Testing SIP Call Flows Using XML Protocol Templates

M. Ranganathan, Olivier Deruelle, and Doug Montgomery

Advanced Networking Technologies Division
National Institute of Standards and Technology
100 Bureau Drive, Gaithersburg, MD 20899, USA
{mranga,deruelle,dougm}@antd.nist.gov
http://w3.antd.nist.gov

Abstract. A Session Initiation Protocol (SIP) Call Flow is a causal sequence of messages that is exchanged between interacting SIP entities. We present a novel test system for SIP based on the notion of XML *Protocol Templates*, of SIP call flows. These templates can be pattern matched against incoming messages and augmented with general purpose code to implement specific protocol responses. This architecture allows test systems to be easily scripted, modified and composed. We describe these techniques in the construction of a SIP web-based interoperability tester (SIP-WIT) and comment on their potential more general use for scripting SIP services.

1 Introduction

The Session Initiation Protocol (SIP) [8] is a signaling protocol for setting up and terminating sessions for internet telephony, presence, conferencing and instant messaging. The SIP specification has been through a series of changes since the original RFC [9] was issued. Building comprehensive test tools and protocol stacks that both maintain backward compatibility and incorporate the latest specification is a challenging task. In this paper, we present a test system based on an XML-based pattern of a SIP Call Flow which accomplishes the goal of multi-level testing of SIP-enabled applications.

There are two types of components in a SIP-enabled network. Interior components act as signaling relay points. Examples of interior components include back-to-back user agents (B2BUA) and proxy servers. End components are signaling termination points and this is where the end-user application logic usually resides. Such applications include IP phone user agents (UA), chat clients, instant messaging and presence clients and other SIP-enabled user software. Such applications are usually built on a SIP protocol stack. Figure 1 shows a conceptual layering and heirarchical structure of a SIP stack and its relationship to a SIP application.

The lowest layer of the protocol is the *Message Layer* which reads messages off the network and parses them to present to the higher layers. Certain SIP applications, such as stateless Proxy Servers are built directly on top of

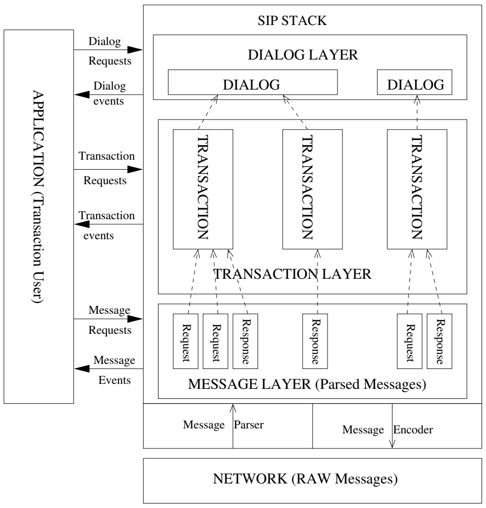

Fig. 1. A SIP application typically consists of the a Transaction User (TU) part where the application logic resides and *Transaction* and *Dialog Layers* which are implemented by a SIP Stack. The SIP Stack interacts with the application using a call/event interface.

the Message Layer. However, most SIP applications rely on the presence of a *Transaction Layer* which conceptually resides on top of the Message Layer. The SIP specification defines a *SIP Transaction* as a SIP Request, the provisional Responses generated by the Request followed by a final Response. The Message Layer presents the Transaction Layer with a stream of parsed messages. These messages can be SIP Requests or SIP Responses. The Transaction Layer is responsible for correlating outgoing Requests with incoming Responses and performing retransmissions of Requests as needed. A SIP Transaction is created as a result of an incoming SIP Request (*Server Transaction*) or as a result of an outgoing SIP Request (*Client Transaction*). A Client Transaction is completed when a final Response to a Request is received and a Server Transaction is completed when the final Response to the Server Transaction is sent out. A given SIP message is part of exactly one Transaction. SIP can run over both relaible and unreliable transports. The Transaction Layer is responsible for re-transmitting SIP Messages as needed.

Some applications such as stateful proxy servers and user agents rely on the establishment of *Dialog*s. A SIP Dialog is a peer to peer association between communicating SIP applications and is established by particular Dialog initiating Transactions. For example, a successful INVITE Transaction results in the creation of a SIP Dialog. Conceptually, the Dialog layer resides on top of the Transaction Layer and a given Transaction is part of exactly one Dialog.

Finally at the highest layer we have the notion of a *SIP Call*, which is identified by a globally unique *Call-ID*. A SIP call can consist of multiple Dialogs. All the Dialogs of the Call have the same Call-ID.

In a correctly functioning SIP application and stack, given a SIP message, the stack can identify to which Call, Transaction and Dialog it belongs to without needing to maintain connection state. Thus although applications and stacks may maintain persistent data structures associated with these abstractions, SIP is often called a stateless protocol.

SIP extensions are under development in a variety of different application domains (for example, instant messaging and networked appliance control), with different Request methods, headers and associated semantics. Still, the concepts of Message, Transaction, Dialog and Call are common to all the domains in which SIP is applied. In all these domains, a SIP application can be envisioned as a state machine that is transitioned on the arrival of messages, creation and completion of Transactions and creation and destruction of Dialogs.

2 Testing the SIP Protocol

Testing a SIP application can be decomposed in roughly the same fashion as the protocol itself. That is, an application may be tested at the Message Layer, Transaction Layer or Dialog Layer. We examine the issues of testing at the various layers in this section.

2.1 Testing at the Message Layer

The most obvious protocol errors are caused by incorrectly formatted SIP messages that do not conform to the specified grammar for URLs and protocol headers or by improperly functioning message parsers. These errors are easily discovered by using a parser that conforms to the specification. While constructing an ad-hoc parser for SIP headers is not difficult, there are pitfalls. The SIP grammar incorporates rules from from various RFCs that define specifications for mail, internet host names, URLs and HTTP. The resultant composite grammar is quite large, context sensitive and easily leads to parser implementation errors. For example, spaces are generally not relevant except in certain cases (for example the Request-Line, Status-Line and URI grammar definitions) where the RFC specifies strictly how many spaces are expected. Another common source of errors which is also an artifact of grammar composition arises from the fact that different sets of characters are legal in different portions of a message. The evolution of the protocol specification through the various revisions has also lead to some issues. For example, SIP URL addresses can appear in SIP messages in various headers. SIP URL addresses appearing in such headers are generally enclosed between pair of <> delimiters except the early RFC did not require this. There were also some early drafts that had context-sensitive disambiguating rules about whether to associate a parameter with a SIP URL or header.

A tool that tests for correct header parsing and formatting must itself be correct in parsing and formatting headers and conform to the SIP RFCs. A good way to achieve this is to use a parser generator. We elaborate on the techniques we have adopted in Section 5.

2.2 Testing at the Transaction Layer

Testing at the Transaction layer can be accomplished by generating messages that will establish and terminate Transactions. The latest SIP RFC (RFC 3261) specifies a robust way of Transaction identification but the earlier RFC (RFC 2543) had some ambiguities. Since interoperability with legacy equipment is often important, a test system must be able to generate both legacy and non-legacy scenarios.

Transaction timeout can be tested by delaying the SIP Response for a Transaction. Transaction matching can be tested by generating a spurious Response, or Responses, with fields left out or mis-specified at various points in the protocol operation. A test system should also be able to generate stray messages that do not correspond to a Transaction and are expected to be rejected by the receiving, stack.

Such tests are, for the most part straightforward but there are a few tricky cases. For example, most Transactions are just Request, Response sequences and do not make sense to abort while in progress. However, long running Transactions such as an INVITE Transaction may be aborted by sending the Server side of the Transaction a CANCEL message while the Transaction is in progress. However, the CANCEL only be processed at a certain point in the protocol operation and further operations that reference the canceled Transaction should result in a TRANSACTION NOT FOUND error. A test system must thus be able to generate such messages at specific points in the protocol operation to generate such erroneous conditions.

2.3 Testing at the Dialog Layer

Testing at the Dialog layer can be accomplished by setting up and terminating Dialogs. Crucial to the identification of the Dialog is the *tag* parameters of the SIP Message. The *From* header *tag* parameter identifies one end (the *Client* side) of the peer to peer association and the *To* header *tag* parameter identifies the other end of the peer association (the *Server* side). *Tags* are assigned in a pseudo-random fashion within the context of a Call. *Tags* were not mandatory in the earlier SIP specification. For legacy support, however, applications may support mechanism defined in the older RFC which specified another algorithm for Dialog identification. To test a stack at the Dialog layer, a test system should be able to generate Requests and Responses for established and spurious Dialogs both for legacy and current systems. Our experience with SIP implementations indicates that some common errors include incorrect assignment of *tag* parameters in the headers that identify Dialogs. Testing at the Dialog layer should also include testing for Dialog termination, which is accomplished by sending a BYE message, which can be issued by either side of the Dialog.

2.4 Call Flow Testing

Testing at the level of a SIP Call requires testing the causal sequence of SIP messages, transactions and Dialogs required to establish and release calls. Such sequences of exchanges are described as SIP *Call Flows.* Clearly call flow testing includes all the other layers outlined above, since a Call cannot be set up and terminated without correctly parsing and formatting messages or correctly establishing and terminating Transactions or Dialogs. Thus a test tool that can test at the level of a Call Flow needs to have facilities to test at the other levels as well.

Our approach to test at this level centers around defining XML tags and attributes to define the causal, event-driven behavior of a SIP end point participating in a Call Flow. We call this XML definition a *Protocol Template.* We then construct an event-driven state machine that interprets the Protcol Template to implement the Call Flow. A customizable User Agent which we call a *Responder* takes the Protocol Template as input and and generates the state machine and necessary synchronization actions for running the Call Flow. In the sections that follow, we further detail the design and use of these XML protocol templates as a basis for scripting a web-based SIP Call Flow tester.

3 SIP Testing with Protocol Templates

Our design goal was define a set of XML tags that can be used to represent a Call Flow as a finite state machine. The reason for choosing this approach was twofold. First, a popular way to debug SIP components is to participate in interoperability test events where the predominant mode of testing involves creation of simple signaling scenarios between components under development. We sought to duplicate this approach to testing in our automated test environment. Second we observe that there is currently no standardized way of expressing Call Flows. SIP-related Internet Drafts and RFCs specify Call Flows using sequence diagrams which are informal and subject to errors in interpretation. By choosing an XML representation for Call Flows and by widespread adoption of the conventions we propose, we hope to reduce interpretation errors in the future.

Figure 2 shows the overall conceptual view of our test system. It consists of a scripting layer (*Event Engine*) built on top of our NIST-SIP stack [12]. The Event Engine constructs one or more state machines after reading an XML file (*Protocol Template*) representing one or more call flows. The Protocol Template may be customized by adding code (*Service Script*) whose functions are invoked at specific points in the state machine operation. The Service Script can be inserted directly into the Protocol Template or specified externally as a JAVA [1] [2] class. We elaborate further on this scheme in section 4.

[1] The identification of specific software / hardware products or trademarked names in this paper is done soley for the purpose of adequately describing our work. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor imply that the products or names idetentified are necessarily the best avialable for the purpose.

[2] JAVA and JAIN are trademarks of SUN Micro Systems.

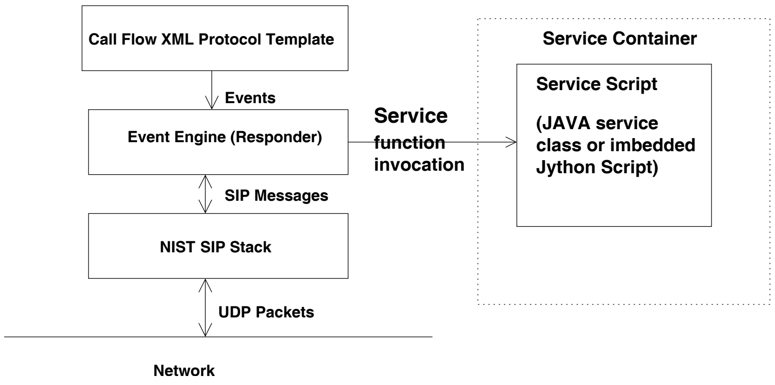

Fig. 2. Scripting Architecture: The Event Engine takes a Protocol Template as an input and constructs a state machine from it. The Protocol Template can invoke Service Script functions at specific transition points in its execution.

4 Protocol Template Programming Model

In this section we elaborate further on the XML representation for defining Protcol Templates. The hierarchy of XML tags that define a Protocol Template shown in Figure 3. Our programming model closely mirrors the layering of the SIP protocol. A **CALL_FLOW** corresponds to a SIP Call Flow and consists of a set of **DIALOG**s. A **DIALOG** is a specification for a finite state machine (FSM) which is represented by a set of XML tags and attributes. This state machine is instantiated when the corresponding SIP Dialog is created and defines the signaling behavior of a SIP-enabled end point. All messages within a Dialog have the same call identifier (Call-Id). Once the Dialog is established, the messages must also have the same *From* and *To* tags. A *Dialog* and its associated instantiation of the state machine defined by the **DIALOG** tag is created when a SIP Request with a previously unseen *Call-Id* or *tag* parameter arrives or is sent out by the system. Subsequently, all matching of messages occurs in the context of the created **DIALOG** so there is a one to one correspondence between an established SIP Dialog and the state machine defined by the matching **DIALOG** that is created as a consequence of the SIP Dialog being established. Because there is a single SIP Dialog for a given SIP message, an outgoing or incoming message can be uniquely associated with at most one instance of State Machine.

Each node of FSM is represented by an **TRANSITION** tag. Each **TRANSITION** tag consists of an optional nested **TRIGGER_MESSAGE** tag and an optional set of **GENERATE** tags. A **TRANSITION** tag can be nested inside of a **CLIENT_TRANSACTION** or **SERVER_TRANSACTION** tag (henceforth referred to generically as a **TRANSACTION**). The **TRANSITION** tags are nodes in the protocol state machine that can be triggered by message arrivals that match the nested **TRIGGER_MESSAGE** and are activated by a boolean combination of events specified in the *enablingEvent* at-

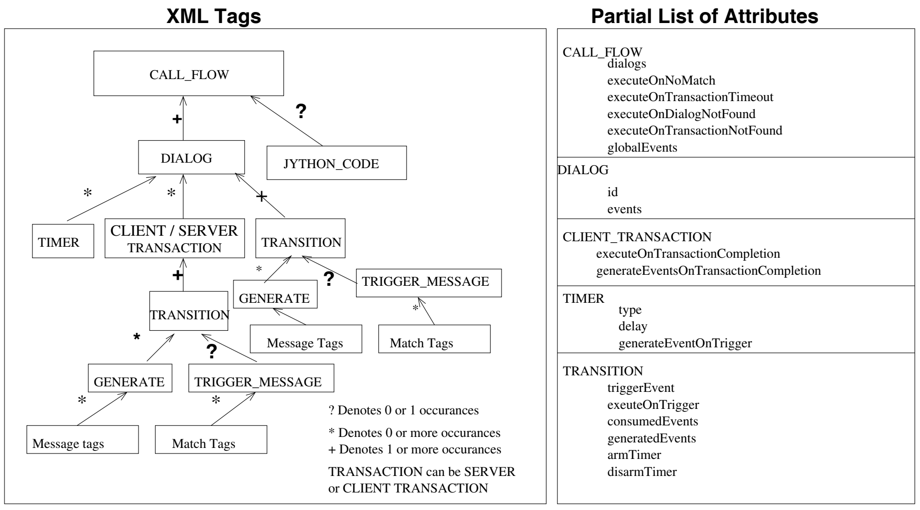

Fig. 3. Hierarchical Arrangement of Tags corresponds to the hierarchy of the SIP protocol. A partial listing of attributes for the XML tags is shown on the right (see text for explanation). The complete DTD is available from [12].

tribute. An *event* is a globally scoped or locally scoped counter (explained below) that is initialized to 0. If an **TRANSITION** node is nested in a **TRANSACTION** tag, then the messages that trigger it must also be part of the enclosing **TRANSACTION**. This provides a way of catching protocol errors related to unmatched Transactions. Note that this is not a static textual match specification because a transaction matching in SIP depends on dynamically generated header parameters. Once a Transaction is created it is associated with an instance of **TRANSACTION** node and **TRANSITION** tags within this node are used to generate state machine transitions in the associated **DIALOG** state machine. When an **TRANSITION** node is activated, it can generate events, activate timers, disable timers or call a Service Script function and optionally generate outgoing messages from the incoming message. The function to be invoked on **TRANSITION** node activation or Transaction completion is specified by the *executeOnTrigger* attribute and *executionOnTransactionCompletion* attributes respectively. The new message to be generated is represented by an optional **GENERATE** tag. The **GENERATE** tag can specify a list of editing rules to be used when generating an outgoing message from the incoming message (assuming that there is one). The **TRIGGER_MESSAGE** is a template that matches incoming messages and can *fire* the **TRANSITION** node if it is activated. If the node is not yet activated because the *enablingEvent* has not yet been satisfied, the fact that the trigger has been seen is noted. If the enabling condition occurs in the future, the node is activated at that time so the order of enabling conditon and trigger message arrival is not relevant. The **TRANSITION** node can be activated by any boolean event expression specified by its *enablingEvent* attribute. If no *enablingEvent* is specified, then the

TRANSITION node is assumed to be always *enabled* and may be triggered by an optionally specified **TRIGGER_MESSAGE**. Initial nodes are specified by the absence of a **TRIGGER_MESSAGE and** the absence of an *enablingEvent* attribute. Initial nodes may be used to start the interaction as soon as the state machine is initialized.

The Figure 3 also shows a partial list of additional attributes. The *generate*$_*$ attributes specifies a list of events to be generated. The *execute*$_*$ attributes specifies functions to be executed when specified events occur. The *consume*$_*$ attributes specify a list of events to be consumed when the specified event occurs. The *enablingEvent* attribute is a boolean expression on the *event* state variables. Events can be scoped either locally (visible only to a **DIALOG** state machine) or globally (visible across the entire **CALL_FLOW**). Global events can enable **TRANSITION** nodes in other **DIALOG**s. Local events are scoped within the **DIALOG** where they occur. The functions invoked from the attributes run the context of a separate class (either a JAVA class or an instance of a *Jython* [6] interpreter) which we call the *Service Class.* The same instance of this class is used for each service call within a Dialog.

Figure 4 shows an **TRANSITION** node *expectOK* from a UAC Call flow. This node is is enabled by the event *INVITEsent.* When the node is activated by this event, an incoming **SIP_RESPONSE** with *statusCode* of 200 can fire the **TRANSITION** node and cause the *OKReceived_ACKsent()* function to be invoked. The firing of the node arms the *byeTimer* timer and generates the *OKreceived_ACKsent* event and generates an outgoing *ACK* message. This message has its *From* header derived form an *agentId* "callee". An **AGENT** is just a way of specifying a list of attributes that are specific to the user that is being called. It allows us to customize a small portion of the code without altering the entire XML File. **AGENT**s may be bound to Registry entries (see section 5).

The service script can directly communicate with the event engine by generating events, starting and stopping timers etc, the same way as can be done with the attributes of the **TRANSITION** tags, allowing for finer grained control at the expense of clarity.

5 Implementation

In order to test for message formatting, our implementation [12] uses the ANTLR [16] parser generator to generate a parser for the SIP grammar. Conversion of the published EBNF to a format that is accepted by popular tools such as YACC and LEX is not straightforward. Advanced tools such as ANTLR make the task easier by allowing for closure on terminals as well as non-terminals, multiple lexical analyzers and the ability to switch between lexical analyzers during parsing and grammar inheritance; however, one must still resort to manual use of as *Syntactic* and *Semantic Predicates* to work through some ambiguities present in the grammar.

Figure 5 depicts the logic of processing incoming messages using our protocol templates. A *Call-Id* header identifies the call for the incoming message. This,

```
  <CLIENT_TRANSACTION
       onTransactionCompletion="onCompletion"
   />
   <TRANSITION
         nodeId            = "expectOK"
         enablingEvent       = "INVITEsent"
        executeOnTrigger    = "OKreceived_ACKsent"
        generatedEvent      = "OKreceived_ACKsent"
        armTimer= "byeTimer"
     >
      <TRIGGER_MESSAGE>
        <SIP_RESPONSE>
           <STATUS_LINE
                statusCode = "200"
             />
         </SIP_RESPONSE>
      </TRIGGER_MESSAGE>
      <GENERATE
          retransmit="false"
          removeContent="true"
        >
           <SIP_REQUEST>
                <REQUEST_LINE
                      method = "ACK"
                      agentId = "callee"
                />
            </SIP_REQUEST>
       </GENERATE>
  </TRANSITION>
</CLIENT_TRANSACTION>
```

```
<JYTHON_CODE>

  def  OKreceived_ACKsent():
       print "OK received and Snet an ACK"

  def onCompletion():
       print "Client transaction is complete"

</JYTHON_CODE>
```

```
  <AGENTS>
  <AGENT
     agentId="caller"
      requestURI="JitterVik@myhome.org"
  />
  <AGENT
     agentId = "callee"
     registryEntry="0"
  />
  </AGENTS>
```

Fig. 4. A Call Flow is represented as a set of TRANSACTIONs Events TIMERs and triggered TRANSITION nodes. TRANSITIONs are triggered by events and messages can generate outgoing messages and events to trigger other Expect Nodes. An AGENT is a short hand way of referring to a user identity. An AGENT entry can refer to a registry entry in the Proxy server. This is bound at run time to an actual value (see Section 5).

along with the *From* and *To tag* parameters of the incoming request are used to retrieve an instantiated **DIALOG** template for the call. If no template is found for the incoming call, then the one is created by looking for a **DIALOG** that can be instantiated. This is done by searching for a ready node. A node is *ready* if there are no outstanding events or messages for the node which prevent it from being enabled. A start node is one for which there is no *enablingEvent* tag and no **TRIGGER_MESSAGE** nested tag. Our stack and parser are written entirely in JAVA and we use introspection and inheritance to implement pattern matching facilities.

```
Let TRANSITION set be the set of TRANSITION nodes
Apply incoming messsage to all nodes in TRANSITION
Let READY denote a set of TRANSITION nodes that are enabled
Let  NON–READY denote a set of  TRANSITION nodes that are not yet enabled

While the READY set is not empty
    mark each TRANSITION node in the READY set unexamined
    for each unexamined TRANSITION node in the  READY set:
        mark it  examined
        if the incoming message matches TRIGGER_MESSAGE nested tag:
            Apply generated events to all  NON–READY TRANSITION nodes set
              and move the enabled TRANSITION nodes to the READY set
        If this message resulted in a transaction completion
            Apply completion events to all  NON–READY EXPECT nodes set
            and move the enabled TRANSITION nodes to the READY set
        Start any timers that are specifed by the armTimer attribute
        Stop any timers that are specifed by the  disarmTimer attribute
        Generate outgoing message list from nested GENERATE tag if it exists
        Invoke the executeOnTrigger method if specifed
        If this is a transaction completion:
          invoke the  executeOnTransactionCompletion method if specifed in
          the enclosing TRANSACTION tag
        Send out the outgoing messages
```

Fig. 5. Event Processing Loop implemented by the Responder. The responder reads the XML Protocol Template Specification and constructs a Finite State Machine for the test before running this algorithm. Similar processing occurs on Timer generated events.

Timers can be used to delay sending responses or to send multiple requests or responses. We keep a list of *Timer* records which is scanned periodically for ready timers. When a timer fires, it can generate events. Timer events are always global and timers may be enabled or disabled from **TRANSIITION** nodes.

The evaluation of the boolean expressions that enable the **TRANSITION** nodes is done using an embedded *Jython* control interpreter (different from the one that is used to evaluate the service scripts).

We have used the Protocol Template idea to prototoype and deploy a SIP web-based interoperability tester (SIP-WIT [11]). Figure 6 depicts the implementation architecture of SIP-WIT as comprising three main components: the test proxy, a Responder Event Engine and a Trace Visualizer. The proxy has a XML-based SIP Message pattern matching facility (not described here) that can be used to invoke external tools (including additional Responder instances) while the test is in progress. Both the test proxy and the Responder generate detailed message logs. The proxy uses the *Record-Route* header to ensure that it is in the signaling path for the entire Dialog. While the test is in progress or after the test is complete, the client can visualize the signaling exchanges using the trace viewer application described in section 5.1.

The entire test system is controlled by a *HTTP Servlet Engine* that acts as a front end. The test system user selects a test case and enters appropriate parameters into a HTML form, which results in an instantiation of a test proxy

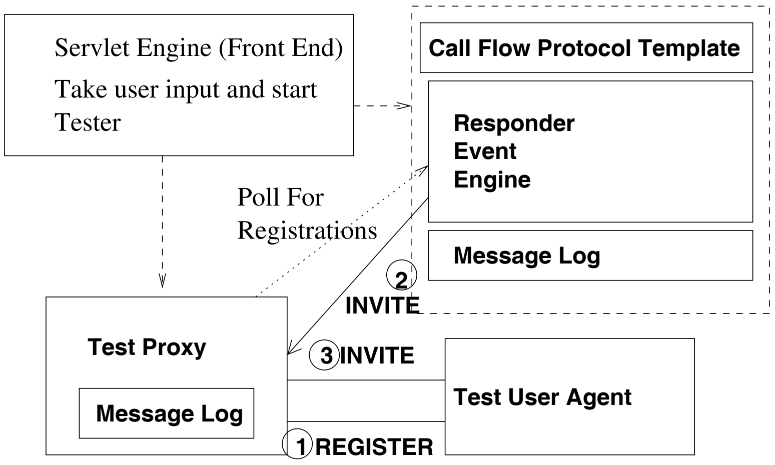

Fig. 6. The Test System: The Servlet Engine is used to start the test components. The Responder takes the Protocol Template as input and constructs a FSM for the test. It polls the test proxy for registrations to synchronize test startup.

and one or more Responders for the test. The trace viewer runs as an applet on the user's browser.

A *SIP Registrar* is a software component in a SIP-enabled network that allows users to register themselves and declare where they may be contacted. Our proxy server implements a registrar funciton and exports its registry entries for access via RPC. The protocol template may have **AGENT** tags which are bound at run time to Registry entries in the proxy. This is convenient when we do not know the identities of the participants of a test a-priori. If such bindings are specified, the *Responder* will poll the Registrar until the binding can be satisfied before it runs the test script. This allows for easy test synchronization and a customization.

5.1 Visualizing the Trace

The Proxy and Responder store their signaling trace for a pre-specified time period and makes it available for viewing by client applications. The traces are accessible via JAVA RMI and are grouped by call identifier. Each trace record has attributes that indicate where the message came from, where it is headed to, transaction identifier and other details that allow for the trace viewer to match Request and Response headers. The stack recognizes a special *NISTExtension* header which allows clients to record status information in SIP messages that are extracted and provided as part of the log file.

The trace viewer application retrieves a trace log from the proxy and can display a message sequence as a set of arcs that pass between stacks. Each stack is identified by IP address and port. The trace is sorted by time and Responses are matched with Requests and color coded appropriately. In order to reduce the

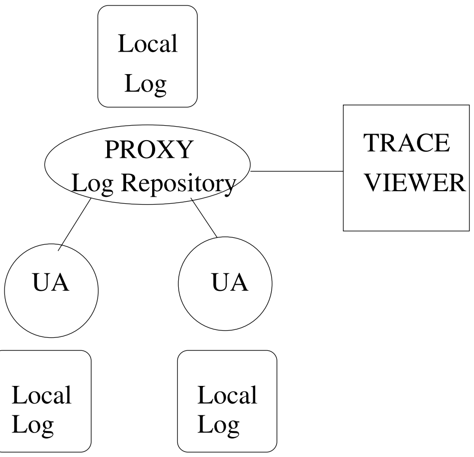

Fig. 7. Signaling Trace Collection : Traces may be collated at a single collection point. The request to fetch a trace is dispatched to slave repositories from the master repository and returned from the master repositry via JAVA RPC.

number of logging-related messages, and have a central collection point, traces are stored locally on each stack and collated on demand. This is done as follows: when a stack is initialized, a remote repository for the trace data may be specified. This is specified as an JAVA RMI URL. When the stack initializes, it registers with the central repository. When a request to retrieve the log comes into the central repository, the central repository dispatches the request to the slave repositories with a log collection request. This request can again be recursively broadcast by the slave repositories to its slaves. The slave repositories respond with the gathered log file. This hierarchical collection structure allows for scalability and decentralization. We have defined an XML syntax for the trace records to allow for possible standardization in future.

In order to avoid clock synchronization problems with merging the traces, we display the traces individually from the point of view of each collection point rather than as a single merged trace. Time stamps for each message are displayed along with the message relative to the beginning of the trace collection time.

In order to aid scalability and clarity, traces are organized by call-Id. Requests and responses are matched by identifying the transaction ID of the request and matching it to the corresponding response. The Figure 8 shows the trace visualization GUI. Each arrow corresponds to a message and the first line of the message is shown along with the arrow.

6 Field Experience

We took our tester implementation to the SIP Interoperability Test Event (SIPIT 11 [18]) where we were able to test against several proxy servers, user agents and IM clients. Our experience with the tester was positive in general but clearly we

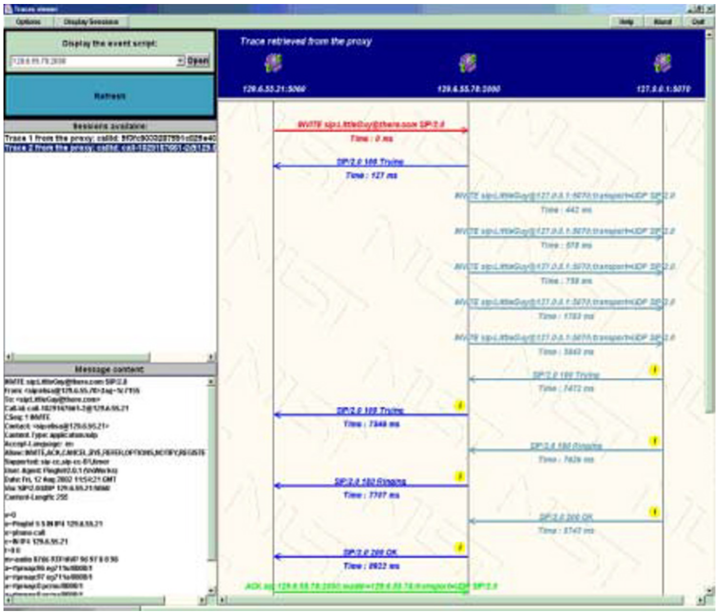

Fig. 8. The Trace Visualizer: This tool can accept traces gathered at the trace repository (which is part of the Proxy) or from an Ethereal trace capture. Trace records are formatted using XML. Arcs are color coded based on transaction and traces are separated by Call-Id.

need to add usability features (we often found ourselves editing configuration files). We were able to test third party call control, instant messaging and simple call flows using our responder and test proxy. We were also able to quickly script tests for extensions that are not part of the test proxy implementation. While further testing is needed, this increases our confidence that the programming model and XML representation are flexible and adequate to handle extensions that we have not yet considered.

An area of concern may be the performance and scalability of the system. When a message is received, the processing engine looks for an available **TRANSITION** node to fire. This currently involves a linear search through all the **TRANSITION** nodes that belong to a **DIALOG**. While this search can be pruned, in practice this turns out not to be a problem because a **DIALOG** usually only consists of a few **TRANSITION**s. The use of *Jython* to evaluate the trigger expressions and enabling conditions does lead to a performance bottleneck. However, *Jython* was only adopted for expedient prototyping purposes and may be replaced in future releases.

6.1 Related Work

As the popularity of the SIP protocol grows, many SIP testers have become commercially available. These are usually geared towards load testing. Load testing involves subjecting SIP components to high signaling loads such as hundreds or thousands of simultaneous calls. Load testing helps in uncovering synchronization bugs and tests scalability. Load testers differ significantly in function and have a different goal than our system.

The ITU-T has adopted TTCN-3 as a basis for building a test suite for SIP [15,13]. TTCN-3 is a procedural programming language with test-specific extensions which is applicable to a wide range of communication protocols. The test cases thus generated are procedural with an explicit encoding of the protocol to be tested through the TTCN language. On the other hand, our approach is explicitly tailored for SIP testing and our test cases are declarative rather than procedural. Like the TTCN testing approach, we base our tests on pattern matching but in addition we construct an XML description of the protocol call flow to run the test, thus leading to a simpler expression of the test case.

Finally, our approach bears a resemblance to Control XML [17] but functions at a lower level and is explicitly SIP-aware.

7 Conclusions and Future Work

In this paper we presented a protocol template based approach for testing the SIP protocol. Its main advantages are the clean separation between protocol actions and test actions and the specification of entire call flows using the protocol template, which leads to customizable scenario-based protocol testing. We demonstrated the viability of our approach by constructing a SIP web-based interoperability test system and have exercised our system at industry wide interoperability testing events.

Our future work will focus on adding the ability for users to customize their own test scripts by giving them the ability to insert their own service functions to be executed during the execution of the Call Flow. While our initial goal is to expand the capabilities of our test system, the addition of these capabilities will require that we address the two critical issues for any dynamic service creation environment: security and resource control. We plan to use bytecode re-writing techniques to address these issues.

One of the attractive features of SIP is the ability to customize call flows on a per-user basis. For example, users may wish to have the ability to customize call forwarding based on time of day or other considerations. Such customizations possible in a restricted domain using CPL. Here we are suggesting a more general technique which could enhance the programmability currently possible with CPL.

The ideas we have outlined in this paper can be applicable to wider domain than test scripting. Indeed, a SIP Stack is a software component that is aware of protocol state and generates events that can be fielded by a piece of application code. API layers such as JAIN-SIP [2] and JAIN-SIP-Lite [3] define an event model and expose the stack to the application at different layers of event abstraction. The application code is able to express an interest in events at different layers of abstraction via the *Listener* mechanism. However, these models constrain the application to one layer of another. What we have done here is to generalize this so that applications may express an interest in protocol events at any level of abstraction (i.e. at the message layer, transaction layer or dialog layer) in one unified framework. Thus, using an extension of the approach we have defined in this paper, we can go beyond test scripting and define standardized means for expressing dynamic behavior for protocol extensions that are yet to be proposed. More system support may be needed to extend this approach to do this. Exactly what support is needed will be determined by actually building such services. We are working on this idea collaboration with others in industry.

Our test system and the implementation of the mechanisms we have described in the paper are available from [12].

Acknowledgement

This work was sponsored in part by the NIST Advanced Technology Program (ATP) and by the Defense Advanced Research Program (DARPA). NIST-SIP includes the contributions of many people including Christophe Chazeau and Marc Bednarek who where guest researchers on this project at an early stage. An early version of the visualization tool was done as part of a student project at ESIAL conducted by Fabrice Burte, Hugues Moreau, Damien Rigoudy and Damien Rilliard.

References

1. Johnston, A., Donovan, S., Sparks, R., Cunningham, C., Willis, D., Rosenberg, J., Summers, K., Schulzrinne, H.: SIP Call Flows. Note http://www.iptel.org/info/players/ietf/callsignalling/draft-ietf-sipping-call-flowers-00.txt
2. Specification Lead Harris, C. (DynamicSoft Inc.): JAIN SIP 1.0 API. Note http://jcp.org/aboutJ ava/communityprocess/final/jsr032/
3. Specification Lead Rafferty, C. (Ubiquity Ltd.): JAIN SIP LITE API. Note http://jcp.org/jsr/detail/125.jsp
4. Specification Lead Kristensen, A. (DynamicSoft Inc.): SIP Servlet API. Note http://jcp.org/jsr/detail/116.jsp
5. Lennox, J., Schulzrinne, H.: CPL: A Language for User Control of Internet Telephony Services. Note http://www.ietf.org/internet-drafts/draft-ietf-iptel-cpl-06.txt
6. Hugunin, J., Warsaw, B., van Rossum, G.: Jython: A Python implementation in JAVA. Note http://www.jython.org
7. Lennox, J., Schulzrinne, H., Rosenberg, J.: Common Gateway Interface for SIP. Note http://www.faqs.org/rfcs/rfc3050.html
8. Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley M., Schooler, E.: SIP: Session Initiation Protocol RFC 3261. http://www.ietf.org/rfc/rfc3261.txt
9. Handley, M., Schulzrinne, H., Schooler, E., Rosenberg, J.: SIP: Session Initiation Protocol RFC 2543. Note http://www.ietf.org/rfc/rfc2543.txt

10. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol–HTTP/1.1 (RFC 2068). Note http://www.ietf.org/rfc/rfc2068.txt
11. NIST Advanced Networking Technologies Division: NIST-SIP Web-based Interoperability Tool (SIP-WIT) Note http://www.antd.nist.gov/sipwit
12. NIST Advanced Networking Technologies Divsion: NIST-SIP Parser and Stack. Note http://www.antd.nist.gov/proj/iptel
13. Wiles, A., Vassiliou-Gioles T., Moseley, S., Mueller, S.: Experiences of Using TTCN-3 for Testing SIP and OSP. Note http://www.etsi.org/tiphonweb/documents/Using_TTCN_3_for_Testing_SIP_and_OSPv8.pdf
14. Dahm, M.: Apache Byte Code Engineering Library (BCEL). Note http://www.apache.org
15. Schieferdecker, I., Pietsch, S., Vassilou-Gioles, T.: Systematic Testing of Internet Protocols - First Experiences in Using TTCN-3 For SIP. Note Africom 2001, Capetown, South Africa, http://www.testingtech.de/technology/Africom2001.PDF
16. Parr, T.: ANTLR parser gnerator. Note http://www.antlr.org
17. Auburn R.J., et al.: Call Control XML. Note http://www.w3.org/TR/ccxml/
18. SIP Interoperability Test Event.: Note http://www.pulver.com/sipit11/