Using Event-Based Parsing 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

University of Colorado
Department of Computer Science
Technical Report CU-CS-947-03 March 2003

© 2003 Nathan D. Ryan and Alexander L. Wolf

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, delivery modes, and message sequences. Event-based parsing techniques are used in a novel way to present to the application the semantic concepts embodied in 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 syntactic details into the code of an application means that changes to the protocol might force alterations to the application in order 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 proposed to address the problem. For example, the database community has explored the problem of multi-database integration, and has offered the technology of so-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 pair-wise basis. In the software architecture community, the latest attempt at solving the problem comes in the form of the *connecter* [8], which is an architectural element whose purpose is to encapsulate inter-component communication. The idea is that it should be possible to replace one connector by another to effect a protocol change, possibly 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 parsing*. The idea behind event-based parsing is to generate "events" as syntactic structures are parsed. This technique is demonstrated to some degree by the XML SAX parser [16], 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.

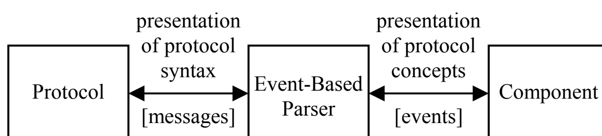

Fig. 1: Separating protocol syntax from protocol concepts.

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. We hypothesize that the event-based parsing technique can be generalized and formalized so that an event-based parser can be constructed on demand. If the protocol specification changes, a new event-based parser can be constructed to replace the old one, and the application can continue to receive the events in which it is interested. We intend to explore this hypothesis in future work.

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. 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 parser, 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. The example is of an HTTP 1.1 client attempting to communicate with an HTTP 1.0 server.

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.

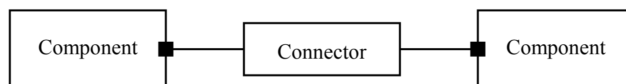

Fig. 2: Connector coupled to components.

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 [12] 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 [11], a powerful (largely symbolic) protocol description language that is accompanied by a wide variety of validation tools.

Although the techniques and languages provided by the Wright model are general enough to be used in almost any situation, the Wright model 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.

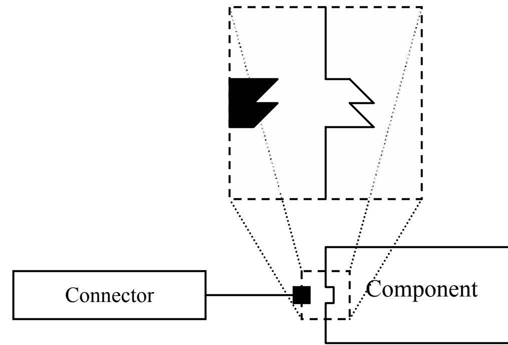

Fig. 3: Mismatch in connector/component coupling.

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 an application. 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 *a priori* predict 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 [20]. 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 [11], LOTOS [9], and Esterel [4, 6], as well as the architecture description languages Wright [1], Darwin [18], and Rapide [14, 15]. 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, it appears that Rapide's poset specification language is the most suitable.

We turn now to event-based parsing. For that we have looked most closely at the XML SAX parser [16] 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, 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 parser.

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. (For simplicity we generally use only the term "event-based parsing", as for example in Figure 1, but we intend this to also include "event-based composition".) XML may serve as a model here as well. Various toolsets, 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 might possibly be 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 a recognizer. 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 recognizers. 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 recognizers can be efficient, as well.

3. TECHNICAL APPROACH

The general approach we have developed to support dynamic protocol evolution consists of two main elements: a four-part specification of protocol syntax, and an event-based parser technology for translating from the specified syntax into semantic concepts. The four-part specification serves as a means to conveniently modularize the different aspects of a protocol that might evolve, while the event-based parser, whose behavior is driven by the protocol specification, presents semantic concepts as abstract events.

3.1 Protocol Specification

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 and 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 may require several specifications for each interaction of a protocol, one specification for each "role" a component may assume while using the protocol. Further, we need descriptions for the modes of message delivery, which include 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, but their influence on a protocol must be acknowledged.

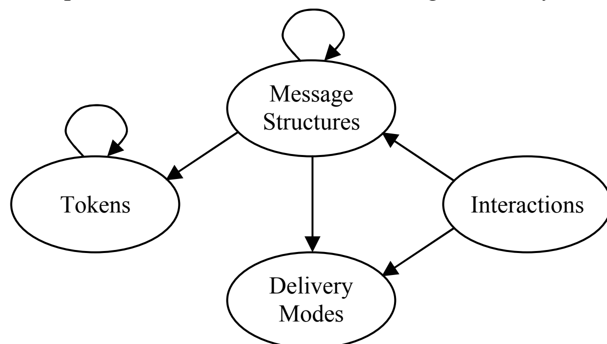

Fig. 4: The four parts of a protocol specification.

Thus, we have a four-part protocol specification. Figure 4 depicts the "uses" relationship among these four elements.

Given this framework for protocol specification, a description language must be chosen (or developed) to satisfy each of the four 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 four 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 all known token specifications is regular, while the expressive power required by all known message structure specifications is essentially context-free. Message structures do require some amount of context-sensitive specification to handle, 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 we use description languages akin to LEX, for tokens, and YACC, for message structures [13].

For the interaction specifications, a more powerful language is necessary, specifically state machines with guarded transitions. We have adopted the poset language of Rapide [14].

The choice of a description language for message delivery modes is problematic, since notions such as the recomposition of messages from possibly out-of-order communications and transport bindings do not immediately lend themselves to an obvious description language. Thus, we have not yet settled upon a suitable language for this part of the protocol specification.

3.2 Event-Based Parsing Approach

The primary elements of the event-based parsing approach are the four-part *protocol specification* described above, an event-based parsing *system* that generates and manages event-based parsing *units*, an event-based parsing unit that recognizes the protocol, and an application *component* that interfaces with the event-based parsing unit. 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.

The four-part protocol specification is used as input for the generator element of the event-based parsing system. The generator creates a unit that recognizes the specified protocol. The unit is responsible for deconstructing protocol messages via the parser, constructing protocol messages via the composer, and maintaining state for each protocol interaction. The unit also uses an event handler (not shown) to apprise the event-based parser system of notable discrepancies, as determined by both the parser and the composer.

Protocol messages are input to the parser of the unit from some source component. 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 the event handler of the component. Once delivered, it is up to the component to decide how to interpret them.

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.

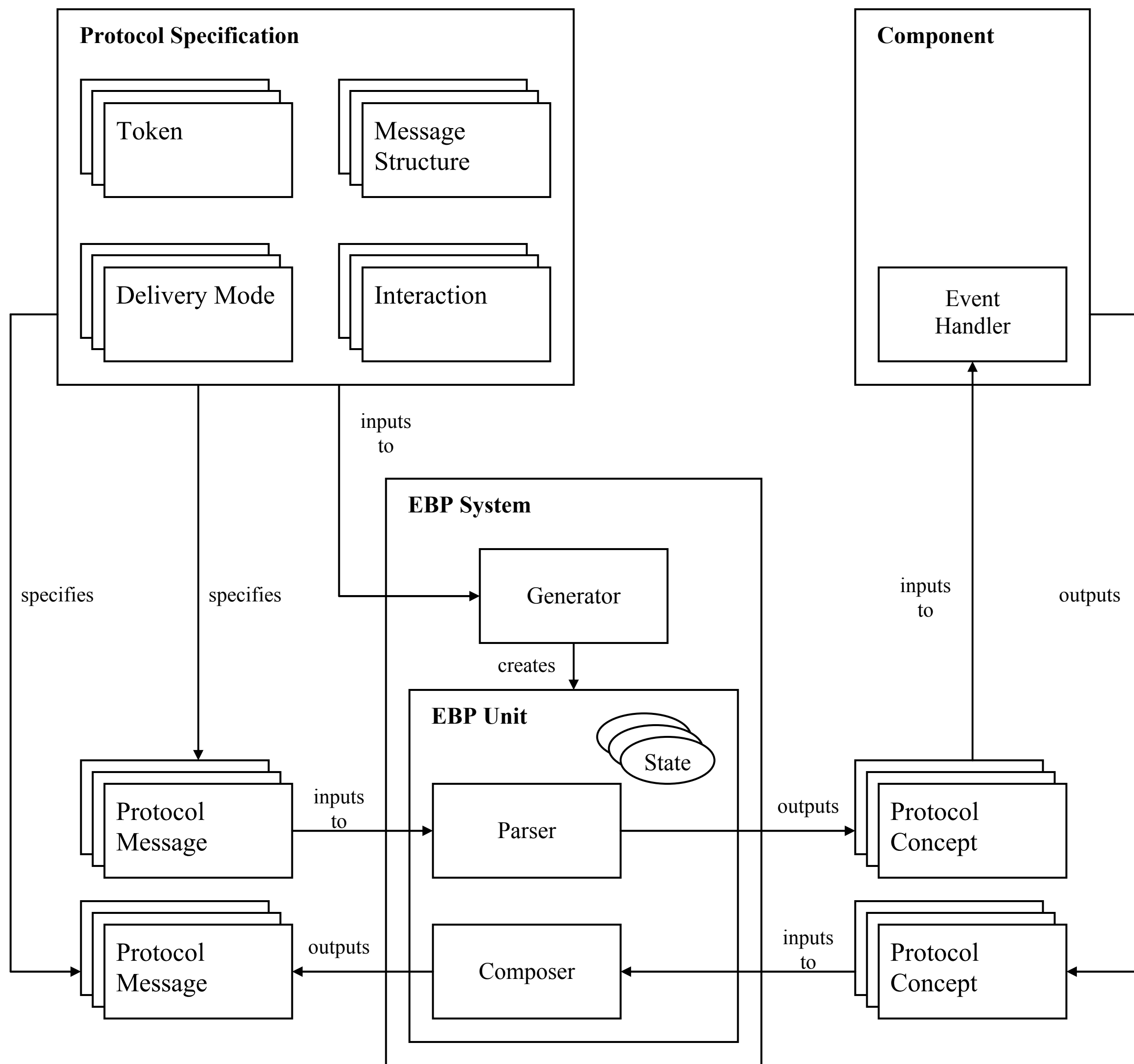

Fig. 5: Relationships among the major elements in the event-based parsing approach.

Notice that the application component must cooperate with the event-based parsing elements, 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. As we demonstrate in the next section, a cooperative component such as this may simply be a proxy for a "real" component, thereby hiding from that component the very existence of the event-based parsing mechanism.

4. Example

In this section we present an example application of our approach to dynamic protocol evolution. The scenario involves an encounter between an HTTP 1.1 client and an HTTP 1.0 server, the goal being to enable these two components to communicate using their respective versions of the protocol, but without modification to either component. For details of the HTTP 1.0 and HTTP 1.1 protocols used in for this example, see RFC 1945 [3] and RFC 2068 [10], 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

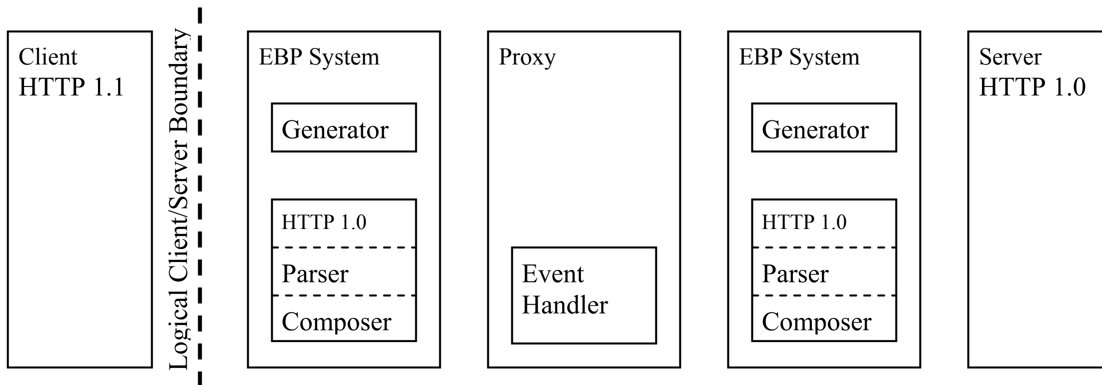

Fig. 6: Initial configuration of the HTTP application scenario.

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 [17]. In theory, however, many (if not most) HTTP 1.1 requests are manageable 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 configuration of the HTTP application constructed using our approach is shown in Figure 6. The client is an unaltered HTTP 1.1 client and the server an unaltered HTTP 1.0 server. The proxy is a simple application that receives HTTP events from the parsers, and forwards HTTP 1.0 concepts to the alternate composers. The event-based parsers and composers are derived from a complete specification of the HTTP 1.0 protocol. This specification, which is not shown here but is available elsewhere [19], uses the description languages discussed in Section 3.1.

The essence of the scenario is as follows: The client sends an HTTP 1.1 GET request. For the purposes of this example, we assume that the GET request is the simplest form of such a message (e.g., no optional headers are present). The message is parsed, filtered through a server proxy that can differentiate HTTP 1.0 concepts, is reconstituted as an HTTP 1.0 message, and then presented to the server. The server replies with a similarly minimal HTTP 1.0 status 200 response message, with the translation process performed in reverse for the benefit of the HTTP 1.1 client.

Interpreting the specification of HTTP 1.1 we find that the minimal GET request will generate the following series of events:

```
[01] token       MISC.METHOD_GET
[02] token       URI.PATH_REL
[03] token       URI.PATH_ABS
[04] token       URI.REQUEST
[05] token       MISC.VER_HTTP_ID
[06] token       MISC.VER_MAJOR
[07] token       MISC.VER_MINOR
[08] token       MISC.VER
[09] structure   GET.LINE
[10] token       HEAD.HOST_NAME
[11] token       URI.HOST
[12] token       URI.HOST_PORT
[13] token       HEAD.HOST_VALUE
[14] token       HEAD.HOST
[15] structure   GET.HEADER
[16] structure   GET.HEADER_LIST
[17] structure   GET.REQUEST
[18] interaction server_1
```

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 in a header. Event 9 is a message structure event that indicates the recognition of a line in a GET request.

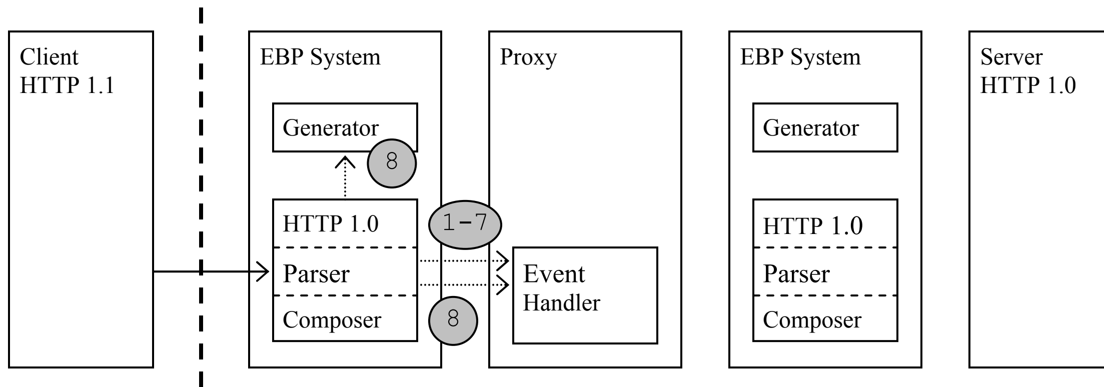

Fig. 7: Scenario events 1 through 8.

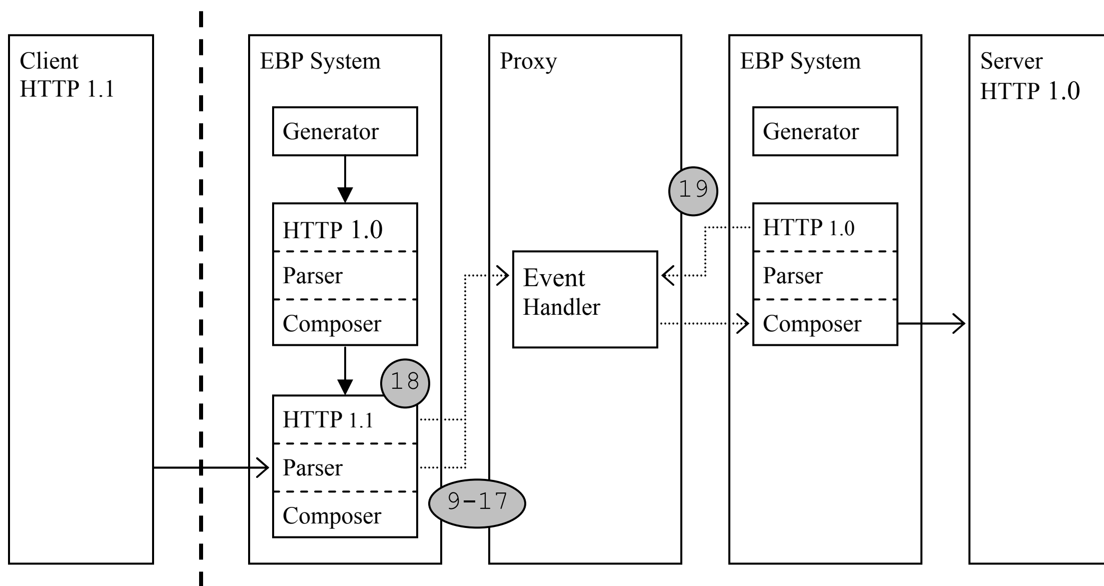

Fig. 8: Modified configuration and scenario events 9 though 19.

The previous eight events indicate the recognition of tokens constituting that line. Event 18 is an interaction event that indicates a state change in the state model representing the interaction; the name of that state as given in the protocol specification happens to be “server_1” and represents the fact that a server has received a request.

We now begin to describe the scenario, step by step. Figures 7 and 8 depict the processing of the first eight events and the remaining eleven events, respectively.

The first step is for the client to compose and send the request. Instead of being received directly by the server, the request is received (or intercepted by) the client-side EBP system. Our scenario will make the assumption that any incoming message can be routed directly to the HTTP 1.0 EBP unit, the parser of which then begins to parse the message. The first seven events (all token events) are generated and sent to the proxy’s event handler.

A crucial point is reached with the eighth event, which is sent to both the event handler of the proxy and the event handler of the HTTP 1.0 EBP unit. An inconsistency of version is found, and the EBP event handler temporarily suspends the EBP from further parsing, forwarding information to the generator and indicating the state of the current interaction and the elements that caused parsing to be suspended. Control is now transferred to the generator.

At this point the generator must somehow locate a specification for HTTP 1.1. There are many options for how this might be done, but a detailed treatment of this issue is beyond the scope of this paper. However, consider possibilities such as a resource discovery process or explicit addressing within the protocol.

Once the generator has a specification for HTTP 1.1, it creates an HTTP 1.1 EBP unit using standard compiler techniques, with an interaction state synchronized to that of the HTTP 1.0 EBP unit. The EBP unit generator also informs the HTTP 1.0 EBP of the creation of a new EBP unit that will assume control over parsing the request.

The consistency of the two parsers can be validated (relative to the text already parsed) once control over parsing is transferred from the HTTP 1.0 EBP unit to the HTTP 1.1 EBP unit. The HTTP 1.0 EBP unit must transfer both the remainder of the client request and the associated parsing state to the HTTP 1.1 EBP unit. (The interaction state was already transferred by the EBP unit generator, when the HTTP 1.1 EBP unit was created.) Once parsing control has been passed to the HTTP 1.1 EBP unit, parsing can resume, with both the client and the proxy being unaware of any changes in the way the protocol is being parsed. The parser of the HTTP 1.1 EBP generates the next eight events.

The proxy will likely not understand events 10 through 14, since they are concepts outside the proxy's context of HTTP. (Recall that the proxy is a filter for HTTP 1.0 concepts only.) Regardless, all eight events will be sent to the proxy's event handler, and the proxy may choose to ignore them or process them as desired; in the case of our scenario, we assume that the events are ignored, and thus the concepts associated with the "Host" header will be filtered.

The HTTP 1.1 EBP is now finished parsing the GET request, and will generate an event to be sent to the proxy indicating such. This event indicates to the proxy that it has been provided a collection of concepts that comprise a full HTTP message. The proxy may now forward these concepts to the composer of the HTTP 1.0 EBP unit of the right-most EBP system, so that the message can be reconstituted according to the specification of the HTTP 1.0 EBP unit, presumably in a form that will be understood by the server. This event also indicates to the EBP unit a change in the interaction state.

The HTTP 1.1 request has now completed the process of being sent from the client, deconstructed into HTTP concepts, reconstituted into a HTTP 1.0 request, and delivered to the server. The next step, then, is for the server to craft and send a response, which would follow an analogous process.

We note that the configuration of proxies, parsers, and other elements in this scenario 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 a perhaps 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 the proxy would be unconcerned with the order of the tokens, it would not be impacted by this modification, and the request would

go through normally. Today, such a simple change would likely confuse every current implementation of an HTTP server.

On a higher level, completed component interactions (as identified in the state machine maintained by an EBP unit) 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 pre-defined duration. If the limits of the specification were exceeded, the EBP unit 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 much easier to detect. Further, the specification of this larger protocol could be dynamically updated with new heuristics for such attacks.

As we point out elsewhere in this paper, much remains to be done to achieve our goal of fully supporting dynamic protocol evolution. For one, we must complete a formalization of event-based parsing to permit the construction of automated event-based parser generators. For another, we must settle upon a suitable description language for message delivery modes. Finally, we must explore the range of possible methods for employing our techniques in a variety of protocol evolution scenarios.

6. ACKNOWLEDGMENTS

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

This work was supported in part by the Defense Advanced Research Projects Agency under agreement number F30602-01-1-0503. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. 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.

7. REFERENCES

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

[2] J.W. 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*, UNESCO: 125-132, 1959.

[3] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945, Hypertext Transfer Protocol -- HTTP/1.0. MIT/LCS and University of California Irvine, May 1996. http://www.w3c.org/Protocols/rfc1945/rfc1945.txt

[4] F. Boussinot, and R. de Simone. The ESTEREL Language. In *Proceedings of the IEEE*, Vol. 79(9): 1293-1304, September 1991.

[5] W.H. Burkhardt. Universal Programming Languages and Processors: A Brief Survey and New Concepts. In *Proceedings of AFIPS 1965 Fall Joint Computing Conference*, Vol. 27(I): 1-21, 1965.

[6] C. Castelluccia, *et al*. Tailored Protocol Development Using Esterel. Rapport INRIA No. 2374, October 1994.

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

[8] P. Clements, *et al*. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2003.

[9] P.H. J. van Eijk, C.A. Vissers, and M. Diaz. *The Formal Description Technique LOTOS*. Elsevier Science Publishers, 1989.

[10] R. Fielding, *et al*. RFC 2068, Hypertext Transfer Protocol -- HTTP/1.1. University of California Irvine, January 1997. http://www.w3c.org/Protocols/rfc2068/rfc2068.txt.

[11] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[12] C. Jones, A. Romanovsky, and I. Welch. A Structured Approach to Handling On-Line Interface Upgrades. In *Proceeding of the Workshop on Dependable On-Line Upgrading of Distributed Systems*, 2002.

[13] J. Levine, T. Mason, and D. Brown. *LEX & YACC*, 2nd Edition. O'Reilly, October 1992.

[14] D.C. 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.

[15] D.C. Luckham, *et al*. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, Vol. 21(4): 336-355, April 1995.

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

[17] J.C. Mogul, R. Fielding, J. Gettys, and H. Frystyk. RFC 2145, Use and Interpretation of HTTP Version Numbers. DEC, University of California Irvine, and MIT/LCS, May 1997. http://www.w3.org/Protocols/rfc2145/rfc2145.txt.

[18] N. Pryce and S. Crane. A Uniform Approach to Communication and Configuration in Distributed Systems. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, May 1996.

[19] N.D. Ryan. Realizing Dynamic Protocols via Event-Based Parsing, University of Colorado, Boulder, Colorado, 2003.

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

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