Experience with Modularity in an Advanced Teleconferencing Service Deployment

Eric Cheung
AT&T Labs — Research
180 Park Avenue, Florham Park, NJ, U.S.A.
cheung@research.att.com

Thomas M. Smith
AT&T Labs — Research
180 Park Avenue, Florham Park, NJ, U.S.A.
tsmith@research.att.com

Abstract

In this paper, we describe our experience with the design of an advanced teleconferencing service under two different frameworks — an early implementation of the Distributed Feature Composition architecture, and the SIP Servlet API. The usual design goals of software modularity for encapsulation and reuse are pursued. Interestingly, two very different designs resulted. This paper discusses the factors that influenced our design decisions. In particular, we examine the different characteristics of the two frameworks as well as the maturity of project requirements, and illustrate the ways in which these factors affect various mechanisms for achieving software modularity. We also aim to draw on this experience to propose a set of design guidelines for building modular, composable SIP Servlet applications for Voice over IP and converged services.

1. Introduction

This paper reports our experience designing and fielding a large-scale Voice over IP (VoIP) teleconferencing system. Our focus is the software modularity considerations we encountered when we developed the original system, as well as when we redesigned the system for a more standard framework. The initial design (Phase 1) used a proprietary application server and protocol, and drew heavily from a previous commercial deployment. A subsequent redesign (Phase 2) was for a standards-based application server. The nature of the VoIP application server ecosystem had matured considerably between design efforts, which had a surprisingly radical effect on the resulting designs. Effective software modularity was a design goal for both phases. This paper explores different forms that such modularity can take, and the considerations in choosing appropriate mechanisms in different contexts.

The teleconferencing service contains a number of familiar functions. The general user experience is described below:

- A potential conference user calls an advertised number, known as a *bridge number*. The call is answered by an interactive voice-response (IVR) system that prompts the user for information about the user's identity as well as the desired conference.
- Upon successful entry of the information, the user is switched from the IVR system to a conference bridge. At this point the user can hear and speak to other users who are on the same conference.
- During the course of the conference call, a user may enter special touchtone sequences that affect some aspect of the call. Examples include muting or unmuting the user's call leg, or temporarily switching away from the conference call to an IVR that offers a menu of other options (starting and stopping call recording, for example).
- Users may use a web interface to the conference call. The web interface shows the current list of participants on the call and provides a number of controls. These controls provide an alternative way to invoke all the functions available via touchtone sequences, as well as some extra functions that would be difficult to provide via IVR menus (such as blocking an individual participant who has called in from a noisy phone).
- The web interface also allows users to switch between conference calls if they are unfortunate enough to have more than one such call going on at the same time.

The work discussed in this paper focuses on the design of the telephony subsystem of the teleconferencing service. The context of the telephony subsystem is shown in Figure 1.

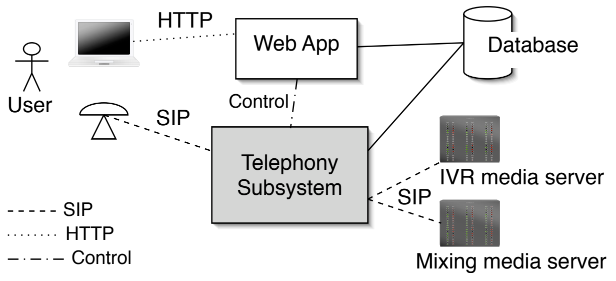


Figure 1. Context of the Telephony Subsystem in the Teleconferencing Service

Section 2 discusses the Phase 1 design in the context of the prevailing technology of the time. Section 3 covers the Phase 2 redesign effort, motivated primarily by advances in standards-based technology. Section 4 presents some analysis of the themes uncovered by the redesign effort, with implications for future design efforts in this domain. Section 5 offers some design guidelines for choosing between different modularity mechanisms. Section 6 summarizes our findings and discusses future work.

2. Design for DFC

The original service design was purely based on Distributed Feature Composition (DFC), a modular architecture for telecommunications services [7]. A *feature* is considered to be an increment of call-control functionality. DFC first introduced the notion of applying pipe-and-filter composition of individual feature modules to telecommunications services. The feature-modular nature of the architecture makes it very well-suited for managing feature interactions.

Composition of modules via pipes and filters is well known in software engineering [12, 9]. In the domain of telecommunications, this mechanism implies that modules are composed at call-setup time and communicate between themselves using a telecommunications protocol. DFC pipes and filters differ from those in classic data-flow formulations, in that DFC pipes can carry protocol messages in both directions, and DFC filters can also operate bidirectionally. DFC modules generally operate autonomously, and may not even be aware of the presence of other modules.

DFC also defines a routing algorithm that is used to dynamically compose feature modules into a *signaling graph*, which specifies the order and connectivity of these modules. When a module issues a call setup signal, the DFC routing algorithm calculates which feature should appear next in the signaling graph. The routing algorithm takes into account the *addresses* of the parties in the call, the *feature subscriptions* of each address, and any specified *precedence relationships* between features that are subscribed to by the same address. Since features can issue call setup or teardown signals at any point during a call, the signaling graph can change shape during the lifetime of a call.

Our work that reified the abstract architecture of DFC into a working implementation began in 1999, the earliest days of VoIP application development. At that time, the protocol landscape was considerably different than it is today. The dominant protocol at the time was H.323 [6], though the Session Initiation Protocol (SIP) had just released its initial standard [5]. DFC defines its own protocol, which has a number of properties that makes it attractive for application developers. Our architecture, as shown in Figure 2, contained features that communicated via the DFC protocol, and interfaces at the edges of the architecture to serve as protocol converters for interoperability with other VoIP protocols. Thus we created a DFC Application Server (DFC AS), capable of communicating with the outside world via multiple protocols.

In the intervening years, the influence of SIP increased greatly at the expense of H.323, becoming the dominant protocol for VoIP systems and vendors. SIP application servers began to appear, and by 2003, a standard for application development in Java via SIP servlets was standardized [8]. Commercial server offerings began to appear, and these systems provided advantages over our homegrown server in terms of performance, stability, and other operational issues. We chose to keep the runtime environment for our DFC features and jettisoned the multiple interfaces at the periphery in favor of embedding the feature runtime environment within a SIP servlet. The SIP servlet container provided the support for communicating via the SIP protocol with the outside world, as well as the operational efficiencies that come from using a commercial server. Our features did not need to change; they continued to use the DFC protocol to communicate among themselves.

When we began to design the components required to realize this teleconferencing system, we had not finalized the requirements. Indeed, a prototype with basic functionality was created first, and as we used the system internally, we began to add features iteratively. We were not always certain that new features would be as useful as we envisioned, so we generally chose to encapsulate such speculative new technology in standalone components, so that they could be added or removed with no change (or minimal changes) to the remaining components. The DFC routing algorithm takes care of this simply by using modified feature subscriptions, which are enumerated in a configuration file.

Another factor influencing the component design was the fact that we had recently completed a deployment of DFC technology as a commercial offering [3]. This deployment contained a number of reusable call-control components;

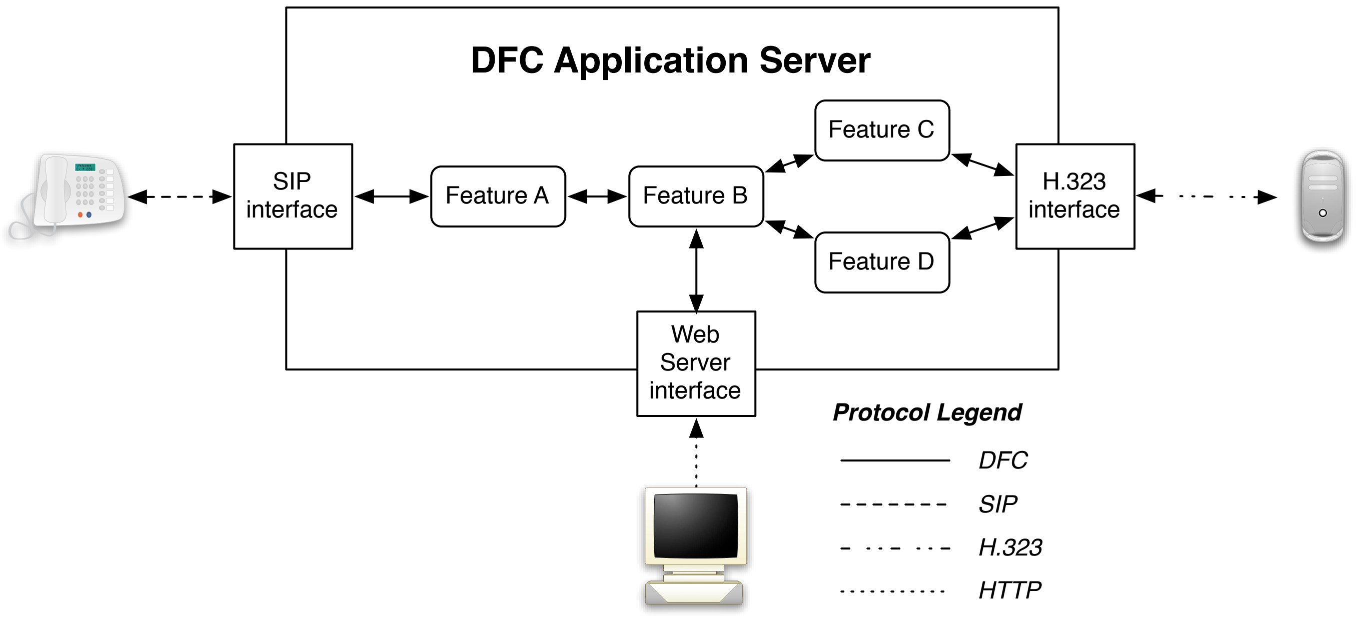


Figure 2. DFC Application Server Architecture

in particular it contained a simple conference calling capability. The existence of a sizable catalog of components that had been developed, tested, and deployed commercially made the idea of component reuse very attractive.

After months of iterative design, development, and user testing, we finalized the feature set. Some features that we had tried were eliminated from the final design, as the functions they provided were deemed not worth the risk to performance and stability for a large-scale deployment. Removal of these call-control features was trivial, as we only had to update the feature subscriptions to prevent these features from being inserted into the signaling graph.

A signaling graph for a typical successful call to a conference, reflecting the final feature design, is shown in Figure 3. The constituent features are described below. The order in which the features appear in the graph is carefully chosen for proper operation; however, with a few exceptions, the logic behind the ordering is beyond the scope of this paper.

In the sections below, we will use the following nomenclature:

- *User action* — an action taken by an end-user with the intention of modifying the behavior of the system. Examples user actions include pressing touchtone keys to mute the call, or clicking a button on a web page to hang up a call.

- *Touchtone signal* — a telecommunications protocol signal that conveys touchtones (entered via a user action) along the signaling graph.

- *Command signal* — a telecommunications protocol signal that conveys a high-level command (e.g. mute) along the signaling graph.

2.1. Interfaces

As noted above, features within the DFC AS communicate using the DFC protocol. As such, *interfaces* are required for communication with external (non-DFC) components, such as media servers, web servers, and gateways between SIP networks and the Public Switched Telephone Network. The interfaces required for the teleconferencing application are described in this section.

2.1.1. SIP Interface — The SIP Interface performs protocol conversion between the DFC and SIP protocols. This interface relies on a *SIP stack* to perform the required operations of parsing and formatting SIP messages, ensuring message orderings are protocol-compliant, etc. For Phase 1, we used the SIP Servlet container as a SIP stack by writing a protocol conversion application using the SIP Servlet API.

2.1.2. Mixer Interface — The teleconferencing application relies on an external mixing media server to provide specialized teleconferencing functions such as media mixing, playing of prompts, and call recording. The media server requires the use of a vendor-specific control language, conveyed via SIP protocol messages. A Mixer Interface is used to encapsulate this server-specific logic. The Mixer Interface handles all conferences in the system.

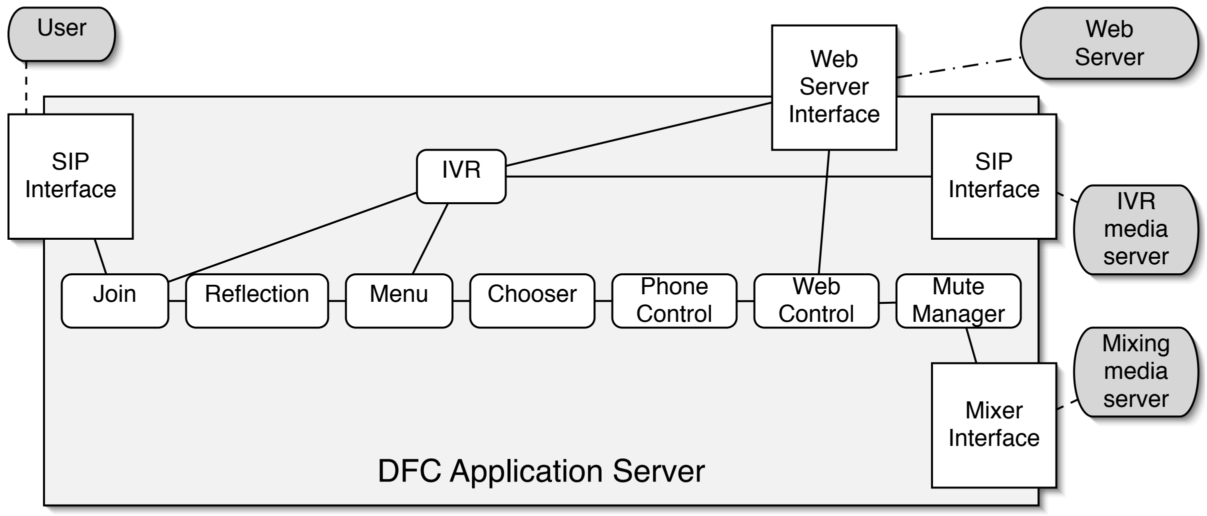


Figure 3. Signaling Graph When a User Is Connected to a Conference

2.1.3. Web Server Interface — In order to provide consistency for feature development, interactions with external web servers are modeled as DFC signaling channels. The Web Server Interface provides a centralized listening point for messages from an external web server, and can convey command signals to features via the DFC protocol.

2.2. Features

The feature list described herein is not quite exhaustive, but is comprehensive for the typical scenario of a user successfully dialing in to the conference bridge.

2.2.1. Join — The Join feature routes incoming calls to an IVR system to collect the information required to join a user to the desired conference. When the feature receives an *authentication* command signal from the IVR, it tears down the call to the IVR system and places a new call to the Mixer Interface, indicating which conference the user should join. After this point, the Join feature becomes quiescent.

2.2.2. IVR— The IVR feature serves as an adapter between the call-control features and the IVR system. It takes care of the protocol requirements for connecting to the IVR media server, which may differ between products from different vendors. In our case, the IVR systems at our disposal required different forms of the destination address in the call setup message. Employing an adaptor frees feature developers from understanding and implementing all the possible addressing requirements when interacting with an IVR system. This feature also provides the critical function of connecting to the web server interface to accept command signals (such as *authentication*), which it then merges into the signaling path back to the call-control features.

2.2.3. Reflector — The Reflector feature is designed to detect the receipt of certain signal classes, and to "reflect" such signals back in the direction from which they came. The rationale behind such a feature is discussed in Section 2.3.

2.2.4. Menu — The Menu feature reacts to a special touchtone signal coming from the user, indicating the user's desire to be presented with a menu of mid-call functions. When the feature receives this signal, it places the current conference on hold, and then connects the user to an IVR menu to control various aspects of the call. When the IVR exits, the feature reconnects the user to the conference.

2.2.5. Chooser — The Chooser feature allows a user to switch between multiple conference calls that are going on at the same time. On the user side of this feature, there is exactly one call (to the user's phone), but on the conference side, there can be multiple calls to different conferences. In this case, one of those calls will be active, and the remaining calls will be on hold for that user. Upon receipt of a command signal, the Chooser places the current call on hold and switches the user to a different conference.

2.2.6. PhoneControl— The PhoneControl feature reacts to certain touchtone signals coming from the user; when the feature observes these touchtone signals, it translates them into command signals (e.g., mute and unmute) that get sent along the signaling graph toward the mixing media server. This feature provides the user with a shortcut alternative to using the Menu feature for a few common functions.

2.2.7. WebControl — The WebControl feature provides a complementary function to the PhoneControl feature. Instead of issuing command signals based on received touchtone signals, it conveys command signals from the web server interface to the other features, based on corresponding user actions (e.g., start recording). It also reports call status back to the web application (e.g., this party has hung up).

2.2.8. MuteManager — The MuteManager feature provides the logic required for handling command signals to mute and unmute a user. In the previous deployment [3], touchtone signals were the only source of such commands, but when we added a complementary web interface, a new source of command signals was introduced, setting up the possibility of competing commands acting at once. A user may mute himself, but the conference host may also use the web to mute any user. If a user is host-muted, then the user is not allowed to unmute himself. During the prototype phase, there were yet more conditions that could also result in a user's being temporarily muted, though these were eventually dropped. This feature mediates the potentially competing commands, deciding under what circumstances such command signals should be propagated.

2.3. Design considerations

A number of the features described above already existed in our feature catalog (Join, IVR, Menu, and PhoneControl). This provided a strong incentive to reuse these features without modification, as they had gone through extensive field testing. In some cases, the decision to reuse these features dictated the need for a new component. For example, the Reflector feature was designed to complement the existing Menu and PhoneControl features. For the previous commercial deployment, touchtone detection was performed by the SIP interface, so these features had been designed to expect touchtone signals coming from the user. However in the current architecture, touchtone detection is performed by the mixing media server, and the Menu and PhoneControl features were not programmed to detect touchtone signals coming from this direction in the signaling graph. Thus a simple Reflector feature was developed to programmatically detect messages of a given type, and "reflect" those messages in the opposite direction. This allowed the other features to work as designed, and furthermore required no new feature interaction analysis to determine the appropriate ordering for features that could conceivably operate on the same signals.

Another factor affecting the system design was support for *application convergence*, i.e., applications with which the user interacts via more than one protocol. For our requirements, the user could interact via a telephone or through a web interface. However, the features in our catalog did not support interactions with the Web Server Interface. Rather than retrofit each of the existing features to handle command signals from the Web Server Interface, we chose to create a single new feature (WebControl) that served as a command signal dispatcher for all the other features. The WebControl feature was responsible for providing status updates to the web server, so that the web interface faithfully reflects the current status of the call, as well as accepting command signals from the web server, which the feature then sends along the call-control path. This approach allowed features such as MuteManager and ConferenceChooser to be developed without consideration of the interaction with the web server; they would simply look for commands coming along the normal call-control path.

With a composition of venerable and novel features, we were able to support the requirements for the teleconferencing service, which has been operating at a substantial scale for over a year. The system handles millions of minutes of calls on a typical workday. In the two years since a trial system has been deployed, we have observed on the order of ten (10) bugs in the feature code, which provides a measure of validation for our reuse-oriented approach, as well as the entire DFC architecture.

3. Design for Multiple SIP Servlet Applications

3.1. Motivation for Reengineering

After Phase 1 was deployed, we began investigating the feasibility of implementing the telephony subsystem of the teleconferencing system as a composition of multiple SIP servlet applications.

The SIP Servlet API supports application composition using a pipe-and-filter architecture similar to the DFC architecture. Multiple applications may be deployed to a SIP servlet container, and each application acts as a standard SIP entity. Version 1.1 of the specification [1] standardized the application selection process around the time Phase 1 was completed. This design of the application selection process is based heavily on the DFC architecture [4]. When a container receives a call setup message (INVITE request in SIP), it queries a special component called the Application Router (AR) to select an application to handle the message. When that application in turn sends a call setup message, the container queries the AR again to select the next application. This process is repeated until no additional applications are required and the message leaves the container, thereby establishing an application path. Subsequent messages within the call are then passed along the established path.

In Phase 1, we used a high-level, finite state machine based development tool based on the ECharts language [2] to write the DFC features. At the onset of Phase 2, we had already ported this to the SIP Servlet API, and named it ECharts for SIP Servlets [13]. Besides the close similarity of the DFC and SIP servlet composition models as well as the availability of tools, there were a few factors that motivated the reengineering effort:

- Developers writing features in the DFC AS must understand the proprietary DFC protocol. At the same time, the SIP Servlet API is the dominant standard for developing VoIP applications in the industry. In Phase 2 the development responsibility is transferred to an internal development organization with no prior experience with DFC. Therefore the reengineering will help with the technology transfer and ongoing maintenance.

- As the applications exhibit regular SIP behavior, they can be specified by SIP message flows, and can be unit tested with standard SIP test tools.

- While not mandated by the specification, SIP servlet containers typically support high availability cluster for fault tolerance and performance. In the cluster environment, the containers provide replication and persistence of application states, which in turn require the application states to be serializable. The DFC AS framework is too large and complex to be easily serialized, and therefore cannot take advantage of such features.

- Typically SIP servlet containers provide extensive operations support. For example they may support live upgrade of an application, or monitoring and management of application instances.

3.2. Modularity Mechanisms in SIP Servlets

There are three primary mechanisms for supporting software modularity in the SIP servlet environment at our disposal when we embarked on the redesign process:

- *Pipes and Filters* — This is the mechanism of composing SIP servlet applications using the AR as discussed above. The applications communicate between themselves using SIP messages.

- *Call Control Fragments* — The ECharts language encourages reuse by providing support for parameterized finite state machines (FSMs) that can be embedded in states of other FSMs. Over the years we have been creating telecommunications applications, we have identified a set of common call-control patterns and created reusable *fragments* of FSM logic that perform the required functions. Examples include terminating a call, switching a call from one endpoint to another, or behaving transparently. The application may control a call using multiple call control fragments concurrently, and may change the set of fragments controlling a call at any time. Developers can use these ECharts mechanisms to compose these well-tested fragments, along with other application-specific logic, to create the desired behavior.

- *Object Libraries* — The final modularity mechanism available is the familiar use of software libraries to encapsulate desired units of logic. In Java, our target language, the method for achieving such modularity is through object APIs.

3.3. The Redesign Process

A straightforward approach would be to rewrite each DFC feature into a SIP servlet application. However, our preliminary performance measurements on two independent SIP servlet container implementations suggested that the computational complexity increased with the number of applications invoked in the application path. Furthermore, there are a number of differences in the two environments that prompted us to rethink which modularity mechanisms to use in different situations.

We shall refer to the new architecture used in Phase 2 where multiple SIP servlet applications are composed in the container as the SIP Servlet API application server (SSA AS), to distinguish it from the DFC AS in Phase 1 where one SIP servlet application hosts multiple DFC features.

3.3.1. The External Interfaces — In the DFC AS, the SIP Interface is required to translate between DFC protocol messages and SIP Servlet API calls. In the SSA AS however, as the applications send and receive SIP messages directly, such translation is no longer necessary. Therefore, the SIP Interface is no longer required.

The SIP Servlet API supports convergence of SIP with HTTP or other non-SIP protocols. Each application that needs to receive commands from the web application can expose an interface directly. A convenient choice is to expose an HTTP interface, essentially making it a web service that provides complex call control functions to other components. Because of this new capability, the Web Server Interface, as a centralized point of interfacing with the web application, can be eliminated.

The functionality of the Mixer Interface is still required. This feature controls the mixing media server to create and destroy conferences, join and unjoin participants to a conference, play prompts to an individual participant or to an entire conference, and so on. It does so by sending control messages specific to the media server. The Mixer Interface is turned into the *Conference Manager* application.

3.3.2. Participant Leg and Conference Control — Next we considered the PhoneControl, WebControl and Mute-

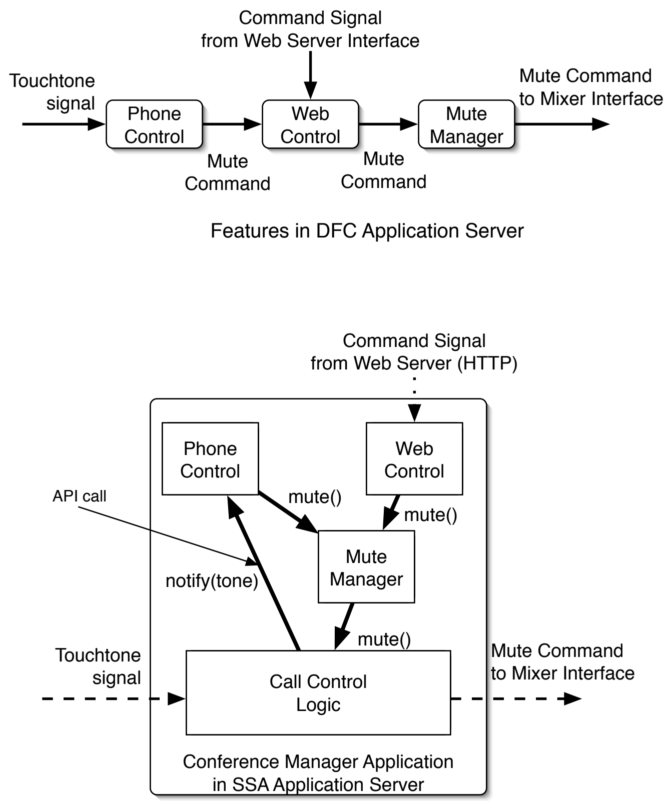


Figure 4. Reengineering of Participant and Conference Control

Manager features. Collectively, these features receive and interpret user commands and instruct the Mixer Interface to modify properties of the conference. In the DFC AS, these three features and the Mixer Interface exchange command messages amongst themselves. However, the features do not perform any call control functions themselves, nor do they necessarily care about the call state. Therefore, they are good candidates to be refactored from features directly on the call signaling path into software modules within another SIP servlet application.

Figure 4 shows these features incorporated into the Conference Manager application as Java objects. The Call Control logic, upon receiving touchtone signals in the signaling path, notifies the Phone Control object with a method call. If the touchtone sequence matches the mute command, Phone Control calls a mute method on the Mute Manager object. The Mute Manager maintains the mute state of the participant, and, if appropriate, makes a method call to the Call Control logic to instruct it to send a command to the mixing media server to mute the participant. Similarly, the Web Control object receives HTTP requests from the web application, and calls the Mute Manager.

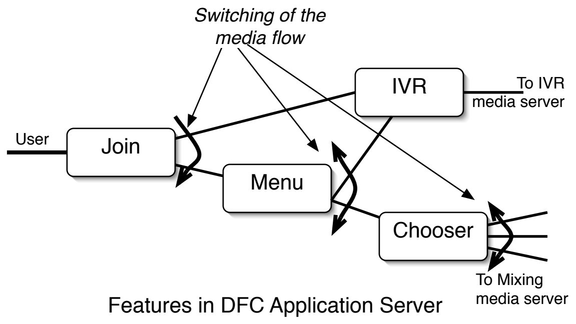
3.3.3. Caller Control — The Join, Menu, and Chooser features all perform similar functions — they connect the caller on one side to a new or existing call. It is interesting to note that in the DFC pipe-and-filter architecture proximity confers priority. For example, because the Menu feature is closer to the caller than the Chooser feature, if Menu decides to connect the caller to the IVR media server then the action of Chooser has no effect on the caller until Menu switches back to Chooser.

On the other hand, it is necessary for these features to coordinate their actions via the exchange of messages. Consider a particular scenario in which a participant triggers the Menu feature and leaves the conference momentarily to interact with the IVR menu. While this participant is on the IVR, the host ends the conference. The desired behavior is to notify the participant immediately, and ask him/her which conference to switch to. In this situation, when the Chooser feature receives a call teardown signal from downstream it needs to inform the Menu feature with a signal. After the Menu feature informs the participant of the new situation, and if the participant elects to switch to another meeting, then Menu will switch the participant from the IVR media server to Chooser, and send a command signal to instruct Chooser to switch the user to the desired conference.

The decision on how to reengineer these three features into SIP servlet applications is an exercise in managing complexity. On the one hand, combining these features increases the complexity of the call control logic. On the other hand, leaving them as separate applications introduces the complexity of inter-application messaging and dependency. Section 4 discusses call control fragments as another kind of reusable software module in this domain. By composing reusable fragments that handle switching, we were able to manage the complexity of the call control logic. Therefore, we settled on combining the functions into a single application called the User Manager, illustrated in Figure 5. However, it should be noted that the feature interaction management that is provided automatically by the feature precedence in the pipe-and-filter architecture is lost, and therefore the single application must now take care to give the Menu function priority over the Chooser function.

3.3.4. Interaction with IVR Media Server — In the DFC AS, the IVR feature is used by both Join and Menu to interact with an IVR media server. It adapts the destination address to a form appropriate to the type of media server, and sets the address of the media server. It also interacts with the web server interface, and sends the result upstream to Join or Menu.

As discussed above, in the SSA AS each application can expose an interface to the web application. Therefore the latter function is no longer required. As the address trans-



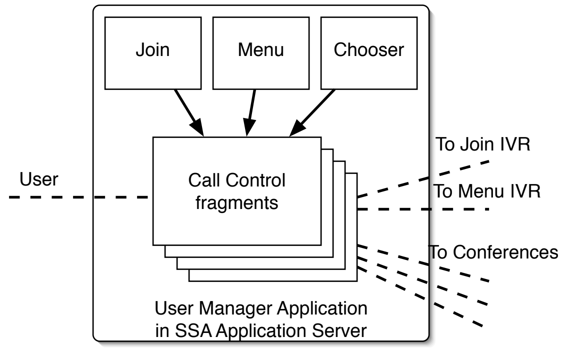


Figure 5. Reengineering of Caller Control

lation only requires certain fields in the SIP message to be modified and does not require sending or receiving additional messages, the address translation function can be simply achieved by a software module within each application. A cross-application configuration can be used to specify the type and location of the IVR media server.

3.3.5. Touchtone Handling — As discussed in Section 2.3, when the caller uses touchtones to trigger the menu or the mute/unmute functions, the touchtone signals logically originate from the user side. However, in this implementation they are detected by the mixing media server and sent from the opposite direction. Therefore, in the DFC AS the Reflector feature is required to reflect the signals such that the Menu and PhoneControl features can continue to expect the signals from the caller side. This also preserves priority of access to the signals.

When we began Phase 2 design, the requirements were more stable and it was known that there was no overlap in the signals that triggered the menu function (incorporated in the User Manager application) and the mute function (incorporated in the Conference Manager application). Therefore preserving the precedence of access to signal was no longer required. And as the reengineering required new applications to be developed, it was trivial to modify the code to expect signals from the media server side. Therefore we opted to dispense with the Reflector function in favor of a

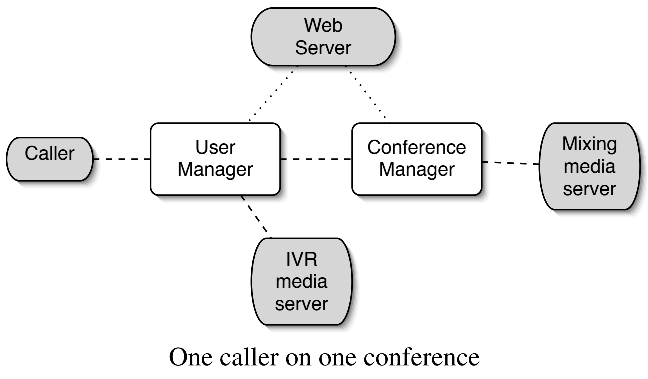


Figure 6. Final Application Design in the SSA AS

simpler design. It should be noted that this does require that future changes in the requirements do not introduce overlapping signal sets between the two applications.

3.4. Final Design

After the redesign, the application path of a successful call to a conference is shown in Figure 6. In comparison to the DFC AS feature graph in Figure 3, it is clear that much of the pipe-and-filter modularity has been changed to object and call control fragment modularity.

Keeping the User Manager and Conference Manager as separate applications is advantageous. First of all, as will be discussed in Section 5, the two applications serve different entities. Secondly, the separation helps manage the complexity. Figure 7 shows an application graph with multiple users, some of them on multiple conferences. It is clear that if combined into a single application the complexity will become unmanageable.

4. Lessons Learned in Modularity

Throughout the design lifecycle of this service, we have made extensive use of software modularity principles. The three primary mechanisms for modularity appropriate for this domain are discussed in Section 3.2. We have observed a number of instances where the modularity mechanism for a particular function has changed from Phase 1 to Phase 2. In this section, we consider the factors affecting the choice of modularity mechanism.

4.1. Pipe-and-Filter Modularity

The pipe-and-filter modularity mechanism, which was prevalent in the Phase 1 design, provided outstanding flexibility and excellent reuse characteristics. Many of the

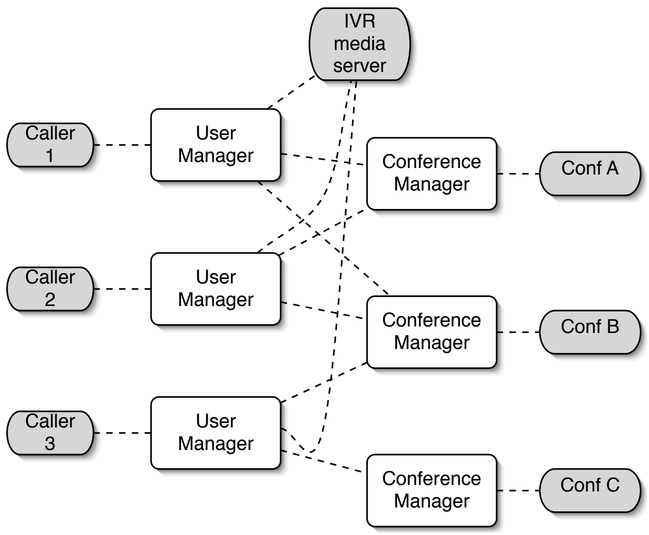


Figure 7. Multiple Callers on Multiple Conferences

Phase 1 features were reused as is from our previous commercial deployment, which reduced project risk. Because the DFC AS was designed to support pipes and filters, in some cases we only had to change a configuration file in order to alter the signaling graph to achieve new behaviors. During the prototyping phase, when different external components were being tested, we could employ protocol adapters (e.g., IVR) in the call path to insulate the features (e.g., Join and Menu) from the need to change to support different vendor implementations. Finally, using pipes and filters allowed us to use DFC principles to manage feature interactions [14].

4.2. Project Maturity

A good modularity design should confine and isolate likely changes to within a single module [11]. As the project matured, many sources of uncertainty disappeared. For example, the vendors for external components were finalized and the attendant protocol requirements were frozen. The interactions between features were all well understood, reaching a point where the rigor of DFC-style composition could be relaxed in favor of appropriately chosen object libraries. We reached the conclusion that a more mature project may require less modularity than a new project, or at least that the benefits of achieving modularity through configuration, as in the previous section, are less valuable later in the system lifecycle. For well-specified systems, achieving modularity at compile time (through call control fragments or use of APIs), rather than at runtime, may be perfectly satisfactory.

4.3. Design for Changeability

The above two sections discussed the varying need for flexibility in different phases of our project lifecycle. In some cases, deployed systems must still support a large degree of flexibility. This suggests that a new measure of *runtime changeability* may be useful in comparing pipe-and-filter modularity against the more static mechanisms of object libraries and call control fragments. Our teleconferencing system is rather stable — the IVR media server is unlikely to change, and furthermore all users and conferences are handled by the same application logic. As an example of a system with high runtime changeability, consider a service where each user can subscribe to different sets of applications. The pipe-and-filter architecture can compose the required applications for each user at runtime. Furthermore, when two or more applications interact, the feature interaction can be managed to yield a satisfactory outcome by the correct ordering of the applications in the signaling path. However, there may be more than one satisfactory outcome and different users may desire a different outcome. In this case, the pipe-and-filter architecture can impose a different ordering depending on the user. The changes in user subscription and ordering are isolated from the applications.

4.4. Differences in Protocols

In pipe-and-filter modularity, the ease and efficiency of inter-module communication depends greatly on the signaling protocol and the platform that implements it, which in turn influences the choice of the type of modularity.

The DFC AS guarantees delivery and ordering of messages, and acknowledgement of receipt is not required. As the features run within the same JVM, objects may also be sent in binary form. In contrast, because SIP is designed to be a general protocol for transmission over potentially unreliable networks, it builds in acknowledgement of receipt, retransmission, and handling of out-of-sequence messages. As well, SIP is a textual protocol and thus objects conveyed by SIP messages must be serialized into text. Therefore, in the SSA AS the inter-application messaging is more complex and expensive.

4.5. Performance Considerations

While the performance of a given modularity model depends greatly on the underlying platform, it is our experience that in the SIP servlet environment, performance degrades with more applications in the chain. The contributing factors include additional instances of application code, additional application and protocol sessions, context switching from one application to another, serialization and deserialization of messages, additional invocation of the application router, and so on.

On the other hand, while method calls between objects also incur additional overhead, JVMs are typically heavily optimized to reduce the performance impact [10]. Therefore, in terms of performance, modularity by object libraries and call control fragments (which really are specialized forms of object libraries) may be advantageous. Further investigation is required in this area.

5. Guidelines for Modularity in SIP Servlets

We hope that the lessons we learned in this exercise may be useful for other practitioners when designing large and complex services. There are many opposing influences at play, and there is really not a single best solution. However, it is useful to have all the modularity options at hand so they can be considered and evaluated on a case-by-case basis.

In the specific domain of SIP servlet application design, we have found that the following considerations have been useful in guiding our design decisions:

- Each application in the call path should perform call control functions and/or observe signaling events. Examples of call control functions are call rejection, branching, address translation, and tearing down an existing call. An example of an observer of signal event is a call logging application that logs the call connection time and duration. If an application does not fit either category, it may be better incorporated as a module in another application.

  This is the rationale for turning the PhoneControl, WebControl, and MuteManager features into object libraries within the Conference Manager application, as discussed in Section 3.3.2.

- An application either acts for the caller or the callee. Therefore, it is not advisable to combine two applications that act for different parties. This is the rationale for leaving User Manager and Conference Manager as separate applications. User Manager acts for the caller (the user of the conferencing service), and Conference Manager acts for the callee (the conference).

- If the call control functionality of an application is very complex and becomes difficult to comprehend or maintain, separating it into two or more simpler call control modules may help. This may be in the form of separate call control fragments, or separate standalone applications. In our case, the User Manager application has dual responsibilities and is rather complex — it first connects the caller to the join IVR, then after the caller is authenticated, the User Manager connects the caller to the menu IVR or one of several conferences. With the help of call control fragment libraries, the complexity can be managed.

- When adding a new capability, it is worth considering whether the new capability is general-purpose and may be reused in other composed services. If so, it may be useful as a standalone application that can be reused elsewhere, especially in the rapid prototyping phase. Another consideration is whether the new capability is very similar to that provided by an existing application. If so, the existing application may be generalized to serve different variations with slight configuration changes. The flexibility of the runtime composition by the application router may then take advantage of these new applications in different contexts.

- The ordering of applications in an application path helps manage feature interaction. When considering combining two or more applications, it is useful to evaluate whether the ordering of these applications needs to be flexible to provide different overall behavior in different services or for different users. Furthermore, if it is likely that another application may be inserted in between these applications, then combining them would preclude that possibility. The fundamental measure of cohesion is useful here. If an application performs several unrelated tasks, it is likely that these tasks may need to be reordered, or some other tasks may need to be inserted.

6. Conclusion

Many factors influence the choice of modularity scheme. As we took an existing system designed for one platform and reengineered it for a second platform, we experienced how the different characteristics of the two platforms affected our design choices. We also observed the tradeoff between the flexibility, ease of reuse, and feature interaction management offered by pipe-and-filter architecture on the one hand, and the performance advantages of the more static types of modularity by call control fragments and object libraries on the other. Furthermore, we saw that in different stages of the software lifecycle, the increased maturity of requirements and software implementation impacts the design.

The SIP Servlet API is currently the leading standard for programming VoIP and converged applications. At the time of writing, there are at least five commercial and three open-source SIP servlet container implementations available, together with a number of development tools and libraries. As more and more developers will be building large and complex services in this environment, good modularity practices become very important. We have begun to formulate a preliminary set of guidelines for designing for modularity in the SIP servlet environment. In our future work, by examining other complex services, we plan to continue to refine and expand this set of guidelines and hope to develop useful design patterns for this domain.

7. Acknowledgements

The considerable job of creating application server software, building complex applications, and establishing good design principles has been eased due to a rewarding, ongoing collaboration with our colleagues Gregory Bond, Gerald Karam, Hal Purdy, and Venkita Subramonian. Tom Everling provided keen insights into modularity tradeoffs throughout the application redesign process. Special thanks are due to Pamela Zave, not only for her theoretical prowess, but also for coaching us to prepare and improve this paper. Finally, the comments of our anonymous reviewers helped us to improve this paper.

References

[1] BEA. SIP servlet API version 1.1, 2008. Java Community Process JSR 289. http://jcp.org/en/jsr/detail?id=289.

[2] G. W. Bond. *An Introduction to ECharts: The Concise User Manual*. AT&T Labs, Inc, http://echarts.org/, May 2008.

[3] G. W. Bond, E. Cheung, H. H. Goguen, K. J. Hanson, D. Henderson, G. M. Karam, K. H. Purdy, T. M. Smith, P. Zave, and J. C. Ramming. Experience with component-based development of a telecommunication service. In *Proceedings of the Eighth International SIGSOFT Symposium on Component-Based Software Engineering*, pages 289–305. Springer-Verlag LNCS 3489, 2005.

[4] E. Cheung and K. H. Purdy. Application composition in the SIP servlet environment. In *IEEE International Conference on Communications*, pages 1985–1990, 2007.

[5] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: Session initiation protocol, March 1999.

[6] ITU. Packet-based multimedia communications systems. ITU-T Recommendation H.323, June 2006.

[7] M. Jackson and P. Zave. Distributed Feature Composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, XXIV(10):831–847, October 1998.

[8] A. Kristenssen. *SIP Servlet API*. Java Community Process, 2003. JSR 116. http://jcp.org/en/jsr/detail?id=116.

[9] D. Mennie and B. Pagurek. An architecture to support dynamic composition of service components. In *Proceedings of the 5th International Workshop on Component -Oriented Programming (WCOP 2000)*, 2000.

[10] M. Paleczny, C. Vick, and C. Click. The Java HotSpot^{TM}server compiler. In *JVM'01: Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium*, pages 1–12, Berkeley, CA, USA, 2001. USENIX Association.

[11] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[12] M. Shaw and D. Garlan. *Software Architecture*. Prentice-Hall, 1996.

[13] 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. ACM, 2007.

[14] P. Zave. Modularity in Distributed Feature Composition. In B. Nuseibeh and P. Zave, editors, *Software Requirements and Design: The Work of Michael Jackson*. 2009.