

Application Composition in the SIP Servlet Environment

Eric Cheung and K. Hal Purdy, *Members, IEEE*

Abstract—The SIP servlet standard is a popular Java application programming interface (API) for developing and deploying Session Initiation Protocol (SIP) applications in Voice over IP (VoIP) environments. A number of commercial container implementations conforming to the current 1.0 specification are available; they provide feature rich and robust environments where a wealth of SIP applications can provide flexible control logic for advanced telecommunications applications. However, while one of the stated goals of the SIP servlet API is to support application composition, it is optional and not standardized in the 1.0 specification. In this paper, the authors propose an application composition framework that is general, powerful, and yet shields developers of individual applications from the complexities of application composition and feature interaction. Such a framework promotes modularity and application reuse. Also presented here are the results of implementing this framework on top of 1.0 implementations. This framework is being standardized for the new 1.1 SIP Servlet specification as part of the JCP JSR 289 process.

Index Terms—Communication system software, Feature Interaction, Programming Environment, Voice over IP

I. INTRODUCTION

The SIP servlet standard is defined in the Java Community Process (JCP)[1]. It defines a container-based model for building Session Initiation Protocol (SIP)[2] applications and specifies a Java API between SIP servlet programs and the container. In a SIP network, a SIP servlet container hosting multiple applications acts as an application server. For example, 3GPP defines SIP servlet containers as one of the standard application server types in the IP Multimedia Subsystem (IMS) architecture[3].

The SIP servlet API (SSAPI) builds on the well-established Servlet API used for programming HTTP servlets; thus it offers a programming model that is familiar to a large developer community. SSAPI provides a level of abstraction that is roughly between the SIP transaction layer and transaction users and as such affords the application programmers access and control over protocol details such as SIP message headers and contents. At the same time, details such as message formatting, retransmission, messages correlation, and protocol timeout are handled by the container. The container also provides additional facilities for proxy, user agent, and back-to-back user agent (B2BUA) applications, further simplifying the task of the programmers. The container manages application state for servlets, and distributed

containers also provide state replication, load balancing and automatic fail-over capabilities. Optionally, a container implementation may support both HTTP and SIP servlets, or it may be a Java Platform Enterprise Edition (Java EE)[4] container and thus may readily and seamlessly support converged web and telephone applications along with business logic and database integration.

At the time of writing, there are at least five commercial SIP servlet containers, making SSAPI arguably the best supported SIP application environment.

A. The Need for Application Composition

In general, application composition in telecommunications refers to communication among people in which two or more software applications are providing a feature or features on behalf of one or more of the parties to the communication. Representative examples of such applications include Call Waiting, Speed Dial, and Location Service. In an environment that supports application composition, it is desirable that each application involved in providing such features be able to do so independently of and without interfering with other applications.

Separate applications active on the same communication episode need to be able to combine their functionality in a well understood, controlled manner to enhance the features and experiences of the participants in the communication. In this way, general goals of feature modularity and code reuse are maximized. Composition of telecommunications applications is also widely regarded as an important design goal for managing the issue of feature interaction.

Even if application composition were not good design, the issue must still be considered. Clearly, the SIP Servlet environment needs to support the composition of SIP servlets written by unrelated parties and to have that composition produce good, predictable results. For these reasons, SSAPI-1.0 concurs with the need for application composition and uses similar reasoning for why it is desirable.

In SSAPI-1.0, application composition occurs when an initial SIP request is received. As each invoked application relays the request, another application may be invoked, thus forming an application chain. Responses and subsequent requests are then passed along this chain. Each application receives SIP messages as if it were the only application executing in the container. Using the example applications above, such an application chain could look like (Caller) — Speed Dial — Call Waiting — Location Service — (Callee).

Each application may comprise multiple servlets or Java

The authors are with AT&T Labs Research, 180 Park Avenue, Florham Park, NJ 07932 USA (cheung@research.att.com, khp@research.att.com).

classes. Composition may be achieved by mapping different messages to different servlets, dispatching requests from one servlet to another, or standard Object Oriented programming techniques. This kind of composition is confined to an application and is outside the scope in this paper.

II. DEFICIENCIES IN SSAPI-1.0

One of the stated goals of SSAPI-1.0 is to support application composition: "It is possible for several applications to execute on the same incoming or outgoing request or response. Each application has its own set of rules and execute independently of other applications in a well-defined and orderly fashion." However SSAPI-1.0 falls short of standardizing a mechanism for application composition. These shortcomings are examined in this section.

A. Application Composition is Optional and Unspecified

In SSAPI-1.0, container support for application composition is optional. Container implementations are in compliance by selecting just one or a subset of applications amongst all applications that match the request. Furthermore, if a container does opt to compose multiple applications, the behavior is not specified and is left entirely up to the container implementation.

B. Lack of Application Prioritization

The ordering of applications in a chain confers priority in handling SIP messages and is important for feature interaction management. However, if an SSAPI-1.0 container opts to invoke multiple applications, the order of invocation is not specified. In fact, the ordering may even be non-deterministic.

C. Lack of Deployer Control of Composition

In practice, it is the deployer of telecommunication services that needs fine grained, dynamic control over if, how, and when services are invoked on behalf of subscribers. It is the deployer that is most interested in rapidly deploying new services and offering flexible bundles of services to customers. Aside from proprietary mechanisms possibly provided by the container implementor, there are no SSAPI-1.0 mechanisms that enable the deployer to have any control, either real time or operational, over when and how SIP servlet applications are invoked to handle communication requests. There needs to be standard mechanisms whereby the deployer has a defined role in dynamically determining the invocation of SIP servlet applications. These mechanisms must allow the deployer to access external data sources (e.g. user profile information or subscriber configuration data) to decide the set and order of applications to be invoked.

D. Uncertainty About Applications' Intentions

Telecommunications features often act during call setup time in ways that influence the subsequent invocation of other applications. Using SIP, this happens when a proxy changes the Request-URI of an INVITE that it receives. B2BUAs also change the way in which requests are routed among applications by terminating one application path and starting a new application path with a new initial request. In both cases,

the problem with the existing specification is that SIP servlets cannot influence how they wish subsequent service invocation to take place. A B2BUA application can receive an initial request as a UAS. When it issues another request as a UAC, it has no way to indicate how it wishes the new INVITE to be considered. Should the INVITE be routed as if it were received *anew* by the container? Or, should this request continue to be routed to subsequent applications as if it were a *continuation* of the service invocation process that had already been started. Unlike the deployer, a SIP servlet application should only be able to signal its intentions so as to limit its influence on the application invocation process. The knowledge of which applications to invoke should rest with the deployer. However, applications do need to be able to indicate the context in which the application invocation decision is made.

E. No Formal Distinction of Caller vs Callee Services

In SSAPI-1.0, applications cannot easily determine whom they are invoked to serve. While some applications always serve the caller or callee, for example Speed Dial or Location Service respectively, many applications are invoked regardless of whether the subscriber is the caller or callee. Examples include Call Waiting and 3-Way Calling. By formalizing the categorization of caller and callee services, applications may find out in which category and for whom they are invoked, and the same application code may be used to handle calls in both cases.

III. DESIGNING A SOLUTION

A. New Standardization Effort

Since the SSAPI-1.0 Final Release in March 2003, container implementors and application developers have gained experience from real world development and deployment. Two areas of needed enhancement have become apparent: application composition, the subject of this paper, and support for converged SIP, HTTP and other types of applications. Beginning in Spring 2005, the authors and other industry experts submitted a new Java Specification Request (JSR) to the JCP to begin the process of updating the SIP servlet specification to address these needs along with other improvements and fixes. In January 2006, JSR 289 was officially launched with an expert group comprising members from 24 companies and research institutions. The charter of JSR 289 is to author a new 1.1 specification (SSAPI-1.1)[5].

The authors are members of the JSR 289 Expert Group and have been the main designers and authors of the solution that addresses application composition.

B. Design Goals

In addressing the deficiencies of SSAPI-1.0 in JSR 289 as regards application composition, the following design goals and trade-offs must be considered.

1) Clean Separation of Concerns and Responsibilities

The role of the application developer and the role of the deployer should be clearly separated. Application developers

are concerned with implementing the logic that constitutes an individual feature. On the other hand, deployers are concerned with combining individual features to provide complete services for their subscribers. By cleanly separating their respective concerns and responsibilities, each party may focus on its job while relying on a contract with the other parties as defined by the standard. This helps reduce cost and encourages portability and reuse.

2) *Generality and Flexibility*

The solution should be general and flexible to support any reasonable application selection scheme or algorithm. It should also allow for dynamic run-time changes to the selection algorithm. The algorithm should also have complete freedom over selection decisions and may thus use any data or criteria in making these decisions.

3) *Efficiency*

There is usually some performance cost to good software principles such as modularity and abstraction. For example, method calls incur a certain performance penalty over inline code. In the case of application composition in the SIP servlet environment, invoking multiple servlet applications adds computational complexity. The solution should aim to minimize such performance impact.

4) *Simplicity*

The details and complexity of application composition should be hidden from application developers. They should be able to implement an individual application as if it were the only application involved in a communication usage and need not nor should make assumptions about the presence and actions of other applications in the same usage.

5) *Clarity*

As discussed in sections II-D and II-E, the application's intention and for whom an application is acting should be clear to all parties.

6) *Compatibility and Standards Compliance*

Backward compatibility should be maintained such that SSAPI-1.0 applications can continue to operate in new containers. SIP specifications should not be violated, and the solution should complement other standardized telecommunications environments, notably IMS.

7) *Redundancy and High Availability*

In the SIP servlet environment, servlets are themselves stateless but may store application state that is managed by the container. The delegation of state management allows the container to provide replication and failover to achieve high availability. This solution should benefit from the same model: the maintenance of state necessary for application selection should be delegated to the container.

IV. PROPOSED SOLUTION

As discussed earlier, an initial request causes applications to be invoked and formed into a chain. The process of selecting applications in an ordered fashion is hereafter referred to as the *application selection process*.

The solution proposed here is adapted from the Distributed Feature Composition (DFC) architecture. The original definition of DFC[6] has been improved as a result of

extensive experience especially in the area of VoIP. DFC applies the pipe and filter model of software design to telecommunications features. This model is well-suited to the SIP servlet environment. Applications may be modeled as "filters" that receive and send messages that pass along SIP protocol "pipes". DFC defines a *router* as the key component that selects and invokes the next feature. In SSAPI-1.1, we introduce a new component, the *application router*, that interacts with the container to perform a similar function. The design and functioning of the application router in the SSAPI-1.1 environment is one of the most novel features of the future SIP servlet environment.

A. *The Application Router*

The application router is controlled by the deployer and plays a central role in the application selection process. An application router is essential but is logically separate from the container, and an API is defined between the container and the application router. The application router does not interact directly with applications. The context of the application router is illustrated in Fig. 1.

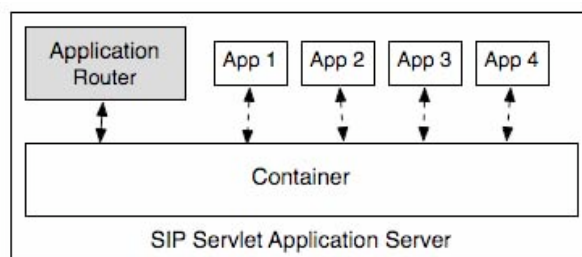


Figure 1. The Application Router

When the container receives an initial request, it queries the application router to obtain the name of the application to invoke. When this application relays the request, the container again queries the router. This process is repeated to build the application chain.

In deciding which deployed applications should be invoked, the application router is unconstrained. In particular, the application router may access whatever resources or databases it wishes. Because the initial request is likely to be relevant to the decision, the container provides it.

The router may need to keep selection state, e.g. it may need to 'remember' the list of applications already invoked for a particular episode. This API incorporates a state preservation mechanism. When the application router is first invoked, it may return a state information object. This state information object is opaque to the container, but the container associates it with the initial request. For subsequent invocation of the same episode, the container passes this object back to the router. In this way, the application router can be stateless.

Identifying when a request relates to the same episode is complicated because it depends on application logic. Often, only the application can reliably indicate whether two sessions it controls are part of the same episode. This topic is discussed in more depth in section IV-C below.

A key requirement for application composition is that the

deployer controls selection of the correct set of applications in the correct order to service a call. The deployer does so by controlling the logic of the application router. The division of responsibilities is now clear. Application developers write applications to provide call control logic to carry out the desired service while the deployer authors the application router logic to invoke the right set of applications for its subscribers. The container provides a framework in which the other components execute along with SIP protocol compliance, robustness, and management.

B. Subscriber Identity and Routing Regions

To better support the close association of applications with subscribers in telecommunications, this solution introduces the notion of a subscriber identity. Because the deployer knows the most about subscribers, the application router supplies the subscriber identity to the container when queried for the next application. The application router is free to identify the subscriber in any way it sees fit. The application uses a new API method to retrieve the subscriber information.

To capture the differing application subscriptions for callers and callees, the concept of routing region is introduced. Unlike other architectures, notably IMS, a single SIP servlet container instance must support the invocation of both originating applications for the caller and terminating applications for the callee. Hence, two logical routing regions are introduced: the *originating* and the *terminating* region. The routing region is also controlled by the deployer. When the container queries the application router for the next application to invoke, the application router also provides the routing region in which the application is to be invoked. The application may in turn obtain the identity of the routing region from the container.

A third *neutral* routing region is also introduced for applications that do not serve a particular subscriber, for example, a Call Logger that simply logs all calls. The deployer may also subdivide each of the originating, terminating and neutral regions into smaller regions.

For good feature interaction management, when the container receives an initial request, the application router begins the application selection process in the originating region. When all originating applications are selected and invoked, the application router progresses to the neutral region, and finally to the terminating region.

The flexibility in allowing applications in all three regions to be present in a single container does not preclude this solution from being used for a deployment architecture where different container instances handle separate routing regions. In fact, an important mechanism is provided to the application router in the API defined between it and the container. Namely, the application router can respond to a query by the container with a URI identifying an external route off the container. So at any point, the application router is free to direct the container to proxy the initial request to another external SIP entity instead of to a local application. Since that external SIP entity can well be another container instance, it is clear that routing regions may be separated onto different containers, even containers under separate administrative

control. Further, using the external route mechanism along with the standard SIP Route header based routing that is also provided to application routers, deployers may support hierarchical deployments where routing regions are subdivided even further and subsets of routing regions are handled on different containers.

C. Request Relaying and Routing Directives

As mentioned in section II-D, B2BUAs complicate application composition. This is because SIP itself defines no relationship whatsoever between the two (or more) SIP dialogs controlled by a B2BUA. The dialogs may be unrelated or they may be linked in that they represent two sides of the same communication episode passing through the B2BUA application. When the dialogs are linked in the simple sense that the application passes requests and responses back and forth between them, we say that the B2BUA is *relaying* requests and responses.

Other relationships between sessions emanating from B2BUA applications are also possible. An example is Call Waiting where three sessions are related to each other in a well defined but not simply expressed way. In this case, the B2BUA application is relaying between different dialogs at different times. While the relationships of dialogs controlled by B2BUA applications are many, one observation is clear: It is only the B2BUA application that understands the relationship of the dialogs that it controls.

From the perspective of composing multiple applications to service a single communication episode, the relationship between dialogs is of paramount importance. For the application router to know which application to be invoked next, it must first learn from the stored state information which applications have already been invoked for this communication episode. To make this determination, the application router must know when the B2BUA application is relaying between two dialogs or conversely, that the two dialogs are unrelated.

To provide the necessary information on the relationship of dialogs, the proposed SIP Servlet API introduces routing directives. Before a B2BUA issues a new request, it indicates to the container whether the request is a *NEW* request or a *CONTINUE* request. The routing directive indicated by the application to the container is then passed from the container to the application router when the B2BUA application actually sends the request, and the container queries the application router. If the request issued by the B2BUA is a *NEW* request, the application router interprets it as unrelated to any previous SIP dialog controlled by the application. For such a *NEW* request, the application router should start fresh a new application selection process. On the other hand, for a request that is marked *CONTINUE*, the new request is related to a request previously handled by the B2BUA application. In this case, the application router knows that this request is part of an overall communication episode which it has already dealt with. With a continued request, the router has access to the previously stored state information and it thus knows which applications have already been invoked to service this

communication episode.

V. IMPLEMENTATIONS

A. Adaptation for SSAPI-1.0 Containers

While the solution described in the previous section has been proposed in the JCP, it will be some time before SSAPI-1.1 is finalized and commercial implementations become available. In the meantime, an interim solution is needed on SSAPI-1.0 compliant containers that is compatible with the future 1.1 standard.

The authors have achieved such an interim solution through the implementation of an adaptation layer (AL). The AL sits on top of SSAPI-1.0 containers and interfaces with applications and a SSAPI-1.1-style application router. This is illustrated in Figure 2. The operation of the AL follows these steps: (1) When an application proxies or sends an initial request, it calls into the AL using an API similar to SSAPI-1.1. (2) The AL invokes the application router to obtain the name of the next application. (3) The AL stores the returned application selection state information in a proprietary header in the request. (4) The AL adds a special proprietary 'app' parameter to the Request-URI of the request, then routes the request back to the container. (5) The matching rules in the deployment descriptor files of the applications are written carefully such that only the named application matches the app parameter. (6) The container invokes the selected application to handle the request. This process is repeated until the application router selects no more application, and the AL routes the request externally.

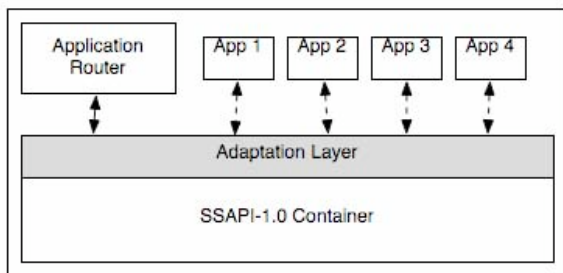


Figure 2. Adaptation Layer for SSAPI-1.0 Container

Our implementation of the AL has been tested on two separate commercial container implementations successfully. It has proven quite useful in testing the viability of the proposed solution presented here. It has especially afforded us the opportunity to try out application router implementations and start getting real experience on how such routers should be designed and built. Our first application router implementation is described in the next section. However, the use of the AL is clearly a temporary solution since it requires explicit cooperation from all applications deployed on a container. Each application must use the special API to call into the AL. It also requires careful crafting of the deployment descriptors. As well, the AL must be customized to work with different container implementations mainly (and ironically) because SSAPI-1.0 application composition is not well defined. For example, on the first container the authors tested

on, the AL is obliged to add a top Route header that points back to the container in order for subsequent applications to be considered. On the second container implementation, the AL must refrain from adding the Route header, because the container always tries to invoke another application anyway.

B. A DFC Based Application Router Implementation

Because DFC serves as the basis of much of this solution, it is logical to implement an application router based on an adaptation of the specific DFC routing algorithm. It is important to realize that while a DFC based application router is a good choice for the application router functionality, it is in fact a choice. A deployer is free to choose other implementations that make use of different logic for deciding on the next application to invoke. The authors have implemented a working DFC based application router. This implementation has been used and tested on two commercial SSAPI-1.0 containers in conjunction with the adaptation layer described above.

To implement a DFC application router, design choices were required as follows:

Source and Target addresses - In DFC, source and target addresses are fundamental properties of messages. For SIP servlets, requests are Java objects representing SIP protocol messages. So, the DFC application router implementation must decide what parts of a SIP message constitute the source and target addresses. For simplicity's sake, the initial decision is that the From header value serves as the source address. The Request-URI is the obvious choice to represent the target address. In more sophisticated implementations, one can imagine the deployer configuring what headers or other parts of the SIP message constitute the source and target addresses. For example, for an application router deployed in an IMS environment, one or more P-Asserted-Identity header field values would need to be included in the source address.

Address to Application matching - In the formulation of DFC, a feature subscription consists of a mapping between an address represented by a literal character string and a feature identifier. For this implementation, this mapping is generalized to be between a regular expression and a SIP servlet application identifier. Using regular expressions is much more convenient for administering feature subscriptions that apply to ranges of subscriber addresses.

Number and sequence of routing regions - DFC defines two routing regions: the source and the target. The SIP Servlet DFC application router adopts the same design except that, for consistency with IMS, the regions have been renamed the originating and terminating regions.

VI. ANALYSIS

The proposed solution is analyzed with respect to the design goals stated in section III-B.

The authors believe that the solution cleanly separates the concerns of application developers and deployers. The API is designed to prevent an application from targeting another application explicitly; thus applications may not directly control composition. On the other hand, the application router

may not modify or respond to requests. Therefore, the application router's sole function remains application selection not the implementation of any feature logic.

The goal of clean separation of concerns must be balanced against the clarity goal. By introducing the routing directive for applications to express intention explicitly, and the API for applications to discover region and subscriber, the authors believe that a suitable design tradeoff is thus achieved.

The solution addresses the efficiency goal by passing message objects between the container, application router, and applications. Within the container, there is no need to format, transmit and parse the SIP messages. A potential criticism of the solution is that the application router is invoked repeatedly after each application is invoked. Unfortunately, this is essential as the application may modify the requests thereby necessitating a re-evaluation of the selection. To address this issue, the application router may cache a pre-computed list of applications in its state, and only re-evaluate when necessary. This is in fact the strategy used by the application router in section V-B. As different application router designs may require different trade-offs, the solution does not dictate a particular strategy.

The solution places no restriction on how the application router performs the application selection - it may read static configuration files, consult external entities (e.g. the HSS in IMS), or rely on dynamic data such as time of day or network conditions. Containers are also required to support deployer supplied application router implementations. In this way, the generality and flexibility goals are fulfilled.

By delegating application selection state to the container, the container remains the central place where states are maintained. Therefore, the high availability goal is achieved.

SSAPI-1.0 constructs are retained and unmodified by the solution, thus existing applications will continue to work. The solution conforms to the relevant SIP standards. Externally, multiple applications executing within a container appear as a single application server, therefore compatibility with the overall SIP environment is not affected.

The goal of simplicity and hiding details of application composition from the application developers is only partially achieved. Application developers now need to have some understanding of composition, most notably in setting the routing directive, but this understanding is restricted to the application's perspective. However, developers may initially find this burdensome or confusing. It is hoped that as the SIP servlet programming model gains a larger base of developers, documentation and code examples aimed specifically at developers will become more widely available thus mitigating this issue.

VII. FUTURE WORK AND CONCLUSION

In the future, we intend to explore several interesting areas.

How does the SIP servlet application composition design relate to other telecommunications software architectures? In particular, how do the ideas presented here relate to JAIN SLEE and Parlay and their concepts of application composition?

How does the SIP servlet application router fit into IMS? Now that the application router is defined in the SIP servlet environment, it seems that some or all of the capabilities of the IMS Service Capability Interaction Manager (SCIM) component can be achieved with suitable application router implementations. Can a cooperating group of application routers distributed across a set of SIP application servers achieve the goals of the SCIM to manage feature interaction? Are there hierarchical application router topologies that could do even better?

How can application and application router design be improved? Our implementation of a DFC compliant application router has barely scratched the surface of application router design. What are the best current practices for application router design and implementation? For applications, with the added machinery that now allows multiple independent applications to coexist together, what guidelines can we propose to ensure application reuse and applicability in a wide range of service deployments?

To conclude, because the SIP servlet programming model incorporates the concept of a chain of applications active for a communication episode, the model is well suited for enhancements that codify good application composition principles. A solution embodying these principles is presented here and has been proposed for inclusion in the next SIP servlet specification, version 1.1. The proposed solution addresses the known deficiencies in the original SIP servlet specification, meets the stated design goals well, and achieves a good balance of control and responsibility among the three cooperating software entities: the container, the application and the application router. Although adapted in a non-general way for existing version 1.0 compliant containers, our implementation of the proposed solution has demonstrated that the ideas work.

ACKNOWLEDGMENT

The authors acknowledge the specification lead, Nasir Khan, along with members of the JSR 289 expert group for their aid in honing these ideas. Our colleagues, Greg Bond, Tom Smith, and Pamela Zave have also provided invaluable comments and suggestions.

REFERENCES

- [1] Anders Kristensen. SIP Servlet API Version 1.0. 2003. <http://jcp.org/en/jsr/detail?id=116>
- [2] J. Rosenberg et al, *SIP: Session Initiation Protocol*, IETF RFC3261, 2002.
- [3] 3GPP IP Multimedia Subsystem (IMS). <http://www.3gpp.org/>.
- [4] Java Platform, Enterprise Edition (Java EE). <http://java.sun.com/javase/>
- [5] JSR 289. <http://jcp.org/en/jsr/detail?id=289>
- [6] M. Jackson, P. Zave, *Distributed feature composition: A virtual architecture for telecommunication services*, IEEE Trans. on Software Engineering XXIV(10):831-847, Oct 1998.