "Host" header will be filtered out.

The server's HTTP 1.1 translator is now finished parsing the `GET` request, and the last event should have indicated to the server that it has been provided a collection of concepts that comprise a full HTTP message. This completes the process of sending an HTTP request from the client to the server. All construction and parsing was handled by the translators, so that the client and server did not need to be concerned with protocol syntax and were only required to handle events that represented concepts of the protocol.

The next step is for an HTTP response to be sent from the server to the client, which would follow an analogous process. No negotiation is required for the return path, since the connection between the translators is still open. Further, the negotiation that took place before the request was sent is a one-time cost, given that negotiators can cache the address of local and remote translators. Note, too, that the creation of the server's HTTP 1.1 translator is also a one-time cost; it need not be generated again for any other HTTP 1.1 messages the client, or even a different HTTP 1.1 client, might send.

The configuration of translators and other components in this example is just one possible way of employing our approach. For example, one could imagine an implementation that coalesces the functionality of some of these elements, making the deployment of the implementation perhaps a less daunting prospect. Exploring these options is one thread of our future work.

5. Conclusion

Enabling dynamic protocol evolution has significant benefits for distributed computing. Aside from the obvious—an application need not be shut down and reconstructed each time there is a protocol update—one such benefit is that an application could potentially process two different versions of the same protocol simultaneously. It could thus avoid a "chain update" problem, which may not have a solution for a given system configuration. Dynamic distributed applications (i.e., distributed applications whose components can be exchanged or whose architecture can be altered at run time) may realize additional benefits, since each component need not have pre-written compatibility for every possible combination of communication protocol. Finally, there is also potential benefit beyond dynamic protocol evolution, such as dynamic protocol negotiation and discovery (i.e., inter-component communication without prior knowledge of protocols).

The success of the example presented in Section 4 clearly relies on the similarity of HTTP 1.0 and HTTP 1.1. But this is to be expected. Only radical changes should have radical effects, yet today even small changes can have radical effects. Our motivation is to better align the effort to make a protocol change with the significance of that change. For example, consider a simple but useful update that should be easily introduced between versions of the HTTP protocol, where the specification of the request line is changed to make it consistent with the specification of the status line—that is, we want to move the HTTP version indicator from the last position to the first. Exactly the same events would be generated for this specification, merely in a different order. However, since under our approach the server is unconcerned with the order of these particular tokens, it would not be impacted by this modification, and the request would go through normally. Today, such a simple change would likely confuse virtually every current implementation of an HTTP server.

On a higher level, completed component interactions, as identified in the state machine maintained by a translator, could themselves be considered "messages" of a larger protocol. The specification of this larger protocol might include a timing restriction, such that the single continuous interaction permits only so many of these messages over a predefined duration. If the limits of the specification were exceeded, the translator could generate an error event indicating this. Coupled with other information in the messages

(e.g., the target URI of an HTTP request, the status code of responses, and elements of a "Date" header), this could make denial-of-service attacks detectable earlier. Further, the specification of this larger protocol could be dynamically updated with new heuristics for such attacks.

Of course, more remains to be done to fully achieve our goal of supporting dynamic protocol evolution. For one, we would like to complete a formalization of event-based translation to permit the automated construction of the event-based-translator generators from the information available in the specifications. We need to introduce a means of correlating protocol versions or otherwise similar protocols. We would like to explore the range of possible methods for employing our techniques in a variety of protocol evolution scenarios. In order to accomplish this, we must add functionality to the existing prototype, including the generalized composition of protocol messages and the management of context-sensitive message structures. Finally, the various configurations discussed in Section 3.3 imply quite different performance and productivity overheads and tradeoffs. We need to understand how these play out for a wide range of usage scenarios.

Acknowledgements

We would like to thank Kenneth Anderson, Antonio Carzaniga, Amer Diwan, Dennis Heimbigner, and William Waite for their helpful comments on this work.

This work was supported in part by the Defense Advanced Research Projects Agency under agreement number F30602-01-1-0503. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

References

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[2] J. Backus. The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. In *Proceedings of the International Conference on Information Processing*, pages 125–132. UNESCO, 1959.

[3] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol—HTTP/1.0. Internet Requests for Comment (RFC) 1945, May 1996.

[4] F. Boussinot and R. de Simone. The ESTEREL Language. In *Proceedings of the IEEE*, volume 79, pages 1293–1304, Sept. 1991.

[5] W. Burkhardt. Universal Programming Languages and Processors: A Brief Survey and New Concepts. In *Proceedings of AFIPS Fall Joint Computing Conference*, volume 27, pages 1–21, 1965.

[6] C. Castelluccia, I. Chrisment, W. Dabbous, C. Diot, C. Huitema, E. Siegel, and R. D. Simone. Tailored Protocol Development Using Esterel. Technical Report 2374, INRIA, Oct. 1994.

[7] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating Efficient Protocol Code from an Abstract Specification. *IEEE/ACM Transactions on Networking*, 5(4):514–524, 1997.

[8] P. Clements, F. Bachmann, L. Bass, D. G. ahd J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Reading, Massachusetts, 2003.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol—HTTP/1.1. Internet Requests for Comment (RFC) 2068, Jan. 1997.

[10] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[11] C. Jones, A. Romanovsky, and I. Welch. A Structured Approach to Handling On-Line Interface Upgrades. In *Proceedings of the Workshop on Dependable On-Line Upgrading of Distributed Systems*, pages 1000–1005. IEEE Computer Society, Aug. 2002.

[12] D. Luckham. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Ordering of Events. In *Proceedings of the DIMACS Partial Order Methods Workshop IV*. Princeton University, July 1996.

[13] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, Apr. 1995.

[14] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Distributed Software Architectures. In *Proceedings of the Fifth European Software Engineering Conference*, number 989 in Lecture Notes in Computer Science, pages 137–153. Springer-Verlag, Sept. 1995.

[15] W. Means and M. Bodie. *The Book of SAX: The Simple API for XML*. No Starch Press, July 2002.

[16] J. Mogul, R. Fielding, J. Gettys, and H. Frystyk. Use and Interpretation of HTTP Version Numbers. Internet Requests for Comment (RFC) 2145, May 1997.

[17] N. Pryce and S. Crane. Communication and Configuration in Distributed Systems. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 144–151. IEEE Computer Society, May 1996.

[18] N. Ryan. *Realizing Dynamic Protocols via Event-Based Parsing*. University of Colorado, Boulder, Colorado, 2003. Available from the author.

[19] R. Taylor, N. Medvidovic, K. Anderson, J. J.E. Whitehead, J. Robbins, K. Nies, P. Oreizy, and D. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22(6):390–406, June 1996.

[20] P. van Eijk, C. Vissers, and M. Diaz. *The Formal Description Technique LOTOS*. Elsevier, 1989.

[21] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *Computer*, 25(3):38–49, Mar. 1992.