A Framework for Converged Telecom Services and Mashups

Gregory W. Bond
AT&T Labs Research
Florham Park, NJ, USA
bond@research.att.com

Eric Cheung
AT&T Labs Research
Florham Park, NJ, USA
cheung@research.att.com

ABSTRACT

We describe a light-weight, Java-based framework for SIP servlet 1.1 containers that enables the development of modular, reusable telecom features suitable for integration into converged services. We show how the framework supports interaction between a feature and its external environment and how it facilitates the discovery task in different scenarios. We also discuss how the framework enables feature reusability and provide representative examples of reusable features that use the framework. Finally we discuss how this framework can be utilized to support telecom service "mashups": customized web services that incorporate telecom services with other services. We also discuss issues that arise when incorporating telecom services into mashups.

Keywords

Telecommunications, VoIP, SIP servlets, web services, mashups

1. INTRODUCTION

A SIP servlet is a Java class that encapsulates telecom feature logic. For example, the familiar Call Waiting feature could be implemented as a SIP servlet. A SIP servlet carries out its functions within the environment provided by a SIP servlet container, an overarching software framework that is responsible for, among other things, conveying SIP messages between a SIP servlet and other SIP entities including user agents.

Phone users are accustomed to accessing a variety of features during a call's lifetime, for example, Call Forwarding, Voice Mail or Call Waiting. Any such collection of features constitutes a telecom service. This, then, raises the question of how to compose a collection of SIP servlet-based features to provide a telecom service. The current SIP servlet 1.0 specification [3] provides no answer to this question. As a result, version 1.0 container vendors provide non-standard support for it. Therefore, to guarantee cross-container support all features must be contained within a single monolithic SIP servlet application, and the mechanism for composing these features is left up to the developer.

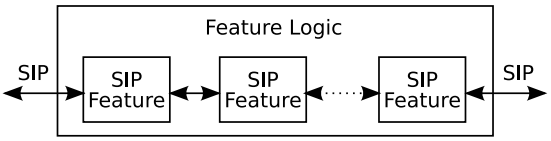

Figure 1: Composed features

Happily, the SIP Servlet 1.1 specification proposed by the JSR 289 working group addresses the feature composition question [6, 11]. This specification includes an *application router* component in a SIP servlet container. The application router determines which feature will receive an initial SIP request, arriving from an external user agent or generated internally by another feature; successive invocations of the application router serve to compose features as shown in Figure 1. From a software engineering perspective the application router offers new opportunities for telecom service development, namely, the ability to develop a telecom service as a collection of composed, modular SIP servlet-based features.

However, a collection of telecom features, modular or monolithic, rarely stands alone. As illustrated in Figure 2, telecom features are normally an integral part of a larger service that may, for example, access a database or provide a web-based user interface. Services consisting of SIP and non-SIP components are called *converged* services. The Sip Servlet 1.0 specification does not address converged services, but the 1.1 specification does. Given that the SIP servlet 1.1 specification supports modular feature development, the next question that arises is how a modular telecom feature should be integrated with its surrounding environment. The proposed SIP servlet 1.1 specification indirectly addresses this question by proposing low-level mechanisms that allow a SIP servlet to be accessed from outside the servlet; this permits, for example, an external entity to influence feature behavior by non-SIP means. Left unaddressed by the specification are higher-level issues of how a feature and its environment discover one another or how interactions between a feature and its environment are structured.

Modularization and convergence play a role in telecom service "mashups" [8]: customized web services that compose ("orchestrate") telecom web services with other web services,

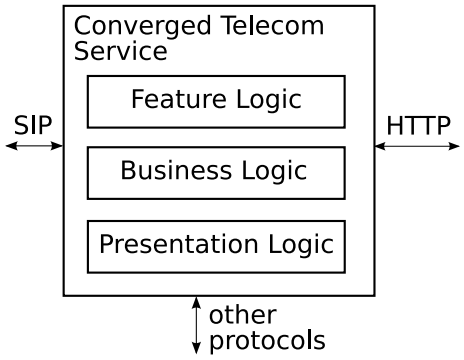

Figure 2: Converged telecom service components

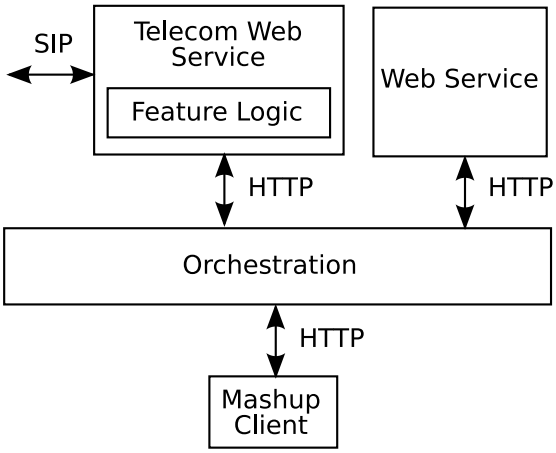

Figure 3: A telecom service mashup

as illustrated in Figure 3. A telecom web service encapsulates telecom feature logic and interacts with its environment using SIP and HTTP. For this reason a telecom web service is a specialized instance of a converged service that faces the same issues that general converged services face.

Another issue related to modularization is telecom feature reuse. Given that telecom features can be modularized it is natural to ask if they can be designed in such a way so as to be reusable in different service contexts. The goal here is to support reuse of telecom features in the same way that existing frameworks such as Enterprise JavaBeans (EJB) [5] support reusable business logic.

In the remainder of this paper we describe a Java-based light-weight framework for SIP servlet 1.1 containers that enables the development of modular, reusable telecom features suitable for integration into converged services. We show how the framework supports interaction between a feature and its external environment and how it facilitates the discovery task in different scenarios. No constraints are imposed on the environment other than that it be accessible via Java. We also discuss how the framework enables feature reusability and provide representative examples of reusable features that use the framework. Finally we discuss how this framework can be utilized to support telecom service mashups and issues that arise.

An implementation of this framework along with a number of reusable SIP features utilizing the framework are available as part of the open source ECharts for SIP Servlets development kit [18]. This implementation is being used to support two production IP telecommunication systems currently under development.

2. INTERFACING

To support development of a SIP feature and the non-SIP external components that it interacts with, an interface must be defined between them. The interface should expose only what is strictly necessary to the accessing component in order to minimize component inter-dependencies. To ensure that the interface is light-weight and that it accommodate the widest range of possible non-SIP environments, we have chosen to demarcate the SIP/non-SIP boundary close to the SIP feature logic. This means that the only constraint on the environment is that it be accessible via Java. Furthermore, if a heavier-weight interface technology is desired, for example SOAP [7], then it can be added on top of the lower level Java interface.

There are two common types of interaction between a feature and its non-SIP environment:

1. Interactions initiated by the feature such as reading provisioned data from a database or notifying the environment of service status. We call this type of interactions *SIP-to-Java* interactions.
2. Interactions initiated by non-SIP components such as initiating the creation of the feature in the first place, or exerting control over its behavior. We call this interaction type *Java-To-SIP* interactions.

Both interaction types are embodied as method calls. For example, a method would be called by the feature to read provisioned data from a database or a method would be called by the environment to exert control over a feature's behavior. Thus, for a given feature the two interaction types can be defined in terms of two interfaces: (1) method signatures comprising its SIP-To-Java interface and (2) method signatures comprising its Java-To-SIP interface. These two interfaces are illustrated in Figure 4. Both interfaces are defined by the feature itself since the decision of how a feature interacts with its environment affects the feature's internal design.

The implementation of a feature's Java-To-SIP interface methods is clearly the responsibility of the feature itself. This is because these methods will most likely access or manipulate the feature's internal state. On the other hand, the implementation of a feature's SIP-To-Java interface is clearly the responsibility of the external non-SIP components that interact with the feature. For example, when a feature calls a SIP-To-Java method to indicate its own status, the method will most likely update the state of some external component. Because the SIP-To-Java interface implementation is not the responsibility of the feature, the identity of the external component and the nature of the interaction with the external component needn't be known by the feature although these will be known by the SIP-To-Java interface implementation.

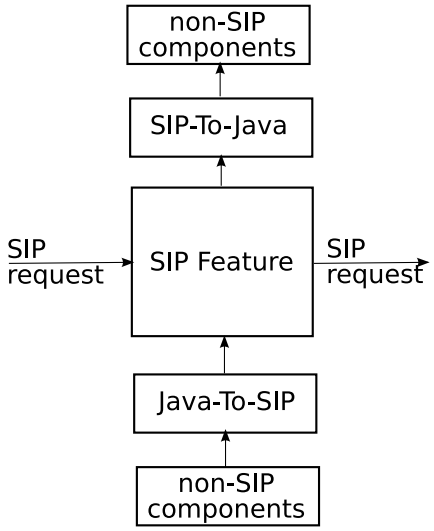

Figure 4: SIP service interfaces

3. DISCOVERY

Specifying interfaces between a SIP feature and non-SIP components constitutes only part of the solution. In general there can be many feature instances, each associated with their own calls. There may also be different external components associated with different calls. For this reason, an external component needs to be able to find the Java-To-SIP interface associated with a particular feature instance, and a feature instance needs to be able to find the appropriate non-SIP components via its SIP-To-Java interface. The nature of these discovery tasks differ depending on whether a feature instance is created in response to (1) the receipt of a SIP request, or (2) a request by a non-SIP component.

In the first case, graphically depicted in Figure 5, the feature instance is created by the receipt of a SIP request from a SIP user agent, independent of non-SIP components. Most traditional call services are created this way, for example Call Forwarding, Call Waiting, etc. The feature instance first gets an instance of its SIP-To-Java interface from the framework. Since the implementation of this interface is the responsibility of external components, the interface implementation instance has embedded within it the information required to contact the appropriate external components. The feature instance then uses its SIP-To-Java interface to publish identifying information required by an external component to access its Java-To-SIP interface. This way, an external component can use this information to obtain the feature instance's Java-To-SIP instance from the framework.

Normally, the framework is responsible for creating, storing and retrieving interface instances. Except in one special case, described below, these actions are hidden from features and external components. Instead, these components request to `get` an interface instance using a unique feature instance identifier, denoted by `sasId` in Figure 5, as an argument[1] The framework uses the `sasId` as a key to store and retrieve an interface instance associated with a SIP feature instance. When a component requests to get an interface instance, the framework returns the currently stored copy. If none exists then the framework first creates a new interface instance and stores it for subsequent retrieval.

In the second case, graphically depicted in Figure 6, a feature instance is created by a non-SIP component via a feature factory call, as would be the case with a Click-To-Dial service, for example. In this case, the feature instance needn't publish its `sasId` for discovery by a non-SIP component since an `sasId` is generated by the non-SIP component in the process of creating the feature instance[2]. With this identifier, the non-SIP component is able to obtain a Java-To-SIP interface instance from the framework.

As shown in Figure 7, since the non-SIP component initiates the creation of the feature instance, we also permit the non-SIP component to provide the framework with a SIP-To-Java interface instance where the instance is created by the external environment itself. The ability of the environment to `set` the value of an interface instance is the exception to the general rule of hiding all but the `get` methods mentioned above. This exception is a convenience that allows a non-SIP component to directly initialize a SIP-To-Java interface instance. The dual scenario in the SIP-initiated case, where a SIP feature sets the value of the non-SIP environment's Java-to-SIP interface instance is not applicable because the non-SIP environment is intended to be outside the purview of the feature in order to support feature reuse.

4. EXAMPLES

In the following section we provide examples of how the framework is utilized in different contexts: for a converged non-SIP initiated feature, for a converged SIP initiated feature, and for a converged multi-feature telecom service.

4.1 ClickToDial Feature

Figure 8 graphically depicts a ClickToDial feature, abstractly specified as a state machine. The ClickToDial feature is a converged feature that is initiated by non-SIP means [17]. Once initiated, the feature calls a specified first party address and then, after the first party is connected, calls a specified second party address. When the second party is connected, the two parties are connected to one another. Since the ClickToDial feature is initiated by non-SIP means then either of the non-SIP initiated discovery scenarios shown in Figures 6 and 7 can apply. In the following discussion we assume the latter scenario.

The ClickToDial feature's Java-To-SIP interface exposes a method for dropping its calls, and its SIP-To-Java interface includes methods to obtain the data values required for initiating the calls e.g. the first and second party addresses, and

[1]This identifier is actually the SipApplicationSession ID assigned to a SipServletApplicationSession instance by a container. We assume here that exactly one SipApplicationSession instance is associated with a feature instance.

[2]In less abstract terms, an external component would first create a SipApplicationSession instance and obtain its SipApplicationSession ID (not shown) prior to indirectly creating an associated SipSession instance, represented by the `createSipService()` method call. The SipSession instance is created by the container in response to an external component sending an initial SIP request.

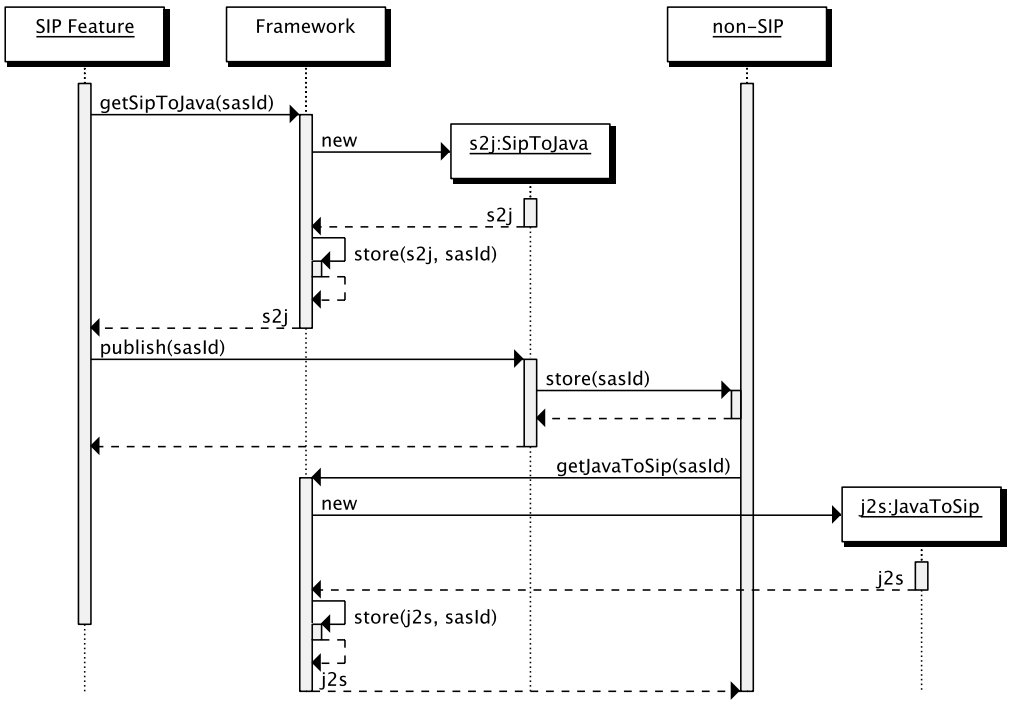

Figure 5: SIP-initiated interface discovery

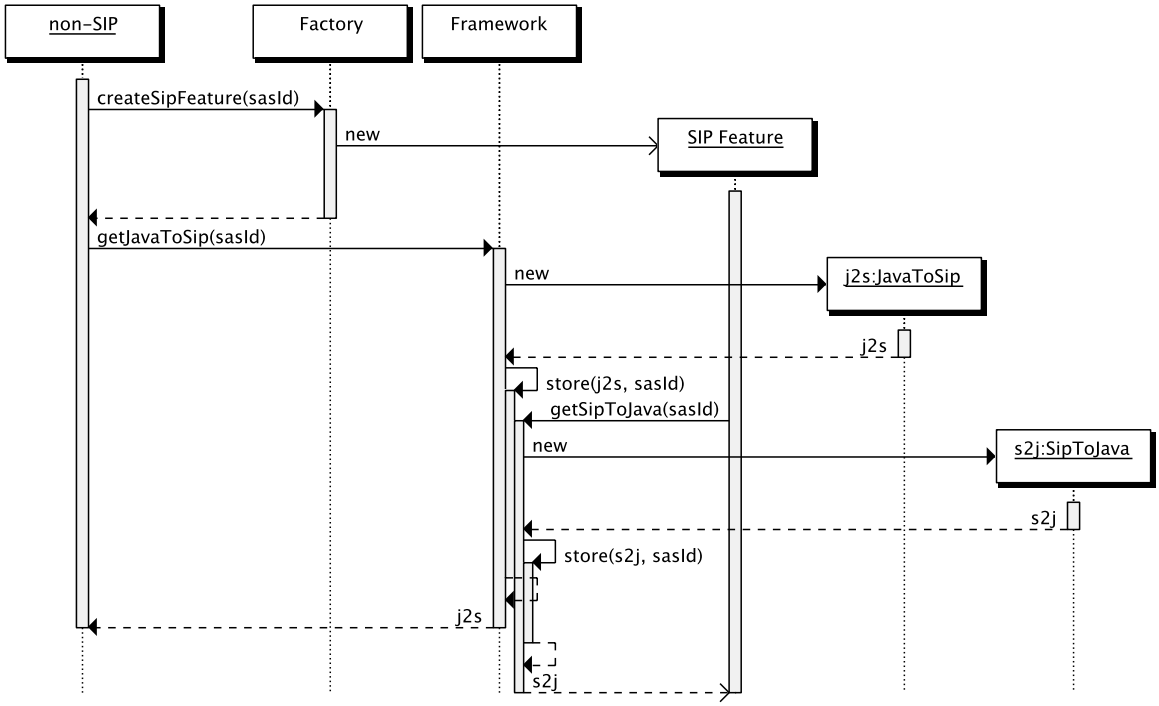

Figure 6: Non-SIP-initiated interface discovery - Scenario 1

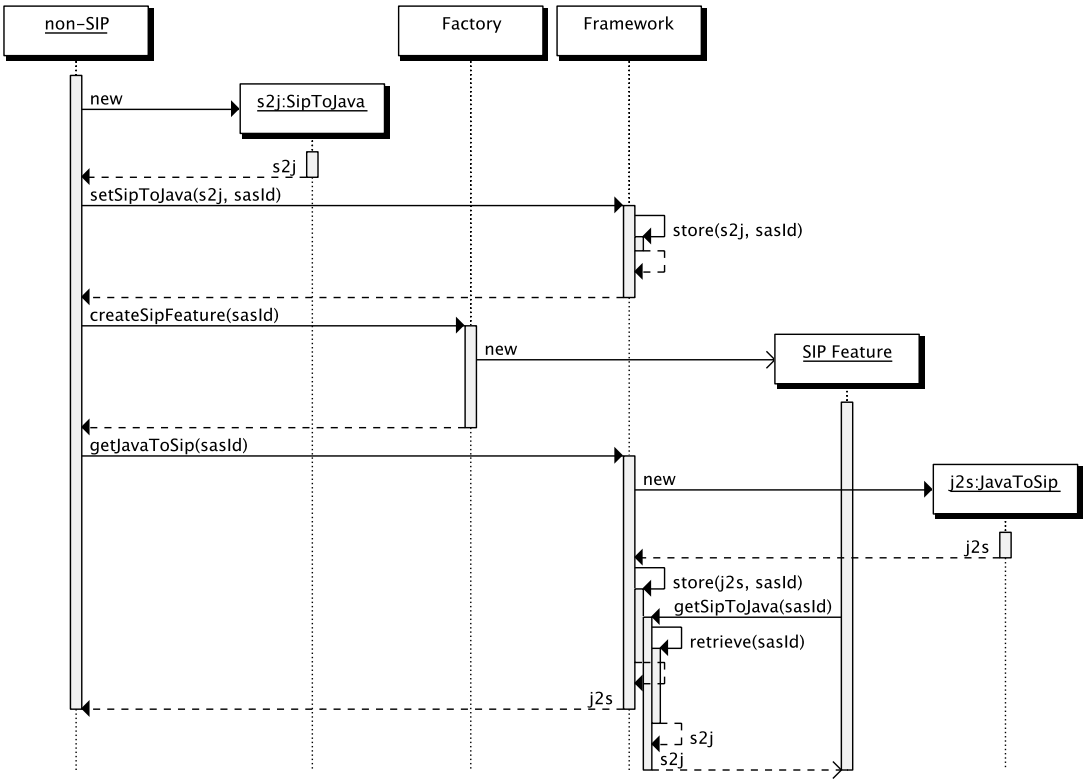

Figure 7: Non-SIP-initiated interface discovery - Scenario 2

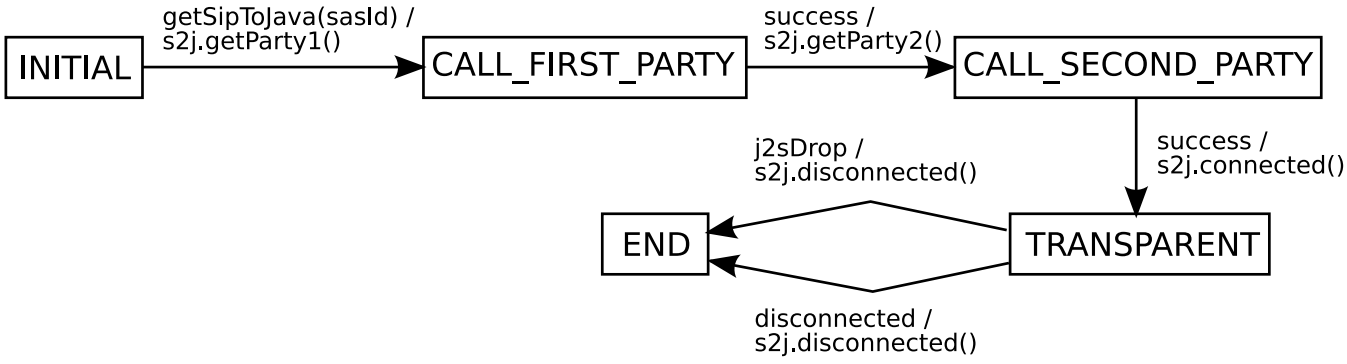

Figure 8: Click2Dial feature state machine

methods to indicate call status: calls connected or disconnected. Since the scenario of Figure 7 is assumed, we further assume that the non-SIP environment creates a SIP-To-Java instance that is initialized to include the data values for initiating the calls. This means that when the feature calls the SIP-To-Java methods to obtain the data values, the locally stored data values are returned. In contrast, if the scenario of Figure 6 had been utilized, then the same SIP-To-Java method calls would necessitate obtaining the values from non-SIP environment, a potentially slower operation.

The *event / action* syntax used for machine transitions denotes that when *event* occurs, then *action* is executed. The "sunny day" scenario for the feature is that it first obtains its SIP-to-Java interface instance from the framework and it obtains the first party address from the interface instance when it transitions from the `INIT` state to the `CALL_FIRST_PARTY` state. Upon successfully connecting with the first party, the machine obtains the second party address from the SIP-To-Java instance and transitions to the `CALL_SECOND_PARTY` state. Upon successfully connecting with the second party, the machine notifies the non-SIP environment that the two calls are connected and transitions to the `TRANSPARENT` state. At this point there are two paths to the machine's terminal `END` state. One path is traversed when one of the two parties disconnects. The other path is traversed when the non-SIP environment invokes the Java-To-SIP method for dropping the calls. In either case, the machine reacts by notifying the non-SIP environment that the two calls are disconnected.

4.2 MonitorControl

Given the similarity to the previous example, we only briefly mention an example of a converged, SIP initiated feature. Such a feature is MonitorControl. This feature specifies both SIP-To-Java and Java-To-SIP interfaces and therefore partakes in the discovery scenario shown in Figure 5. The MonitorControl feature monitors call state and provides control over call state similar to the ClickToDial feature. In particular, the feature notifies the non-SIP environment of current call state (e.g. ringing, connected, disconnected) via its SIP-To-Java interface, and drops any in-progress call in response to a Java-To-SIP method call by the non-SIP environment.

4.3 An Example Telecom Service

Figure 9 depicts an example telecom service that uses our framework. It consists of a number of telecom features, including the ClickToDial feature discussed in Section 4.1, that interact with the service's business logic via their respective Java-To-SIP and SIP-To-Java interface instances. The service also provides a web browser-based interface that can be used to indirectly interact with the telecom features. For example, the presentation and business logic could support the ability of a user to drop an in-progress call via the web interface: when the user requests to drop a call from the web interface, the business logic will invoke the ClickToDial Java-To-SIP interface instance "drop call" method.

5. REUSABILITY

The previous sections describe how our framework supports the development of converged telecom services using modular features, but not necessarily *reusable* features. It is common practice for non-SIP developers to avail themselves

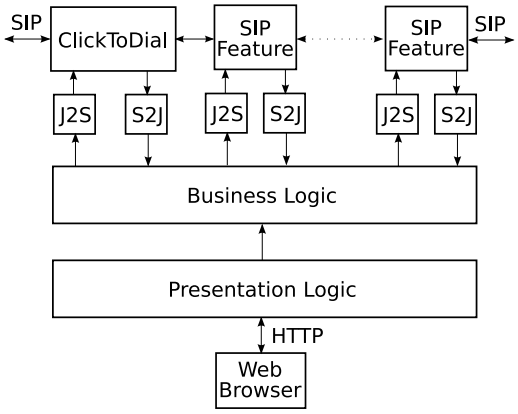

Figure 9: An example telecom service

of reusable components. This speeds development and improves system reliability and maintainability. However, there is no such standard practice for SIP servlet-based development because, as mentioned in the introduction, the SIP servlet 1.0 specification does not support feature convergence. But a version 1.1 container combined with our framework provides the infrastructure required to develop modular and reusable features.

Using our framework, a reusable feature is customized for use in a particular service context by providing a customized implementation of its SIP-To-Java interface. The implementation is free to implement the interface methods that are required and ignore the others. This way, the feature's interactions with external components are customized for those components. For example, the ClickToDial feature's SIP-To-Java interface includes a method that is called by the feature when a call is dropped. An implementation of this method for one service context may be to update a database entry when the method is called. An implementation for another service context may dispatch a Java Messaging Service (JMS) event. The advantage of using our framework is that the feature needn't be aware of its service context since all of its interactions with non-SIP components take place through its interfaces. Therefore only the SIP-To-Java interface implementation needs to be customized for a given service context, while the feature itself remains unchanged.

6. DIMENSIONS OF REUSE

To this point we have discussed modularity and reuse in terms of features that are composable using SIP Servlet 1.1-style application routing. In this context, we have shown how our framework supports customization of reusable features for use in different service contexts. However, we should mention that there are other options available for reuse.

One, finer-grained, option for reuse is the availability of reusable code from *within* a feature. For example, the JSR 309 working group is developing a standardized Java API for interacting with media servers [15]. The JSR 309 API defines methods and classes for performing common media server tasks such as establishing a conference or establishing an IVR session. An implementation of this API serves as a

reusable library of methods that can be called by a feature to perform media server operations.

Another fine-grained example is found in the ECharts for SIP Servlets framework [18]. This framework supports the notion of reusable state machine fragments: reusable parameterizable state machines that perform common functions such as making an outgoing call, receiving an inbound call, or switching a call between endpoints. ECharts for SIP Servlets machine fragments are intended to be embedded in a parent machine that defines overall feature behavior. As such a collection of machine fragments serves as a library of reusable telecom logic much as an implementation of the JSR 309 API would.

A coarse grained instance of reuse is the bundling of related features. For example, a feature that receives and processes a REGISTER request would naturally be bundled with a feature that receives an INVITE request for a user and forwards it to the user's current registration address. In carrying out its function the forwarding feature retrieves the data stored by the registration feature. The resulting bundle can be reused in application suites requiring registration/forwarding support. Using our framework it is also possible that the forwarding feature can be a customized instance of a reusable forwarding component whose SIP-To-Java interface includes a method to obtain the current address for a given user.

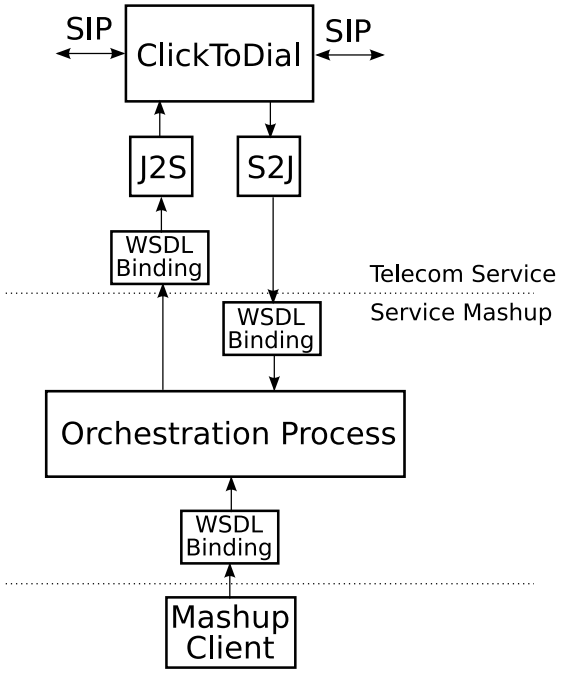

Figure 10: An example telecom service with a mashup interface

7. TELECOM SERVICE MASHUPS

A service mashup is an aggregate web service composed of a collection of simpler services. One classic example combines an HTTP-based mapping web service with data obtained from a regional real estate web service. The resulting mashup, displayed as a web page, graphically identifies houses for sale on a map. The notion of a mashup has been generalized to encompass any aggregate web service, whether it includes a web page-based interface or any other kind of interface like SMS or VXML. Mashups have become increasingly popular since they allow creative combinations of existing services to suit a customized need. Technology for creating a mashup has matured to the point where little programming skills are required (e.g. Yahoo Pipes [19]). The popularity of mashups has prompted one telecom analyst report to ask where the telecom mashups are [8]. The report cites British Telecom's Web21C project [10] and Microsoft's Connected Services Framework sandbox [16] as two promising efforts for supporting the incorporation of telecom services into mashups. The basic idea underlying both efforts is to provide a telecom service with a web services "wrapper" thereby making it available for mashing up with other web services. Once a component's web services interface is defined using, for example, the Web Services Description Language (WSDL) [2], web services can be composed (mashed up) with other web services using an orchestration process specified with, for example, the Business Process Execution Language for Web Services (BPEL4WS) [4], or Yahoo Pipes, which serves as an coordinating intermediary between the component web services and mashup clients.

One aspect of telecom mashups, then, is making a feature or a telecom service available as a web service. This can be achieved using a fine-grained, feature-oriented approach or coarse-grained service-oriented approach. To expose an individual feature as a web service, as exemplified in Figure 10, the feature's Java-To-SIP interface must be specified in WSDL and published, and the feature's SIP-To-Java interface should be implemented to call back the orchestration process's web services interface. To expose a telecom service as a web service, as exemplified in Figure 11, the telecom service business logic can expose methods via a WSDL-specified interface that call selected Java-To-SIP methods of its underlying features, as well as selectively propagate SIP-To-Java calls from the underlying features to an orchestration process's web services interface. The choice of which approach to use is a design decision outside the scope of this paper, however it is important to recognize that our framework supports both approaches.

As mentioned above, another aspect of the telecom mashup is composition, also called orchestration, of constituent services. While it is tempting to envision a web services framework where web services based on individual telecom features are selectively composed with one another as well as with other non-telecom web services, the reality is that the state of the art doesn't support this vision. There are two reasons for this: one technological and one fundamental.

The technological reason is that, for SIP-based telecommunications, SIP messaging proceeds on a separate signaling path to the web services orchestration control path. This means that the orchestration process cannot exert control over, or monitor a call at the session initiation level. To address this problem, one research project has experimented with abandoning SIP altogether in favor of using their own web services-based session initiation dialect [12]. Although this effort addresses the signal path problem it doesn't directly

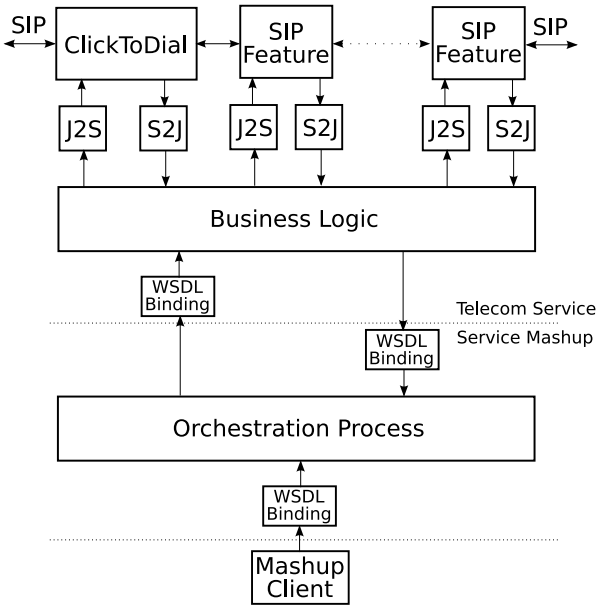

Figure 11: Another example telecom service with a mashup interface

address the service composition problem, the implication being that a suitable web services orchestration language such as BPEL would be used. However, apart from being a relatively heavyweight approach to compose telecom features, web services orchestration languages like BPEL don't support important notions necessary for telecom feature composition such as subscriptions, routing regions, or routing directives [6].

While there may exist less drastic approaches for addressing the signaling path problem, the more fundamental problem, not addressed in the literature to date, is that of composition constraints imposed by feature interaction [20, 14]. Unlike traditional web services, telecom features cannot be composed willy-nilly to produce a mashup. Feature interaction constrains how individual features can be composed to provide correct behavior. On the one hand, feature interaction can be used advantageously to compose complementary features. On the other hand, arbitrary composition of individual features may result in a mashup whose behavior includes undesirable feature interactions.

As an example of composing individual telecom web services to advantage, consider separate web service implementations of the ClickToDial feature, discussed in Section 4.1, and the FindMe feature. FindMe is a SIP initiated feature that, in response to an inbound call, initiates a series of outbound calls, either sequentially or in parallel, to a specified list of addresses. When one of the outbound calls is answered, it is connected to the initial inbound call and the remaining outbound calls are dropped. A simplistic web service implementation of ClickToDial might expose a *call* method that takes two parameters: addresses for the first and second parties. Invoking this method would have the effect of attempting to connect the two parties. A simplistic web service implementation of FindMe might expose a *setAddresses* method that takes two parameters: a subscriber address and a list of

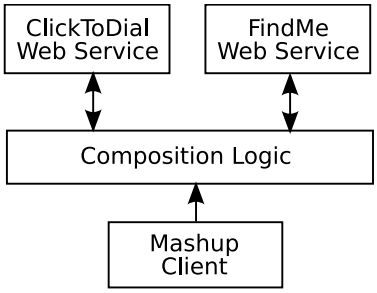

Figure 12: A ClickToDial/FindMe mashup

forwarding addresses that the subscriber may be found at. These two web services can be used individually but they can also be mashed up. A possible mashup, depicted in Figure 12, would permit a user to initiate a ClickToDial session to a second party without the user having to specify their own (first party) address. Instead, the mashup would attempt to contact the user using the list of addresses already provisioned via FindMe. The invocation of the FindMe feature by the Click2Dial feature would be handled by the mashup's composition mechanism. This example illustrates the desirability of composing individual telecom web services into mashups, but it also highlights the unanswered question of how to best compose these services. Given that the SIP servlet 1.1 specification includes comprehensive support for telecom feature composition via an application router, there arises a tension between the traditional approach to web services composition and telecom web services composition. How to manage this tension, however, is a topic for future research.

8. RELATED WORK

The Avaya (née Ubiquity) SIP Application Server [1] provides a proprietary framework called "SOOF" (Service Oriented Object Framework) that supports reusable, modular converged SIP services for SIP Servlet 1.0 containers. Because it is based on the SIP Servlet 1.0 specification, it relies on non-standard approaches to application routing and service composition. SOOF support for SIP-to-Java and Java-To-SIP interaction are tightly integrated with the greater SOOF framework. However, in the abstract, one can map SOOF Callbacks to SIP-To-Java interaction and SOOF Eventlets to Java-To-SIP interaction. Both forms of interaction require that the SIP entity define an interaction interface. SOOF Callbacks is a mechanism whereby a SIP component asynchronously broadcasts an event to a list of (non-SIP) listeners. The SIP component defines an interface specifying the events it can broadcast and the listeners are responsible for implementing the interface and adding themselves to the listeners list. SOOF Eventlets is a mechanism whereby a SIP component defines and implements an interface synchronously callable by non-SIP clients. While bearing gross similarities to our own framework, the major drawback to the SOOF approach is that it is a proprietary, embedded technology intended for SIP Servlet 1.0 containers. It is not clear how the technology could be extracted from the SOOF framework and applied to SIP Servlet 1.1 containers.

The SOOF framework also supports telecom service mashups,

albeit its support is tightly integrated with its proprietary approach to routing and composition. The SOOF framework only supports mashups of the form depicted in Figure 11; the framework does not permit publishing a web services interface for an individual feature. In the SOOF framework context, this constraint effectively prevents the external environment from affecting the composition of telecom features. As a result, the web services orchestration process is isolated from the telecom feature composition process. In addition to stand alone support for telecom mashups, the SOOF framework has been integrated with Microsoft's Connected Services Framework [16].

In 2006 British Telecom launched their Web21C project [10]. The project exposes a number of telecom features as web services suitable for creating telecom service mashups. The telecom features offered are conference calling, placing a call (click-to-dial), call flow (CCXML-like IVR [9]) and an SMS gateway. Unfortunately, the details of the Web21C project infrastructure are proprietary and it is not clear from the API documentation to what degree feature composition is supported, if at all.

9. CONCLUDING REMARKS

In summary, the proposed SIP servlet 1.1 specification is good news for system architects and designers: it offers the promise of developing converged services with modular telecom features. The light-weight framework presented in this paper delivers on this promise by providing a structured approach to interaction between features and non-SIP components and integrated support for discovery of features by non-SIP components and vice versa. The framework goes even further by supporting the development of reusable features and features suitable for inclusion in telecom mashups.

From a development process perspective, a SIP servlet 1.1 container combined with our framework facilitates the concurrent development of a telecom service where one team develops new features or incorporates reusable features and other teams concurrently customize SIP-To-Java interfaces and develop external components such as web applications and back-end business logic.

The framework described here is available as open source software as part of the ECharts for SIP Servlets Development Kit [13].

Acknowledgments

The authors would like to acknowledge the valuable contributions of our research team colleagues: Hal Purdy, Tom Smith, Venkita Subramonian and Pamela Zave, all of whom contributed to the development of the ideas presented in this paper.

10. REFERENCES

[1] *Ubiquity SIP Application Server 7.2 & Appcelerator SOOF Feature Pack 1.1 Developers Guide*. Available from `https://udn.devconnectprogram.com/support/user-guides`.

[2] *Web Services Description Language (WSDL) 1.1 Specification*. W3C, 2001. Available from `http://www.w3.org/TR/wsdl`.

[3] *SIP Servlet API Version 1.0*. JSR116, 2003. Available from: `http://www.jcp.org/aboutJava/communityprocess/final/jsr116`.

[4] *Business Process Execution Language for Web Services (BPEL4WS) 1.1 Specification*. IBM, 2007. Available from `http://www.ibm.com/developerworks/library/specification/ws-bpel/`.

[5] *Enterprise Java Beans Version 3.0*. JSR220, 2007. Available from: `http://jcp.org/en/jsr/detail?id=220`.

[6] *SIP Servlet API Version 1.1*. JSR289, 2007. Available from: `http://jcp.org/en/jsr/detail?id=289`.

[7] *SOAP 1.2 Specification*. W3C, 2007. Available from `http://www.w3.org/TR/soap12/`.

[8] Telco web 2.0 mashups: A new blueprint for service creation. *Light Reading's Services Software Insider*, 3(2):1 – 31, May 2007. Available from `http://www.networkmashups.com/docs/ssi_0507.pdf`.

[9] *Voice Browser Call Control (CCXML) 1.0 Specification*. W3C, January 2007. Available from `http://www.w3.org/TR/ccxml/`.

[10] British Telecom Web21C Site. `http://web21c.bt.com`.

[11] E. Cheung and K. H. Purdy. Application Composition in the SIP Servlet Environment. *Communications, 2007. ICC '07. IEEE International Conference on*, pages 1985–1990, 24-28 June 2007. Available from `http://echarts.org`.

[12] W. Chou, L. Li, and F. Liu. Web services for communication over IP. *IEEE Communications Magazine*, pages 136–143, March 2008.

[13] ECharts Web Site. `http://echarts.org`.

[14] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, 1998. Available from `http://www.research.att.com/~pamela/dfctse.pdf`.

[15] JSR 309 Media Server Control API. `http://jcp.org/en/jsr/detail?id=309`.

[16] Microsoft Connected Services Framework Sandbox Site. `http://www.networkmashups.com`.

[17] J. Rosenberg, J. Peterson, H. Schulzrinne, and G. Camarillo. IETF RFC 3725: Best Current Practices for Third Party Call Control (3pcc) in the Session Initiation Protocol (SIP), 2004. `http://www.ietf.org/rfc/rfc3725.txt?number=3725`.

[18] T. M. Smith and G. W. Bond. ECharts for SIP servlets: a state-machine programming environment for VoIP applications. In *IPTComm '07: Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*, pages 89–98, New York, NY, USA, 2007. ACM. Available from: `http://echarts.org`.

[19] Yahoo Pipes Site. `http://pipes.yahoo.com`.

[20] P. Zave. Audio feature interactions in voice-over-IP. In *Proceedings of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm 07)*, pages 67–78, 2007. Available from `http://www.lulu.com/content/985948`.