# Composable Chat: Towards a SOA-based Enterprise Chat System

Barry Demchak and Ingolf H. Krüger
*University of California, San Diego – Calit2*
*{bdemchak, ikrueger}@ucsd.edu*

# Abstract

*Enterprise Chat has emerged as one of the key tools for rapid communication, decision making, and situational awareness across a wide spectrum of application domains, ranging from massively multi-player online games (MMOs) to multi-national corporations to public safety and defense. This report is motivated by the increasing need to flexibly bridge multiple existing and emerging standards and technology platforms for Enterprise Chat. A promising approach to this effect is the use of Service-Oriented Architecture methodology and technology, as this combination has been used successfully across the industry in other enterprise integration projects. To determine the viability of this approach, we have performed a case study on the use of an SOA-based approach to unifying disparate IM/chat systems into a system-of-systems framework. This approach leverages enterprise-wide services, but without risking the IM/chat systems' existing functionality.*

*In collaboration with the Space and Naval Warfare Systems Command (SPAWAR), we modeled both a conceptual IM/chat system and existing IM/chat systems to understand the fundamental roles and interactions comprising these systems. We then organized these roles and interactions using a Rich Services [1] architectural pattern that defines a hierarchy incorporating an enterprise service layer and local system integration layers. The result is a SOA that informs the design of a system of IM/chat systems that not only meets the flexible integration goal stated above, but enables many practical operational affordances and allows governance across multiple administrative and organizational domains. Additionally, it points the way toward extending IM/chat systems to deliver richer and more effective content.*

*We have found that careful modeling of the Enterprise Chat domain, as well as of the individual standards and technologies, is one prerequisite to successful integration. As a second key prerequisite, we have identified the use of Rich Services to support principled, scalable integration while addressing many crosscutting concerns such as security, policy/governance, presence, and failure management. These lessons learned can be applied across all types of large enterprises that are characterized by multiple administrative and organizational domains, and that can similarly benefit from the integration of their disparate support systems. By supporting a system-of-systems integration, Rich Services has the potential of leveraging existing and emergent enterprise assets to unleash hidden enterprise value.*

# Table of Contents

## 1. Introduction

The purpose of the Composable Chat project we report on here is to investigate an architecture that supports the centralized administration of disparate chat systems. Numerous application domains use chat as a major means for communication and situational awareness. Examples range from massively multi-player online games (MMOGs), to multinational enterprises, to public safety and defense. In many cases, sub-organizations within larger enterprises have identified and deployed chat systems that suited their own requirements, and these chat systems have become mission critical. In many cases, such chat systems function well to facilitate communication, but have few facilities for authentication or authorization, and are inherently unsecure. In other cases, these facilities exist but duplicate or conflict with existing standards. Additionally, since chat systems are often local to sub-organizations (such as individual military commands), it is difficult for individuals outside of a sub-organization to enter that sub-organization's chat room, much less be authenticated, authorized, and shown present while doing so. Finally, by the nature of these chat systems, communications are fundamentally text based and, therefore, do not incorporate rich content critical for imparting situational awareness efficiently.

The Composable Chat project studies a pathway to solving these problems by creating a chat system consisting of chat systems – a system of systems called Composable Chat. Such a system would be able to provide authentication, authorization, and presence services across administrative and organizational boundaries; would be administrable both at an enterprise level and at the individual sub-organization level; would remain unchanged at the level of the individual sub-organizations; and would interface effectively with other chat systems. It would also provide a means for communications to include rich content beyond existing text-based streams. To address high demands of agility during both development and deployment, as well as interoperability demands across the enterprise, a Service-Oriented Architecture (SOA) based approach is proposed for Composable Chat.

Together with our partners at SPAWAR, we have chosen Enterprise Chat involving military commands as the basis for our exploration. In this context, modifying existing chat systems (IRC, XMPP/Jabber, and SIP/SIMPLE along with compatible clients) to provide the above-mentioned features is technically feasible, but operationally infeasible – the present and future costs of deploying and maintaining these modifications are prohibitive. Instead, our approach is to build a Composable Chat system, which would consist of the existing chat systems and middleware that organizes the chat systems so as to incorporate and take advantage of system-wide authentication, authorization, directory, and presence facilities, similar to the DoD's Core Enterprise Services. In this approach, the existing systems would operate largely untouched, and the Composable Chat system would be built around them.

The key challenges in creating a Composable Chat system are:

- Diversity of existing chat systems and users
- Monolithic nature of existing chat systems
- Adding content- and context-rich messaging
- Integration into workflow management systems

While the first two challenges reflect concerns posed by existing chat system deployments, the last two challenges reflect concerns likely to arise in the near-to-intermediate future, and therefore

are worth immediate consideration.

Creating a service-oriented Composable Chat system architecture involves first creating a clear understanding of the architecture of constituent chat systems, then incorporating them into a SOA that provides services that are missing in each chat system. The approach we took for this case study has several steps:

- create a conceptual model describing (the desirable features of) chat systems in general,
- create a domain model for each chat system,
- identify services that implement the interactions between the different roles identified in the conceptual model,
- identify the differences between the conceptual model and the concrete domain models,
- identify the bridging services that must be added to the concrete systems in order to reconcile them to the conceptual model,
- construct Rich Services [1] using combinations of concrete systems and the bridging services,
- demonstrate concrete deployments that realize the Rich Services.

The result demonstrates how existing chat systems can be incorporated into a structure that delivers authentication, authorization, directory, and presence capabilities that can be administered at multiple organizational levels, and allows the delivery of rich content (called Rich Messages). Additionally, the system can also include services for failure detection/mitigation, logging, usage statistics, flexible bandwidth usage, disconnected operation, multilevel auditing and information accountability, language translation, and other services as the need arises. Finally, the system can be deployed without disturbing or risking existing chat functionality, and can be made robust, performant, and maintainable.

The result of this exercise is a set of use cases, domain models, and an architecture that identifies a roadmap for the integration and improvement of individual chat systems toward the fulfillment of enterprise objectives.

### 1.1. Contributions and Outline

The major contributions of this paper are to demonstrate how SOA techniques can be used to build the Composable Chat system using existing disparate chat systems, and to discuss the properties of such a system.

In support of the SOA design process, a conceptual model of a generalized chat system along with deployment models of existing chat systems are presented. These models form the basis on which the Composable Chat system model is built and its properties are understood.

Besides enabling communications between members of different chat systems, the Composable Chat system incorporates new services (including authentication, authorization, and directory), extends existing chat presence systems into the enterprise, and extends chat systems themselves to handle Rich Messages instead of simple text messages. These new services can be either standalone, or can leverage an existing Core Enterprise Services infrastructure, or both.

Finally, the Composable Chat system is designed using a Rich Services SOA pattern, which

defines chat management hierarchies that allow for the definition of policies and governance by various administrative and organizational authorities.

The presentation of our case study is structured into sections that explain the Composable Chat challenge, our SOA-based integration approach, and examples showing the results of applying the approach to particular chat systems. In Section 2, we discuss the challenges of Composable Chat, as well as the underlying assumptions for our case study. In Section 3, we discuss the fit of SOA for the integration challenge. Section 4 introduces Rich Services as the architectural blueprint we use for integrating existing and emerging chat systems into systems of chat systems. In Section 5, we present detailed models for Enterprise Chat in general, and IRC, XMPP, and SIP/SIMPL as representatives of particular chat standards/technologies. This prepares our discussion of a sample integration architecture based on the presented models and architecture blueprint in Section 6. In Section 7, we exemplify this integration architecture by composing IRC and XMPP at the conceptual architecture level. Our summary and conclusions appear in Section 8.

## 2. Challenges and Assumptions

Composable Chat requires attention to a broad range of requirements ranging from a diverse stakeholder set with existing and emerging chat technology standards and platforms to the desire to integrate new presence and other situational awareness features into the chat infrastructure. In this section, we discuss some of the resulting challenges, as well as the underlying assumptions we used for our case study in the context set by our partners at SPAWAR.

### 2.1. Challenges

***Diversity* of systems and users** – Various commands have chosen and deployed chat systems (including server and client components) based on their own needs and without necessarily coordinating with other commands. The result is a diverse collection of chat systems (IRC, XMPP, and possibly SIP/SIMPLE), which themselves have multiple suppliers, multiple configurations, and multiple flavors. While it is plausible that some commands could tolerate the turmoil inherent in replacing working systems with standardized platforms, these systems are mission critical, and the risk of non-mission-related changes is unacceptable. Additionally, because the existing chat system servers and clients are deployed in so many locations and situations, and because they are present in multiple distribution pipelines, direct replacement or upgrade of these components is difficult and probably infeasible.

The challenge is to incorporate these systems into a common operating infrastructure while leaving them operating in place for the immediate future (and upgrade or replace them only as opportunities arise, and which don't put their mission at risk).

***Monolithic* (not open) nature of existing systems** – Existing chat systems appear to be designed to be self-contained – that is, they do not expose service interfaces for use by outside entities for the purpose of peering and composition. While published protocols exist for server-to-server and server-to-client communications, the protocols are positioned for use by builders of chat system components and are somewhat vague and subject to deviations. Consequently, interfacing to these systems from the outside presents a number of risks, including bugs due to an incomplete understanding of the underlying system components, bugs due to variations in system component implementation, and bugs due to implementation characteristics changing as chat system versions change. Peering and composition is further limited when chat systems themselves have limited scalability, which is the case (to some degree) with each chat system.

A subtle but important contribution to the closed nature of existing chat systems is provided by their deployment model. All chat systems rely on server and client software installed on forward-deployed server and client hardware. The resources for identifying, upgrading, and replacing such systems are thin and are unlikely to support any kind of mass migration (supposing that suitable migration targets could even be identified). Consequently, chat systems in the field must be considered to be static and unmodifiable through traditional software development means.

The challenge is to affect these systems so as to incorporate them into a common operating infrastructure, but without breaching their operating integrity or undermining their deployment.

***Rich Messaging*** **instead of domain-neutral messaging** – Existing chat systems attempt to convey situational awareness through the exchange of text messages. While text messages are fundamental to communication and are thrifty with bandwidth, they do not efficiently convey situational awareness when information is contained either in non-text forms or in the correlation of disparate information elements.

Structured multimodal messaging (called *Rich Messaging*) has the promise of reducing the time needed to understand, analyze, and respond to a situation. It is a "way to reduce the ambiguity of chat through an increase in structure, data quality and fidelity – follows the OHIO principle: Only Handle Information Once … [is a] basis for enterprise discovery of collaboration state, outcomes, expertise, availability … [provides] context for agent participation during collaboration" [8]. The properties of Rich Messaging hold special value in an environment where multimedia content can help clarify ambiguous situations currently underserved (or provoked) by existing text translation solutions.

One example of such messaging is a text message declaring that an asset is in need of repair, delivered with a map showing the asset in the context of geography and other available assets, and a tasking priority of the repair of each asset. The objective would be to enable the reader to make a quick and accurate decision regarding whether and how the asset should be repaired, thereby eliminating multiple laborious text-based exchanges and negotiations.

The challenge is to create a message format that contains structured multimodal information, delivers information scaled to the available bandwidth and the security characteristics of the chat operator, and can be implemented on the relatively closed chat systems and clients currently deployed.

**Addressing** ***workflow integration*** – Existing chat solutions address workflow in an ad-hoc manner, trusting that the chat participants can recognize a need, then deliberately cooperate in understanding the process for addressing the need, and again cooperate in solving the need. For complex, timely, and mission critical workflows, this process leaves room for significant omissions, especially when high priority events force chat participants to turn their attention elsewhere. Additionally, tracing and managing such workflows is very difficult because chat threads can rarely be automatically associated with any particular workflow.

Existing workflow solutions are capable of orchestrating complex interactions, but are cumbersome because they use e-mail and other systems that are not immediate.

The opportunity exists to combine the immediacy of chat-based systems with the process management characteristics of workflow systems. Such a combination could increase the number of decisions that could be made quickly and accurately, while also improving traceability and manageability.

Additionally, while some existing chat systems have capabilities for value-added workflow processing built into their own agent and service facilities, such processing is most effectively applied within a particular chat system. While it is possible to design and implement a distributed workflow system as services residing in each chat system, such a design, implementation, and deployment would be time and resource intensive.

The challenge is to interface existing chat systems with existing workflow solutions, given the relatively closed and disparate nature of existing chat systems and the difficulty of deploying to them.

### 2.2. Assumptions

A proper use of SOA can use existing chat systems to create the Composable Chat infrastructure while simultaneously addressing its challenges, subject to certain assumptions:

- For a chat system incorporated into the infrastructure, users logged on to the chat system's servers will continue to use the chat system in the same way as they would in a standalone chat system.

- The business rules enforced by chat systems incorporated into the infrastructure will remain the same as they would were the chat system executing as a standalone system.

- The infrastructure will provide a concept of identity where each user is represented once, but may have multiple roles. Each role may be used with one or more of the different chat systems.

- Users of a chat system *may* be authenticated within the chat system (subject to the chat system's rules), and *must* be authenticated within the infrastructure (subject to the infrastructure's rules). One policy that can be adopted by the infrastructure is that users authenticated by a particular chat system are considered to be authenticated for the purposes of the infrastructure – the infrastructure can choose to trust the chat system's authentication procedures.

- Users of a chat system may subscribe to presence information for any user within the chat system (subject to the chat system's rules), and to information for any user outside of the chat system (subject to policies defined at the enterprise and command level).

- Users of a chat system may communicate with chat rooms, individuals, and services within the chat system (subject to the chat system's rules), and with similar resources outside of the chat system (subject to policies defined at the enterprise and command level).

- For each chat system, it is possible to identify service interfaces that can be exploited as bridges between the chat system and the infrastructure. Such bridges would allow the infrastructure to monitor the status of the chat system's users, chat rooms, and similar resources. Additionally, such services would allow the infrastructure to inject information collected from other chat systems, thereby extending access to those chat systems.

## 3. Services and SOA

Our approach to the design of the Composable Chat system involves the characterization and

analysis of chat-oriented services in a Service-Oriented Architecture (SOA). In this section, we discuss the benefits of SOA in the context of Enterprise Chat and the challenges we have identified above.

In the popular press, the term *service* is often shorthand for a Web Service, and the term *Service-Oriented Architecture* involves the use of Web Services to create Internet-based applications. Web Services leverage stacks based on existing technologies (e.g., HTTP/SOAP for transport/messaging, XML for data marshalling, WSDL for interface description, and UDDI for discovery) to create standards-based interactions between code entities. While these facilities enable the construction of distributed and loosely coupled systems on the Internet, they do not address the more generic problems of understanding the relationships between entities, and designing systems that effectively and reliably leverage these relationships.

Our use of these terms is more basic and generic, though they encompass the popular usage. A *service* is defined by an interaction among the entities involved in establishing the service [2], and can be discussed independently of the particular technologies used to implement it [3]. A service description focuses on well-defined roles played by the entities and the interactions between them. A number of different entities can participate in a service, provided they play the roles defined for the service. For example, in a bank deposit service where a customer deposits money in a checking account, either a business or a person can play the role of the customer, and a commercial bank or brokerage can play the role of the bank account. The interaction would be the deposit of money.

A *Service-Oriented Architecture* (SOA) is an architectural style that models system functionality as a collection of roles and the interactions between them; creating a SOA involves identifying roles and the services that include them. A SOA can model either at the conceptual level or the deployment level.

At the conceptual level, a SOA models roles and interactions independently of how they are implemented or deployed. A well-built conceptual model decomposes a system into well-defined, encapsulated, and extensible services that contribute to the immediate and long term business goals of the system's users. Additionally, careful attention to service definition can lead to separation of concerns and the identification and modeling of crosscutting concerns, thus contributing to long term system reliability, extensibility, and maintainability.

Examples of crosscutting concerns in the Composable Chat context include security, policy evaluation, and governance. More subtle examples include activity logging, threat assessment and avoidance, auditing and information assurance, and failure detection and mitigation.

At the deployment level, a SOA models services as interactions between loosely coupled components that implement the roles modeled at the conceptual level. Such components are self-contained, thus encouraging component-level interoperability and adherence to standards. Communications between components in a SOA is often carried out via messages, though messaging is not a requirement of SOA. The combination of messaging and standards compliant self-contained components enables a great deal of flexibility in deploying components and managing their interactions. Such components can be deployed on the same server or on different servers, depending on the requirements of the system – they can even be deployed at different times, given sufficient queuing mechanisms. Given a means of identifying components and their function, a system can dynamically discover components that can fulfill a service function, then choose amongst them using its own criteria, which may include speed characteristics, space requirements, proximity, cost, or particular characteristics of its functional execution.

At both the conceptual level and the deployment level, SOAs provide value by encouraging manageability, scalability, dependability, testability, malleability, interoperability, composition, and incremental development. Owing to the well-defined nature of services, roles, and the components that implement them, it is possible to reliably embed a SOA into other frameworks (which themselves may or may not be SOAs), embed other frameworks within a SOA [4], and decompose and reuse portions of a SOA in other contexts.

While the Web Services infrastructure provides facilities for the implementation of a SOA, it is not the only infrastructure capable of supporting SOA. To some degree, chat systems employ a network and their own standards to support messaging between self-contained, well-defined components such as chat clients and chat servers. Thus, it is natural to discuss chat systems as limited examples of SOAs and to extend the discussion back to a conceptual level SOA.

For example, given a chat system modeled as a SOA, it is possible to embed the chat system as part of a larger, more capable chat system either at the conceptual or the deployment level. Also, given such a chat system, it is possible to embed into it services (at the conceptual level) and components (at the deployment level) which add value to the chat system using the chat system's own service definitions, yet perform functions that are novel or which enable its participation in a larger chat system. Finally, portions of either a conceptual SOA or a deployment SOA can be readily extracted and validated or tested in isolation, thereby improving reliability and dependability. Likewise, services and components can be readily recycled into yet other systems at very low cost.

Note that chat systems characterized as SOAs are conceptually and deployment-wise compatible with systems implemented using Web Services or other frameworks. Consequently, a chat system or a composite chat system could be managed by administrative systems which themselves use Web Service orchestration and choreography technologies (e.g., BPEL [6] and WSCL [7]) to create and manage workflows both within chat systems and for the sake of administering chat systems.

## 3.1. SOA Capabilities

At both the conceptual and deployment levels, SOA can affect services where roles and interactions between roles are well-defined. When this is true, modeling techniques allow existing systems to be modified in a number of ways.

#### At the Logical Level

**Services can be modified** by inserting roles into interactions between existing roles, and allowing the new roles to intercept the service interactions. Such interceptors can have their own interactions with the other roles, subject to the interactions remaining within the capabilities defined for each of the roles. For example, in the bank deposit service described above, it would be possible to limit deposits to $100 by inserting a new role that would accept the deposit from the customer, test the deposit amount, then either forward the deposit to the checking account or return the deposit to the customer. Such a modification would be possible only if the customer role was able to process a rejected deposit.

**Services can be eliminated** by inserting a role into an interaction, but defining it to quash further service interactions. For example, instead of creating a filtering interaction for the bank deposit service, it is possible to create a new role that rejects all deposits, thereby effectively eliminating the bank deposit service.

**Services can be created** by allowing new or existing entities to perform roles germane to existing services, yet have new interactions with still other roles. In this way, new services can be composed of existing services. For example, a payroll deposit service can be created by combining the existing bank deposit service with an existing company payroll service. The company would be allowed to play the roll of the customer in the bank deposit service for the purpose of depositing payroll checks.

#### At the Deployment Level

**Services can be modified or eliminated** by replacing them with implementations having different or additional functions or no function at all.

**Services can be created** by creating new implementations that interface with existing services using existing protocols, but which expose additional protocols that support additional interaction patterns.

At the conceptual level, a SOA can be used to alter or enhance an existing system when roles and the interactions between the roles are known. Similarly, at the deployment level, a SOA can be used when the components and the protocols between the roles are known, and the messages exchanged between them can be intercepted.

Depending on the deployment technology, SOA can be used to alter or enhance a system even as it is executing through combinations of service discovery and dynamic binding technologies.

### 3.2. Application of SOA to Challenges

**Diversity is Rationalized** – For each chat system, service interfaces can be identified and exploited as bridges between the chat system and the infrastructure. Such bridges will allow the infrastructure to monitor user presence and the status of chat rooms and similar resources. Additionally, similar services in other chat systems will allow the infrastructure to inject this information into those systems.

To the extent that such services exist, the infrastructure will be able to propagate presence and service information between chat systems. In order to depend on these services, a determination must be made that they are well characterized and unlikely to change as subsequent revisions of the chat system are released and installed. Even so, a service interface that suffices for one variant of a chat system may not exist in the same form (or at all) for other variants. A means must be devised for testing a given chat system's service interfaces to determine that they support all of the interactions required by the infrastructure, and that exercising these interactions has exactly (and only) the desired effects within the chat system. (This type of testing is beyond the scope of this paper.)

Additionally, as the Composable Chat infrastructure presents a consistent picture of presence and communications availability across all chat systems, an issue of common semantics arises when integrating chat systems having different properties. For example, when integrating two IRC chat systems, the infrastructure can represent all of the properties of a chat room in one system as available to users in another system – voice-enabled chat rooms could be available to users of both chat systems. However, when integrating an IRC chat system with an XMPP chat system, the infrastructure cannot represent all chat room properties in one system to users of another – XMPP users have no concept of voice-enabled chat rooms, and therefore cannot make proper use of them.

The semantics issue can be solved by representing presence and communication capabilities in a neutral, lowest-common-denominator fashion, or in a semantically rich fashion that trusts the service adapters for each chat system to map the other chat system's capabilities to its own as best it can. Arguments can be made for either scheme, and questions of this sort cannot be solved in the SOA domain. (Settling this issue is beyond the scope of this paper.)

**Workarounds to Avoid Monolithic Solutions** – While the issue of service adapters was treated in the Diversity discussion above, there remains the challenge of deploying enterprise infrastructure support code that implements the infrastructure and service adapters. It is likely that some part of this code should be collocated with the existing chat systems, while other parts of the code might be located on the same servers or on yet other servers. While modeling the particular location and operation of such code is under the purview of a SOA, the logistics of deploying and redeploying it are not. In order to avoid the pitfalls of deploying code without being able to easily update it, a system must be defined whereby code can be deployed and redeployed without risk to the underlying chat systems' operation. While such a system design can be created as a SOA, use of a SOA doesn't automatically solve this problem. (Such a solution is beyond the scope of this paper.)

**Domain Isolation Overcome** – A SOA can help implement a Rich Messaging system at two levels: the dissemination level and the composition/viewing level.

At the dissemination level, a Rich Message is forwarded to individual recipients. At its heart, message dissemination is a service interaction between the entity holding the message and the entity receiving it. A service that delivers a Rich Message must determine which parts of a message a particular recipient is authorized to view, then forward only those message parts. Such a service must be deployed in a trusted resource such as a chat server or an infrastructure server, and a particular deployment would be subject to policies defined at the enterprise and command level.

At the composition/viewing level, it may be possible to create a new, integrated message browser as defined for Collaborative Data Objects (CDOs) [8]. However, this violates assumptions already stated in this paper. Instead, a Composable Chat-related service residing on the client would stage interactions between the network and various discrete chat-related client-resident programs, and would dispatch a particular message segment to a particular viewer according to a predefined mapping. For a client to take advantage of Rich Messages, it would have to have installed the requisite dispatch service and viewers, subject to the deployment concerns described for the Monolithic challenge. Regardless, the dispatch service and viewers would not preclude the continued use of the existing text-based client software, and would act as compliments to it.

Note that SOA does not create the dissemination or composition/viewing services, but these services would be designed and deployed under a SOA discipline. (The definition of these services is beyond the scope of this paper.)

**Workflow Integration Achieved** – Given a SOA approach to Composable Chat, the problem of integration of chat systems with existing workflow engines can be tackled by modeling a composition of services. Roles contributed by the Composable Chat system would include recipients and communication channels (e.g., chat rooms and IM), while roles contributed by the workflow engine would include workflow databases, sequencers, and managers. The composed service would be defined by the interactions amongst the roles. This would be possible even if the workflow engines themselves are not built using SOA disciplines, so long as it is possible to identify exploitable service interfaces within the workflow engines.

This is a similar problem to that posed in the Diversity discussion. To the extent that exploitable service interfaces can be discovered in a workflow engine, composite services can be built involving both the workflow engine and the Composable Chat system. Unlike the chat systems described in the Diversity discussion, workflow engines very often explicitly provide facilities for interfacing to foreign systems, so it is likely that capable and exploitable service interfaces can easily be found. Additionally, because the pool of workflow engines is relatively small, investing in custom interfaces between these engines and the Composable Chat system should be feasible and should have high payoff.

## 4. Rich Services: A SOA Blueprint for Systems-of-Systems Integration

Building on the analysis of the strengths and applicability of SOA techniques for Composable Chat in Section 3, we now identify the process we followed to build a SOA-based integration architecture for our case study. Along the way, we introduce the architectural blueprint we use for large-scale systems-of-systems integration. This blueprint, called Rich Services [1], captures two important aspects of SOAs: (a) message-based decoupling of the services we want to compose, and (b) a mechanism for addressing crosscutting concerns such as security, policy/governance, and failure management. Furthermore, each Rich Service can be hierarchically decomposed.

To create the Rich Service-based architecture for the Composable Chat system, we employ a number of analysis techniques in combination with a Rich Services [1] architectural pattern:

- We model a conceptual chat domain that defines chat in the abstract, checking that the model supports the core capabilities (i.e., authentication, authorization, directory, presence, and Rich Messaging) sought in the Composable Chat system.

- Based on standards documents, we model the domains associated with each of the existing chat systems.

- We identify the services that define the interactions between the roles identified in each of the models, and then we identify the services needed to bridge the roles in the conceptual model to the roles identified in the existing chat systems.

- Finally, based on the hierarchy of roles and interactions elicited during this process, we create the Composable Chat system's Rich Services hierarchy.

The analysis process relies on pre-existing artifacts (such as standards documents) and artifacts generated along the way. In traditional development projects, artifacts such as requirements, use cases, and glossaries might be pre-existing or could be elicited as development progresses. In this case, the requirements are given, and the use cases and glossaries are inferred from the standards documents and domain models.

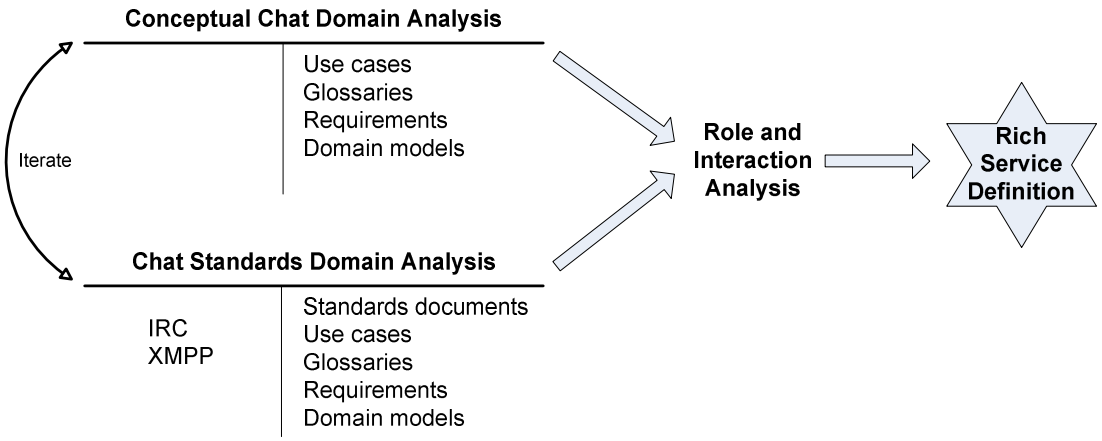


**Figure 1. Analysis Process Leading to Rich Service Definition**

Note that the conceptual model and chat system models are coevolved so as to account for all of the roles and interactions important in meeting the Composable Chat system requirements.

While this section presents the overall solution, subsequent sections describe the analysis processes that lead to these results.

## 4.1. Solution Summary

Our design for the Composable Chat system is a composite of existing chat systems, based on a Rich Services architectural pattern. This pattern leverages the composite pattern [9] to create systems-of-systems – in this case, a chat-system-of-chat-systems.

In this design, there are three layers: the enterprise integration layer, chat integration layer, and the chat system itself. The enterprise integration layer links to the chat integration layer, and the chat integration layer links to a chat system itself. There can be multiple chat systems, and therefore multiple chat integration layers, but only one enterprise integration layer. Each integration layer has significant value-added processing which implements not only the service interactions with connecting layers, but manages crosscutting concerns and provides additional services.

The combination of the composite pattern and the incorporation of value-added processing at each layer achieve key values of a SOA: loose coupling and high cohesion.

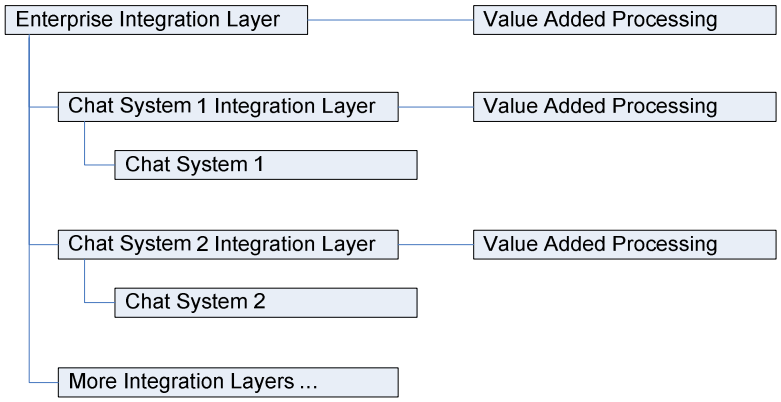


**Figure 2. Layered Design for System-of-Systems**

Additional benefits accrue as the result of the unique features of the Rich Services architecture, which is described in the next section. The details of the Rich Services solution are presented in Section 6.

## 4.2. Rich Services

Existing chat systems can be considered as COTS – Commercial Off The Shelf software, though they may not be strictly commercial or even off-the-shelf and may have considerable variability as compared to a reference standard. COTS integration [4] typically presents interoperability problems and interface mismatches, which should be addressed at the architectural level. An important aspect of COTS integration is use a loosely-coupled architecture that allows specific policies to be enforced by pluggable entities.

SOAs have emerged as a widely accepted solution to this challenge; they use standards-based infrastructure to forge large-scale systems out of loosely coupled, interoperable services. SOAs can create systems-of-systems by mapping existing systems into services, then orchestrating communication between the services. New functionality can be created by either adding new services or modifying communication among existing services. Because of these features, SOA projects are particularly amenable to agile development processes involving extremely short development and redevelopment cycles. Consequently, well-executed SOAs can drive down financial and technical risk by improving flexibility and reducing time to market.

As the number of stakeholders of a SOA project grows, though, so typically does the number and complexity of various business concerns (e.g., governance, security, and policy) and their level of distribution across the architecture. In order to maintain SOA's advantages, the integration of these concerns requires a framework that is scalable to complex systems and provides decoupling between the various concerns. The Rich Service architecture is a type of SOA that accomplishes this while providing direct and easy deployment mapping to runtime systems such as Enterprise Service Buses (ESBs), COTS platforms, Web Services, and combinations thereof.

A Rich Service architecture organizes systems-of-systems into a hierarchically decomposed structure that supports both "horizontal" and "vertical" service integration (see Figure 3).

Horizontal service integration refers to managing the interplay of application services and the corresponding crosscutting concerns at the same logical or deployment level. Vertical service integration refers to the hierarchical decomposition of one application service (and the crosscutting concerns pertaining to this service) into a set of sub-services such that their environment is shielded from the structural and behavioral complexity of the embedded sub-services and the form of their composition.

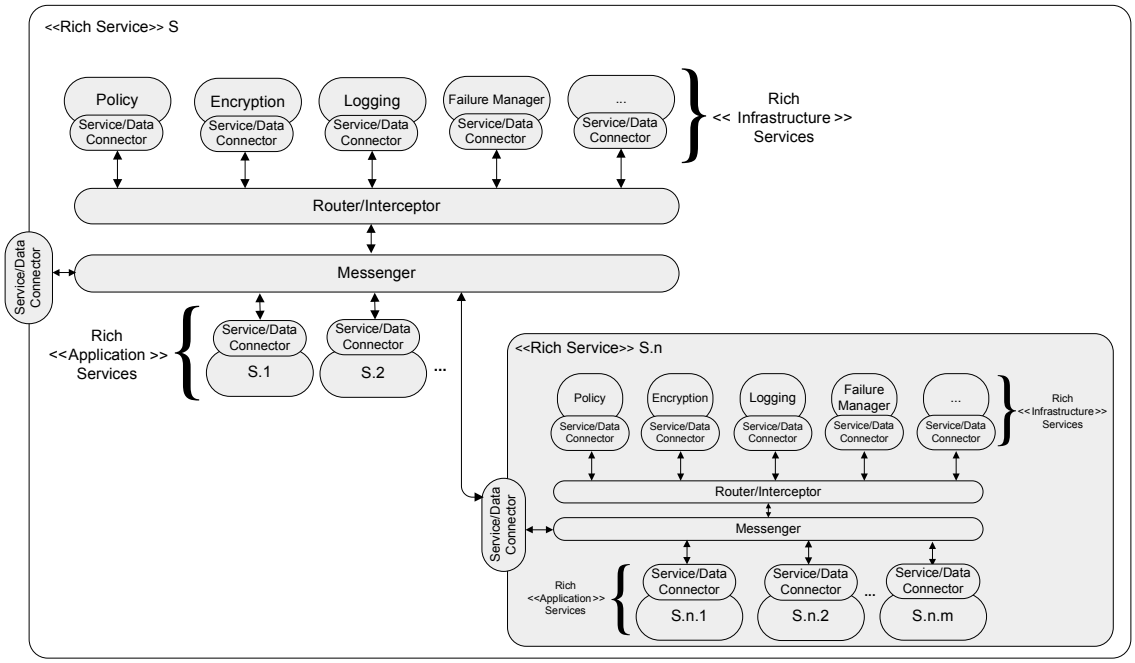


**Figure 3. Rich Services Architectural Pattern**

This architecture is inspired by ESB architecture/implementations, such as Mule [21] and Cape Clear. The main entities of the architecture are (a) the Service/Data Connector, which serves as the sole mechanism for interaction between the Rich Service and its environment, (b) the Messenger and the Router/Interceptor, which together form the communication infrastructure, and (c) the Rich Services connected to Messenger and Router/Interceptor, which encapsulate various application and infrastructure functionalities. In the following, we elaborate on each of these main entities and their role in the system.

To address the horizontal integration challenge, the logical architecture is organized around a message-based communication infrastructure having two main layers. The *Messenger* layer is responsible for message transmission between endpoints. By providing the means for asynchronous messaging, the Messenger supports decoupling of Rich Services. The second layer, the *Router/Interceptor*, is in charge of intercepting messages placed on the Messenger, then routing them. The routing policies of the communication infrastructure are the heart of the Router/Interceptor layer. Leveraging the interceptor pattern readily facilitates dynamic behavior injection based on the interactions among Rich Services. This is useful for the injection of policies governing the integration of a set of horizontally decomposed services.

The main entity of the architecture is the notion of Rich Service. A Rich Service could be a simple functionality block such as a COTS or Web service, or it could be hierarchically decomposed. We distinguish between *Rich Application Services* (RAS) and *Rich Infrastructure Services* (RIS). RASs interface directly with the Messenger, and provide core application

functionality. RISs interface directly with the Router/Interceptor, and provide infrastructure and crosscutting functionality.

Policy enforcement, failure management, and logging are typical examples of RIS, which are mainly intermediary services. The purpose of a logging service, for instance, is to ensure that all messages transmitted over the communication medium are recorded. A traditional service approach would require modifications to every service in the system in order to integrate the encryption mechanism, leading to scattered functionality. On the other hand, the Rich Service architecture introduces logging as an intermediary service that can inform the Router/Interceptor layer to modify the message routing tables to ensure that every message sent to the external network must first be processed by the logging service. Only the communication infrastructure needs to be aware of the encryption service.

A Service/Data Connector is the means by which Rich Services are connected to the communication infrastructure. It encapsulates and hides the internal structure of the connected Rich Service, and exports only the description and interfaces that the connected Rich Service intends to provide and make visible externally. The communication infrastructure is only aware of the Service/Data Connector, and does not need to know any other information about the internal structure of the Rich Service. This helps tackle the vertical integration challenge introduced by systems-of-systems.

In this architecture, each Rich Service can be decomposed into further Rich Services, with the Service/Data Connector acting as a gateway responsible for routing messages between layers. Additionally, the use of the Router/Interceptor layer removes dependencies between services and their relative locations in the logical hierarchy. This approach enables services from different levels of a hierarchy, possibly with different properties (such as logging and auditing requirements) to interact with each other seamlessly without ever being aware of such incompatibilities.

In addition, this architecture can support the specification of dynamic systems, where new services can be introduced at runtime and provide their functionalities without the need to change or adapt the implementation of existing services. Any required adaptation or rerouting can be handled by adding application or infrastructure services to the system.

Service composition can be addressed in the system by following two general approaches. One is to add a new RAS, acting as an orchestrator, which utilizes the existing services to provide the composite service. Another approach is to add a RIS that intercepts messages coming from or going to a specific RAS.

Another important benefit of the Rich Services architecture is the ability to analyze, design and decompose the system according to various major concerns (e.g., security, resource constraints, policies, logging, and governance) in isolation. Each major concern can be designed separately, and then made available to the entire Rich Service via the Router/Interceptor layer. This separate design of common concerns results in reduced implicit coupling among the Rich Service's constituent services. This feature greatly contributes to the design of Large Scale systems where multiple authorities are involved, as would likely be the case for a Composable Chat system. As an example, from the security viewpoint, multiple security-related services can be designed and connected to the system at different levels in the Rich Service hierarchy to address the authentication, access control permissions, and privileges across multiple authority domains.

## 5. Methodology

Creating an instance of the Composable Chat architecture is a multi-step process involving:

- modeling a conceptual chat system,
- modeling individual chat systems,
- defining the service interactions between the roles,
- composing the hierarchy and interactions for actual chat systems under Rich Services.

This section describes each of these steps, leading to an example of a particular Rich Service-based architecture for our Composable Chat case study.

### 5.1. UML Modeling Conventions

A number of modeling languages can be used to describe a system, and each has strengths and weaknesses. In this paper, we model chat systems using two languages: the Unified Modeling Language (UML [14]) complimented by textual descriptions. While UML is a common modeling language, it is still under development and is not always adequate to describe the relationships we model. Therefore, we use textual descriptions to supplement UML, and we also introduce a modest set of UML variations.

Note that we add variations to the UML only reluctantly, and motivated by great gains in clarity and brevity.

**Stereotypes**

In a standard UML class diagram, a stereotype can be used to indicate inheritance from an interface, use of an enumeration as a data type, or to denote a profile-based UML extension.

In our models, we extend stereotype usage to indicate the incorporation of a model-defined behavior. For example, the core domain diagram in Section 5.3 uses the `<<resource>>` stereotype to incorporate the structural and behavioral relationships defined for the `resource` entity in the `resource` class diagram. Thus, a class containing the `<<resource>>` stereotype is said to behave as a `resource` in addition to having the relationships otherwise modeled for it.

**Patterns**

A standard UML class diagram models relationships between entities by representing entities as class boxes and relationships as lines connecting them. In doing so, by looking at a pattern of relationships, it is possible to infer when a design pattern [9] is being used and what the design pattern may be, though the inference may be difficult to make.

Because the identification of a pattern yields a large amount of behavioral information, highlighting pattern usage is important in conveying the meaning of a model. To better expose this information, we make the use of a pattern explicit through a combination of comment boxes and role labeling. When a relationship implements a pattern, we indicate that the pattern is in use by:

- documenting the pattern usage in a comment box and drawing lines to the participating class boxes,
- using the pattern role names as labels for the endpoints of the relationship connectors involved in the model.

An example of this is found in use of the Observer pattern in Section 5.3 in the presence system model.

In the interest of brevity, as we present our models in this and the subsequent sections, we focus on *highlights* of the respective conceptualization, rather than discussing each modeling element in detail.

## 5.2. Pattern Descriptions

Over the last several years, many people in the software engineering discipline have recognized the need for a standard way of describing recurring program and architecture design patterns. As a result, several compendiums have been created, some with precise and elaborate pattern descriptions that have become the de-facto standard for describing pattern functionality, usage, and implementation. Additionally, referencing these patterns in discussions and in writing helps to quickly convey an accurate view of a design and its parameters.

When discussing design issues in this paper, we frequently reference some of the patterns named and described in [9], which has become a de-facto industry standard reference work for design patterns. It is possible to find enlightening discussions of these patterns easily on the Wikipedia Internet site. For convenience, we list the patterns we use in this paper and provide links to appropriate Wikipedia articles:

| **Pattern** | **Wikipedia Article** |
|---|---|
| Observer | http://en.wikipedia.org/wiki/Observer_pattern |
| Composite | http://en.wikipedia.org/wiki/Composite_pattern |
| State | http://en.wikipedia.org/wiki/State_pattern |
| Command | http://en.wikipedia.org/wiki/Command_pattern |
| Proxy | http://en.wikipedia.org/wiki/Proxy_pattern |
| Strategy | http://en.wikipedia.org/wiki/Strategy_pattern |

Note that the Repository pattern is not described in either [9] or in Wikipedia. Instead, it is described in [22].

## 5.3. Modeling a Conceptual Chat System

The objective of creating a conceptual chat system is to understand the essence of all chat systems by modeling chat in the abstract. This decouples the conceptual ideas behind chat from specific deployment choices and constraints found in the concrete chat protocols under investigation; it also allows us to conceptualize the additional features sought for the Composable Chat independent of their presence or absence in the concrete protocols. To accomplish this, we model the essential meaning and operation of a *conversation* independent of server mechanics, specific data structures, and specific deployment details. In doing so, we follow a multi-step procedure:

- Create a high level model and iterate it a number of times to reduce the model to its barest, most general form (in line with the processes described in [22]).
- Identify and develop a discipline of authorization checks in appropriate situations.
- Develop detailed models exploring each of the major entities and their relationships to other entities and sub-entities.
- Develop a list of use cases complimentary to the model, and including an authorization capability model.
- Validate the model by comparing its use cases against the use cases produced for the actual chat systems.

Our conceptual core domain model appears as follows:

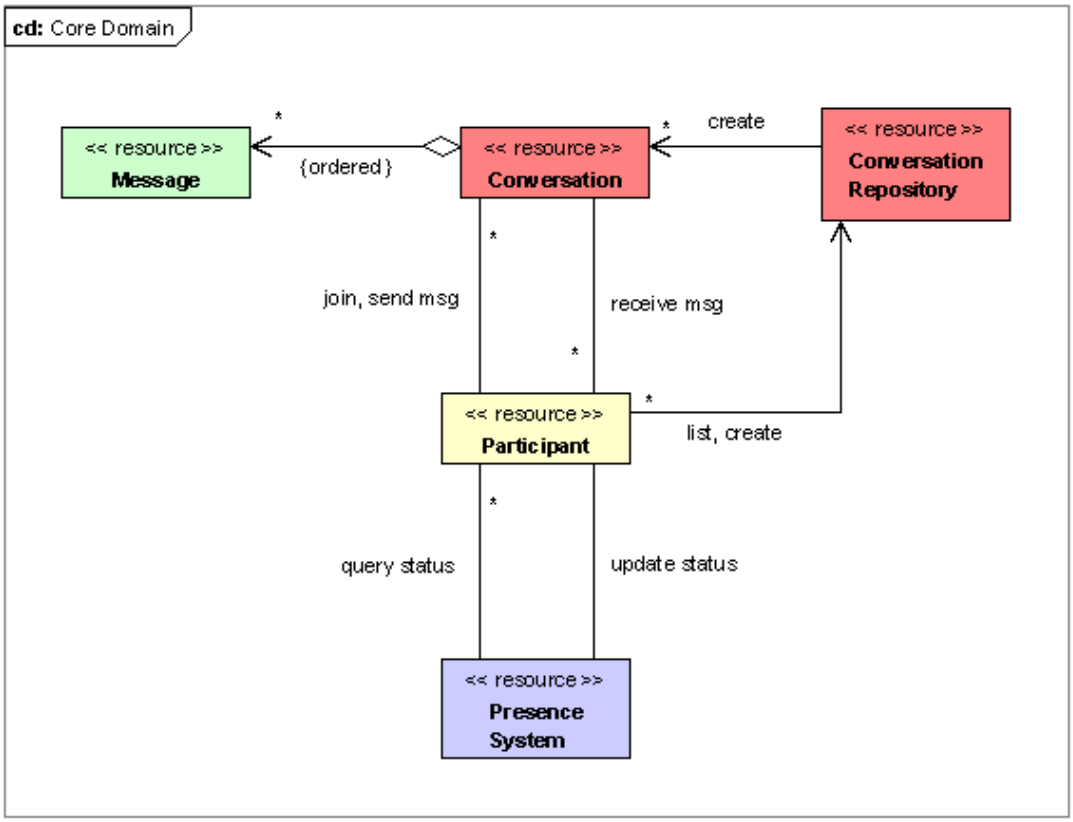


**Figure 4. Conceptual Chat Core Domain Model**

The model represents the abstraction of a conversation independent of any chat system – it could

just as easily apply to conversations between people in physical or virtual contact. The highlights are:

- A conversation is an ordered sequence of messages.
- A conversation repository lists all existing conversations and can create facilities for new conversations, too.
- Participants can create and join a conversation, and can send and receive messages.
- A presence system lets participants know of the status of other participants.

Additionally, in order to have a secure system, the participants must be authenticated, and they must be authorized for any action they would like to perform. An entity on which a participant can perform an action is called a *resource*. Participants performing actions on a resource must have the privilege (called *capability*) to do so – participants are granted lists of capabilities as a result of being authenticated. For example, a conversation may require that a participant have a particular capability before it is allowed to send a message. In this case, the conversation plays the role of resource, and sending-a-message plays the role of action.

In the model, resources requiring authorization prerequisite to actions are marked using the `<<resource>>` stereotype. In fact, we use diagrams such as Figure 5 to *define* the relationships of a concept such as `<<resource>>`, with the understanding that whenever the stereotype is applied in another diagram, the entity to which it is applied incorporates these same relationships.

The entities involved in a resource authorization are also modeled:

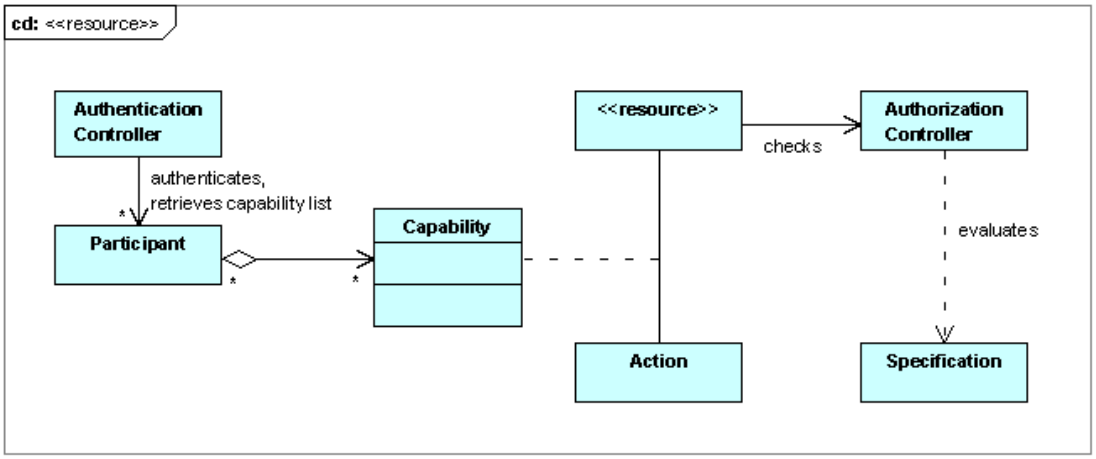


**Figure 5. Conceptual Chat Resource Authorization Model**

Essentially, for a given resource, the question is whether a participant with certain capabilities can execute an action. To answer the question, the resource contacts the authorization controller, which executes an algorithm (which may include checking a policy database), modeled here as a Specification.

While the core domain model relates the major entities, each of the entities requires further elaboration. Should an attempt be made to consolidate all models into a single comprehensive diagram, the diagram would be too complex to readily communicate its contents. Instead, sub-models are created to elaborate each major entity. Additionally, entities within a family of entities

are color coded to ease understanding.

The conversation module models a conversation entity and its relationship to participants, the conversation repository, the presence system, and messages:

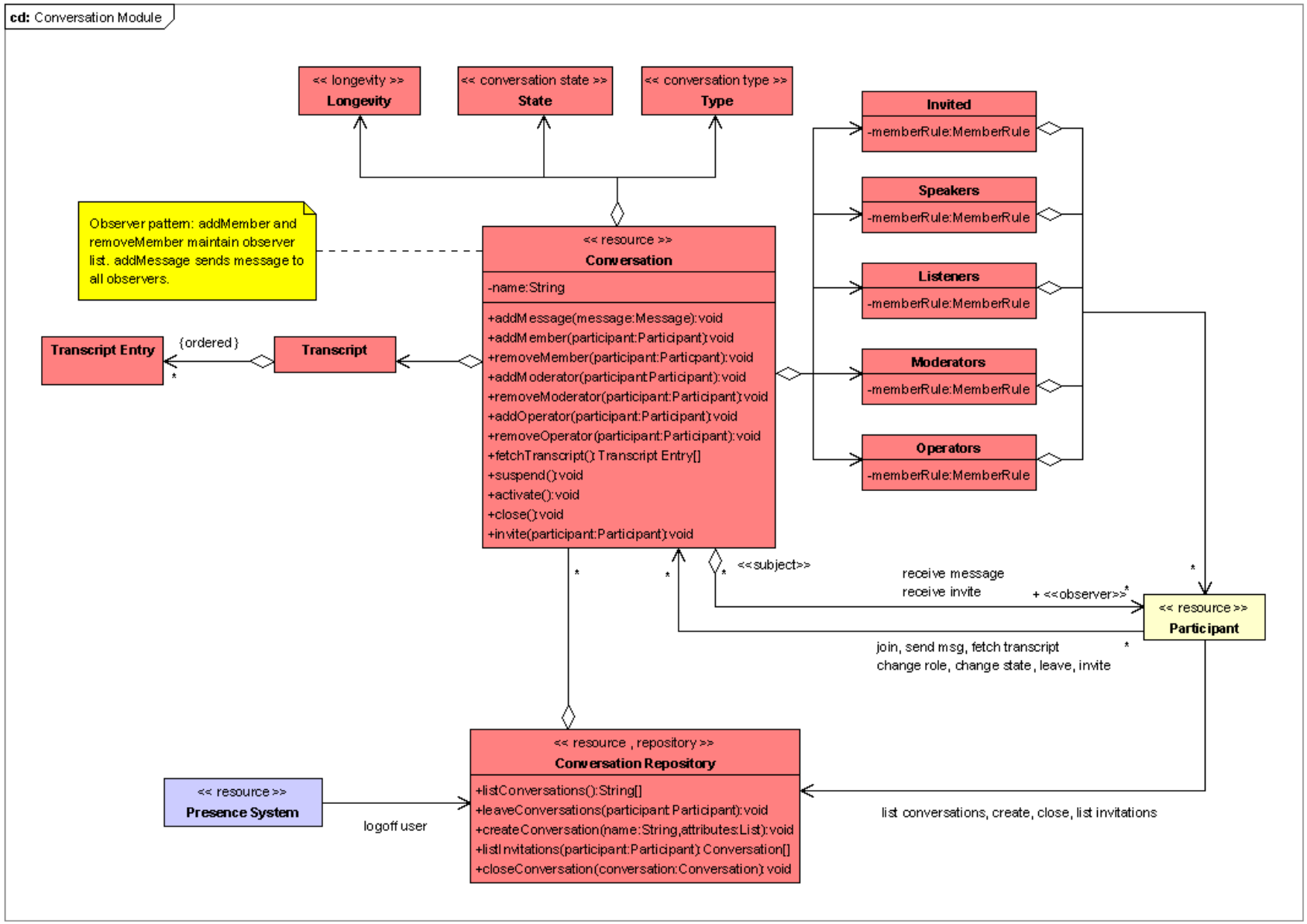


**Figure 6. Conceptual Chat Conversation Model**

The highlights of the conversation model are:

- The conversation represents an interaction between participants. Depending on the Type of the conversation, the interaction can be 1-1 (IM), many-to-many (chat), and other variations.
- The conversation has attributes such as Longevity and State, which indicate how stages of the lifetime of the conversation are managed.
- The conversation repository tracks all existing conversations, including those that might be suspended.
- A transcript is associated with a conversation, and the transcript records all activity in the conversation.
- The presence system interacts with the conversation repository, which returns a list of conversations in which a participant appears. Thus, the presence system can remove the participant from each such conversation when the participant is no longer present.
- A conversation is associated with participants acting in a number of roles. Besides speaking (sending) and listening (receiving), a participant can be a moderator, an invitee, or an operator. Policies govern how many of each kind of role a conversation can support.
- The conversation repository implements the Repository pattern, which means that it can store an existing conversation and all of its attributes for later retrieval. It can also identify and return a reference to an active conversation.
- The conversation implements the Observer pattern, which means that a message sent to the conversation is forwarded to all of its listeners. A participant is added to this list by joining the conversation.
- Authorization checks are performed for any action requested of either a conversation or the conversation repository.

The transcript entry module elaborates on the transcript in a conversation:

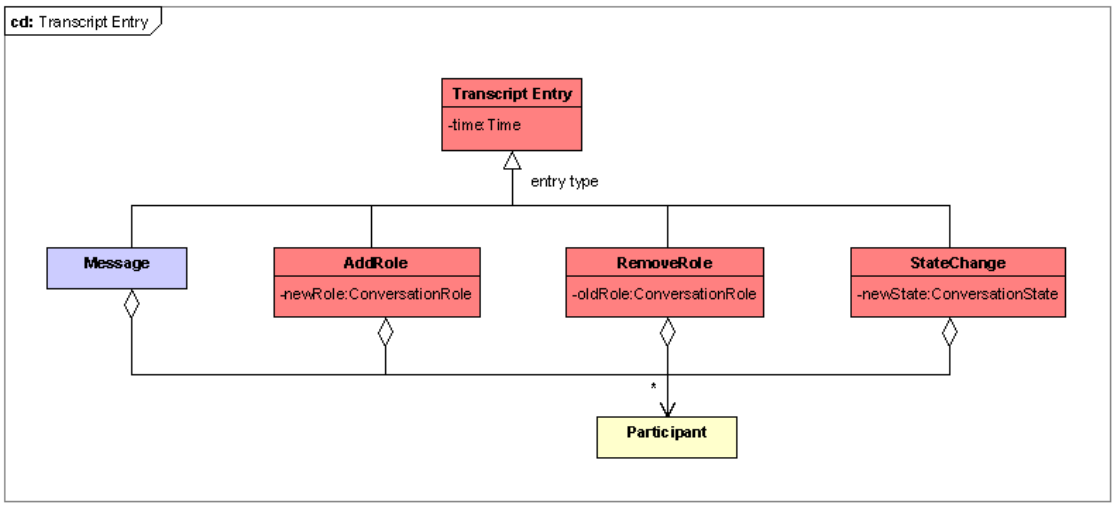


**Figure 7. Conceptual Chat Transcript Entry Model**

The highlights of the transcript entry model are:

- A transcript entry represents an event in a conversation, and there are a number of kinds of events.
- One kind of event is a message, and the sender and receiver(s) of the message are recorded.
- Other kinds of events involve changes in administrative roles and changes in the state of a conversation, and the originator of the event is recorded.
- Other kinds of events are possible.

The message module elaborates on a conversation message:

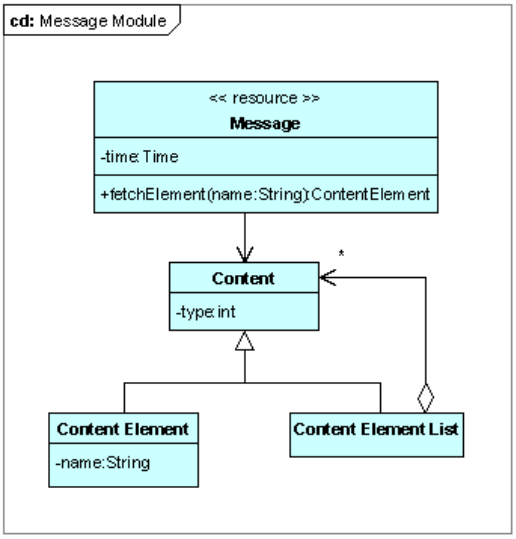


**Figure 8. Conceptual Chat Message Model**

The highlights of the message model are:

- A message can contain a single content element (e.g., a text string) or it can contain a structured list of content elements.
- The content structure implements the Composite pattern, which allows for a content to contain embedded content (e.g., an entire battlefield scenario, including text, pictures, asset lists, resource links, and so on. An element such as an asset list could itself be a composite containing asset identifiers, tasking descriptions and priorities, pictures, audio, video, and other information.).
- Authorization checks are performed before any type of content can be added or fetched, and the authorization can be based on the type of the content.

The participant module models a participant entity and the participant repository, including their relationships with a conversation, the conversation repository, and the presence system:

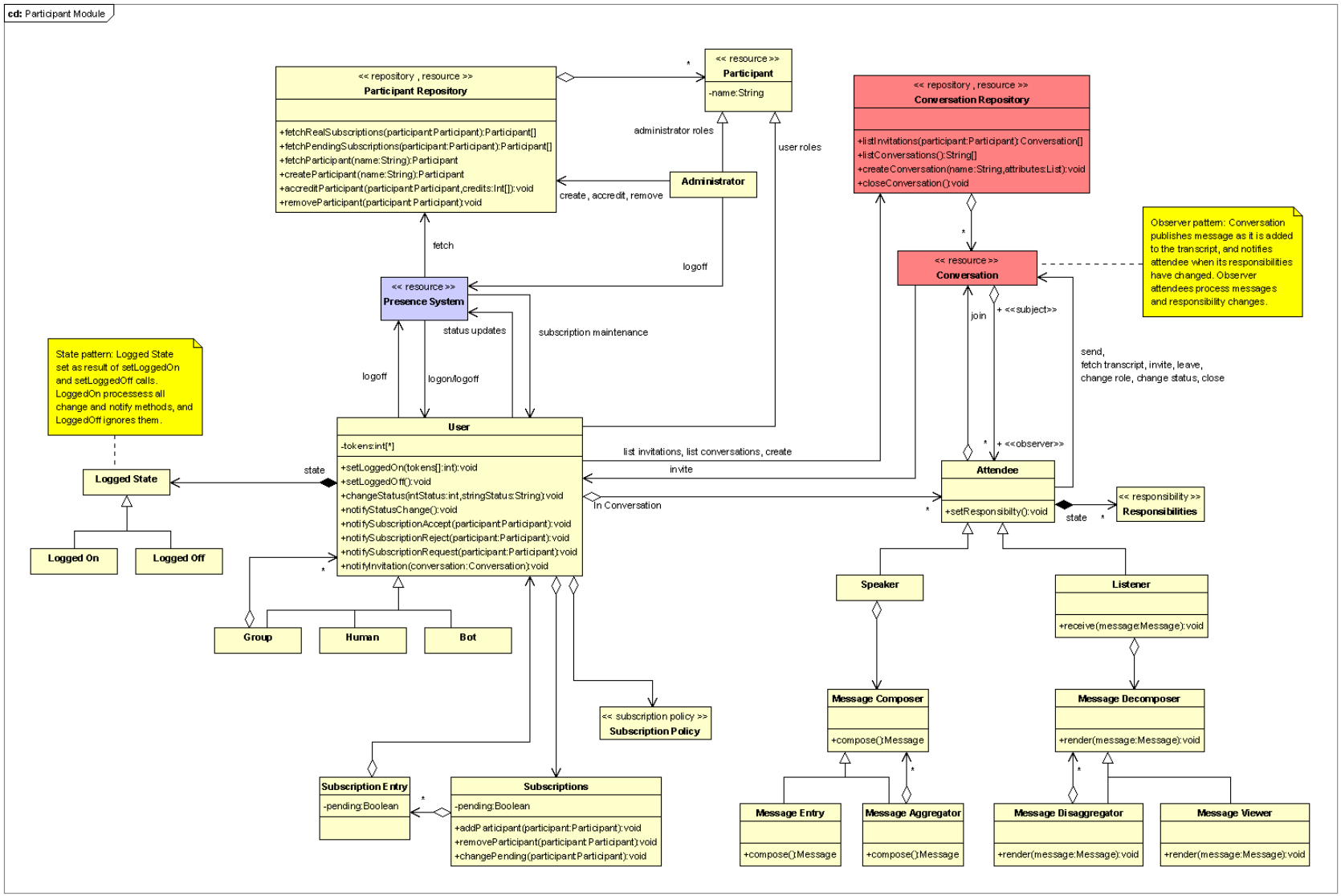


**Figure 9. Conceptual Chat Participant Model**

The highlights of the participant model are:

- A participant can have roles of both regular user and administrator.
- A participant repository stores and retrieves all participant information, whether or not the participant is active in the chat system.
- The administrator role maintains the participant repository, and can add and remove participants. It can also add and remove capabilities assigned to a participant.
- A user is either logged onto or off of the chat system, and this status is influenced by the presence system.
- A user can be a human user, a bot, or a member of a group of either human users or bots.
- A user can have a subscription list, which is a list of other participants whose presence the user is interested in. Additionally, a user can have its own policy as to whether it automatically allows itself to be added to another user's subscription list.
- A user can be an attendee (member) of one or more conversations (IM, chat, etc). As such, it can be a speaker, a listener, or both, and can have other roles in addition.
- As either a speaker or a listener, an attendee can compose or view composite messages.
- Users can use the conversation repository to list conversations they are authorized to list. Likewise, they can join conversations they authorized to join.
- The logon state implements the State pattern, and the attendee implements the listener role in the conversation's Observer pattern.

The presence system module models the presence system and its interactions with the participant and conversation models:

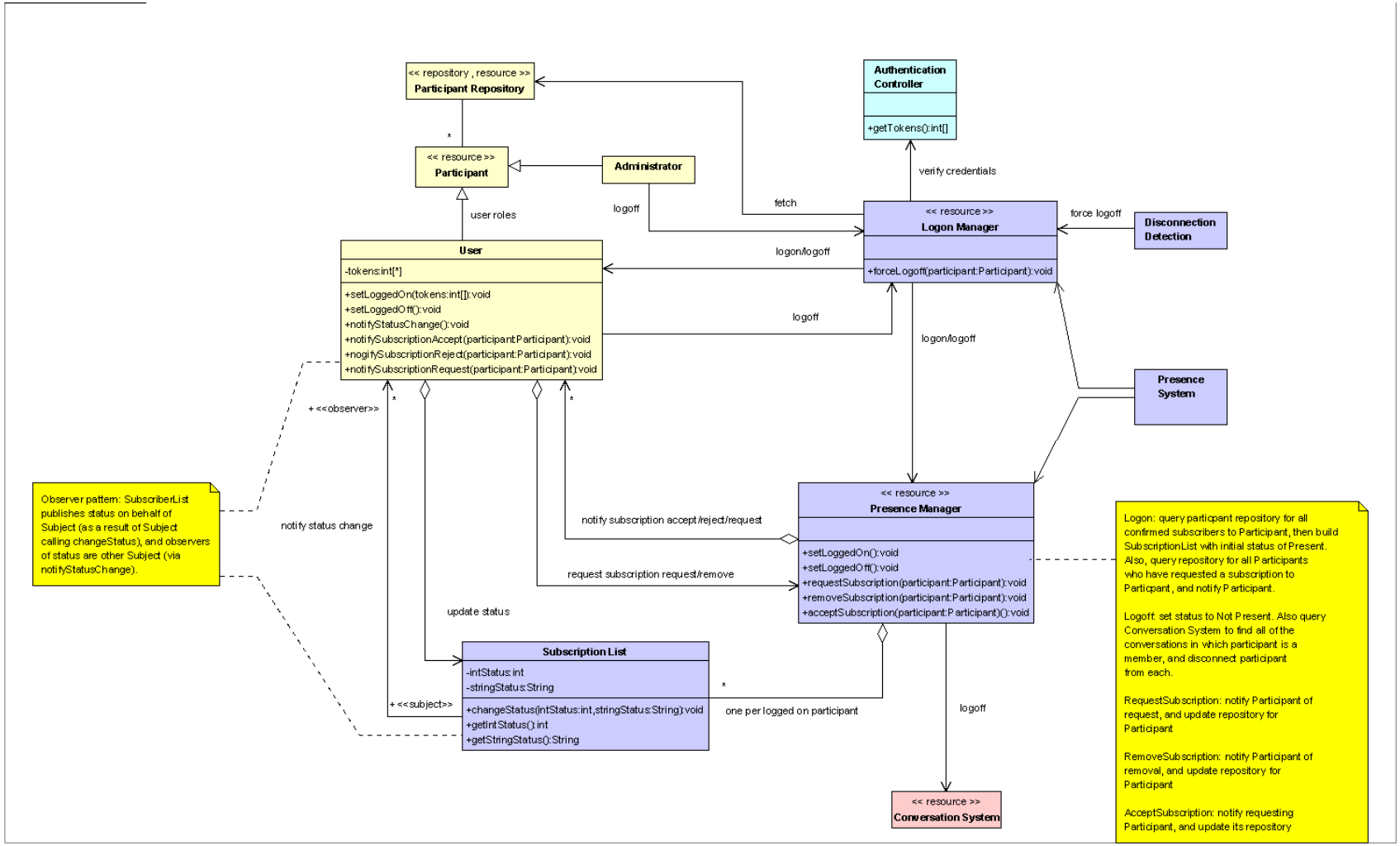


**Figure 10. Conceptual Chat Presence System Model**

The highlights of the presence system model are:

- The presence system consists of the logon manager and the presence manager. The logon manager manages the participant's logon state, and the presence manager manages subscribed presence status interactions.
- The logon manager fetches a newly logged on participant from the participant repository, and it cooperates with the authentication system to acquire a capability token list for the user.
- The logon manager also logs a user off in response to a user or administrator request, or when it senses the user has disconnected from the system.
- The presence manager creates a user's subscription list when the user logs on. When the user logs off, it interacts with the conversation system to remove the user from all conversations.
- The subscription list implements the Observer pattern by distributing a user's presence status to all users subscribed to it. Users play the role of both changing the status and receiving status changes.

The command view module models the commands available in the chat system, including which entities they affect and the state information on which they rely:

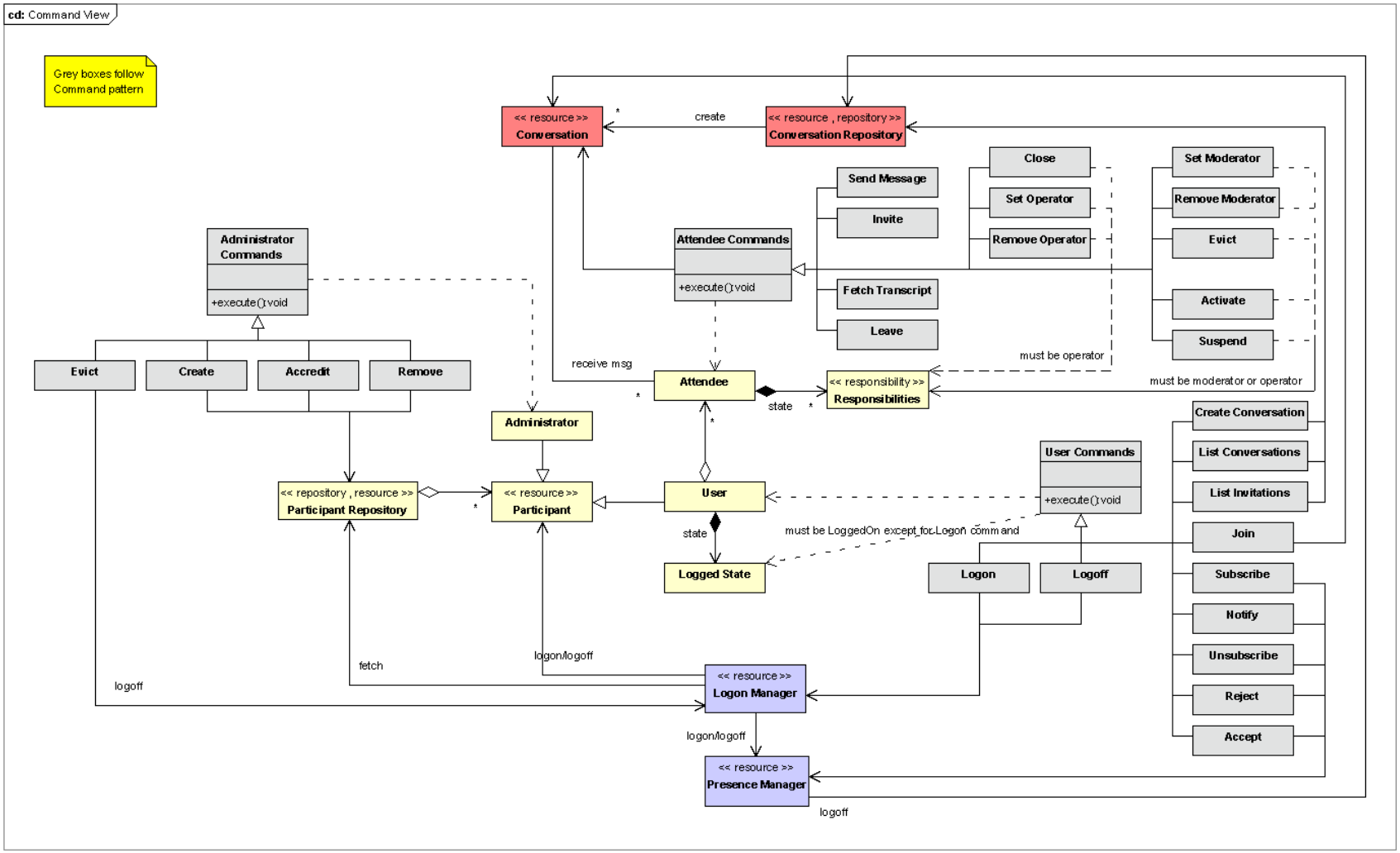


**Figure 11. Conceptual Chat Command View Model**

The highlights of the command view model are:

- There are a number of command groups: administrator commands, attendee commands, and user commands.
- Within each command group is a number of commands, each of which implement the Command pattern. Each command is related to a role and possibly a state variable on which the command's applicability lies.
- Attendee commands interact with a conversation. Amongst the attendee commands, a subgroup relies on the attendee also serving the operator role. Likewise, another subgroup relies on the attendee serving either the operator or moderator role.
- Most user commands rely on the user being logged on. A subgroup of user commands interact with the conversation repository, and another subgroup interacts with a conversation itself. Still another subgroup interacts with the presence system.
- Most administrator commands affect the participant repository.

The enums module models the various enumerations used as types within the other models. Note that the entities are color coded to indicate the models to which they apply:

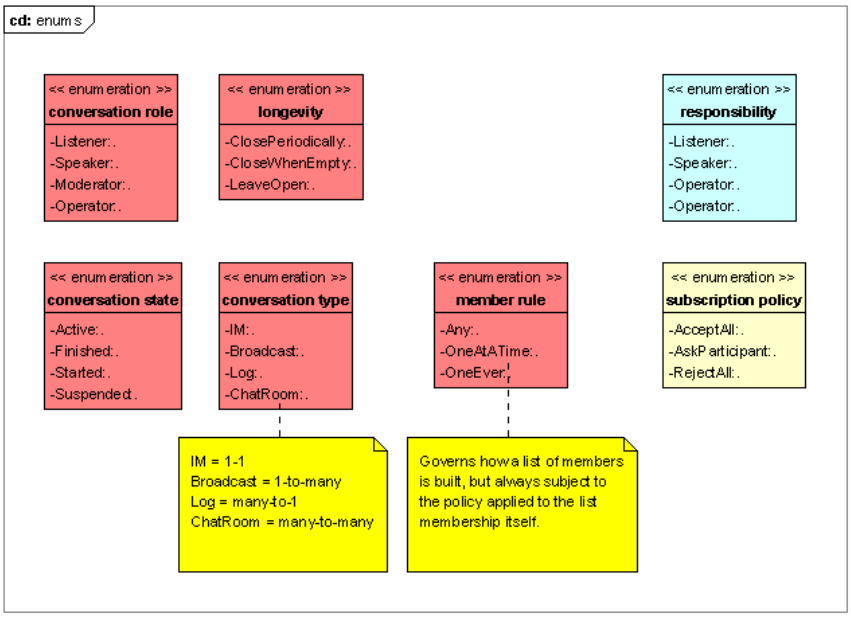


**Figure 12. Conceptual Chat Enumerations**

Elaborating the core domain as a full domain model confirms the primacy of the core domain model – the relationships between the major entities and the sub-entities remained as natural and sensible as the core domain model.

The conceptual model is validated by creating a list of use cases and comparing them for coverage in the existing chat model use case lists. Additionally, we augment the conceptual model use case list with the information required to make authorization decisions, including the participant/role, the resource, and action, and the capabilities required by the participant in order to execute the action:

| ID | Use Case | Resource | Participant | Role | Action | Capability |
|---|---|---|---|---|---|---|
| UC1 | A participant wants to add a participant to the system | Participant Repository | Participant ppp | Administrator | Create | EC: Can participant ppp add participants to the system |
| UC2 | A participant ppp wants to respond to a request for presence subscription from participant ppp0 | 1. Presence Manager<br><br>2. Participant ppp0 | 1. Participant ppp<br><br>2. Presence System (on behalf of ppp) | User | Notify | 1. EC: Can participant ppp accept/reject a subscription request from participant ppp0<br><br>2. I: Can receive subscription accept/reject from participant ppp |
| … | … | … | … | … | … | … |

For some use cases, multiple authorizations are required (as in UC2).

To align with the three tier integration architecture presented in Section 6, we model capabilities in three tiers: definable at the enterprise level (E), definable at the military command (or, more generally, the sub-system) level (C), and definable by individual users (I). The capability for the first authorization step in UC2 is read as "either an enterprise-defined policy or a command-defined policy can enable a user to accept or reject a presence subscription request from a particular other user". The capability for the second authorization step is read as "a user-defined policy can enable reception of a subscription request from a particular other user". (This capability amounts to a model of a subscription block.)

Given our conceptual model use case list, our cross-check with the use cases for the existing chat systems, and the capability elaboration, we were satisfied that the conceptual model covers the main cases for the actual chat systems, and that the security policy model was tractable.

Note that rigorous validation of the conceptual model is not called for in [22], nor is it practical in the time available for a real-world modeling and implementation project. Instead, it is expected that further iterations will take place over time as deeper understanding of the model is called for or as system requirements change.

## 5.4. Modeling Individual Chat Systems

The objective of modeling an existing chat system is to understand the chat system domain and to

gain perspective for creating the conceptual chat system. Modeling a chat system involves a number of steps:

- Determine basic relationships
- Determine a domain vocabulary
- Create use cases
- Determine the important entities and roles
- Create a model using the UML

An initial survey of a chat system involves acquiring an authoritative source for the chat system definition, then making an initial pass through it to understand the relevant entities and their relationships. For IRC, we accomplished this using the RFCs for IRC and creating a mind map [23]:

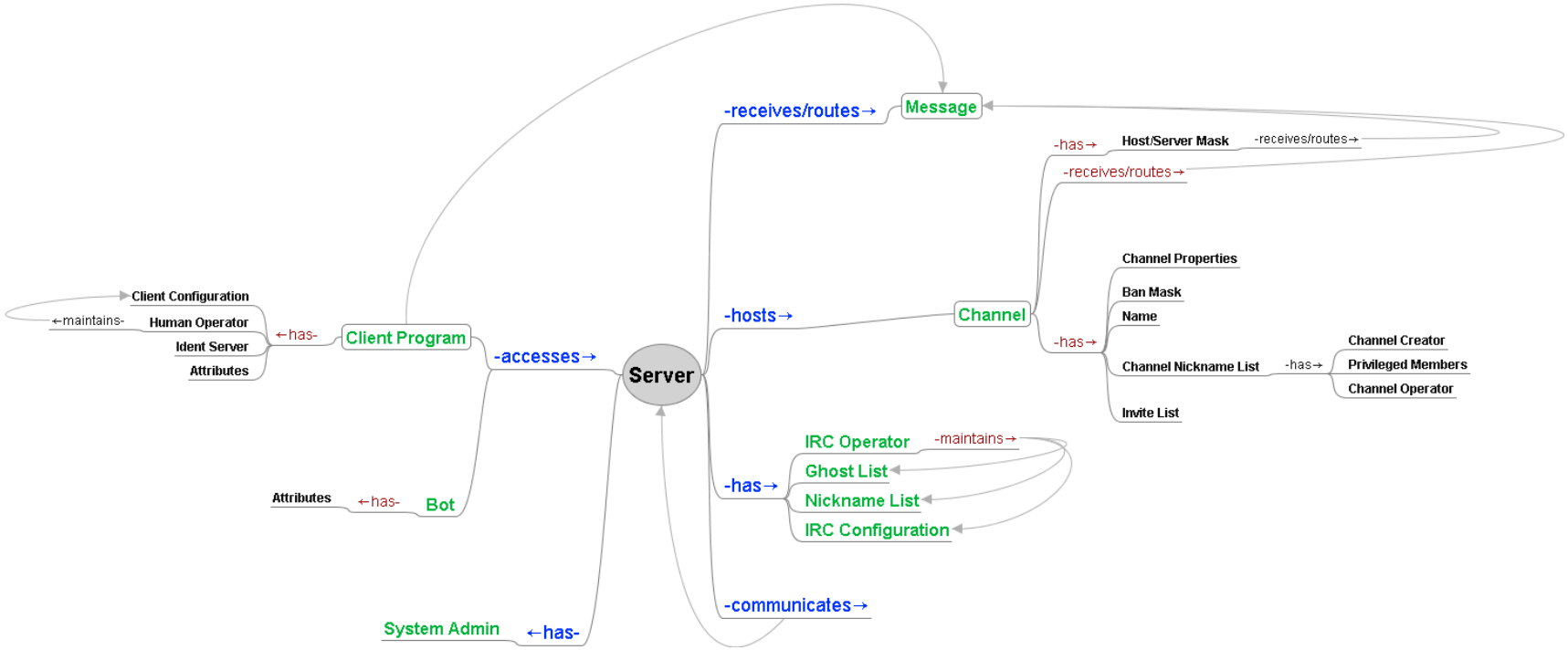


**Figure 13. IRC Mind Map**

Next, we again use the authoritative documents to create a formalized domain vocabulary. For XMPP, we also used RFCs, and created a glossary containing the authoritative document and section for each term, the term itself, and the definition:

| Source | Term | Meaning |
|---|---|---|
| RFC3920-3.1 | Address | A string that uniquely identifies a network endpoint such as a resource, a client, or a service. It has the format [node "@"] domain ["/" resource]. |
| RFC2779-1 | Administrator | A principal with authority over a local computer and network resources. |
| … | … | … |

Next, we again use the authoritative documents as an aid in creating a first cut of a comprehensive list of use cases. For XMPP, we used the RFCs to create a use case list that uniquely identified each use case, the authoritative document and section(s) for the case, and the use case itself:

| ID | Source | Use Case |
|---|---|---|
| UC1 | RFC3920-5.1 | A client attempts to establish a session with an XMPP server, but is rejected due to insufficient credentials. |
| UC2 | RFC3920-5.1<br>RFC3921-5.1.1 | A client attempts to establish a session with an XMPP server, and is accepted. The client is established as "present". |
| … | … | … |

(For IRC, this process yielded 39 terms and 56 use cases. For the XMPP core, 35 terms and 31 use cases were produced. For SIP/SIMPLE, the RFCs we used contained explicit glossary sections and numerous use cases, so it was unnecessary for us to generate our own.)

Next, we use the vocabulary and use cases to identify the central entities and roles, then attempt to model them using the UML. Once the model is complete, we validate the modeled relationships by checking that each of the vocabulary terms and use cases are properly represented and related in the model. Modeling in this way is an iterative process, and converging on a model requires multiple passes through the vocabulary and use cases. For the sake of closure, we defer modeling unlikely and highly idiosyncratic terms, cases, and relationships.

The resulting models are shown in Section 5.5 – they represent deployment models for the actual chat systems.

## 5.5. Detailed Look at Chat Systems and Domains

This section describes various chat systems using a combination of textual description and the UML 2.0 notation as described in [14]. Note that these systems are described in terms of their formal specifications, which are inherently deployment-oriented. A conceptual treatment of chat systems is provided in Section 5.3.

**IRC**

The IRC (Internet Relay Chat) system is described in a collection of RFCs [10][11][12][13], though the RFCs serve as a basis for departure rather than as a rigorous standard. Therefore,

while a number of IRC systems exist, they incorporate various extensions and implementation flavors that make each unique. Consequently, "IRC" connotes a family of chat systems rather than a chat system standard.

An IRC system is based on a handful of principles:

- A user is identified by a user-provided nickname, and is associated with the server he or she connects to.
- A user can subscribe to a chat room, and messages sent to a chat room are sent to all subscribers to that chat room.
- IRC servers communicate amongst themselves in order to synchronize their user lists, presence status, chat room lists, and all other chat system information.

The result of the IRC server communication is a high availability system that can deliver chat services even when a network partitions, and can regain consistency when a network heals.

A high level, client-centric view of IRC is:

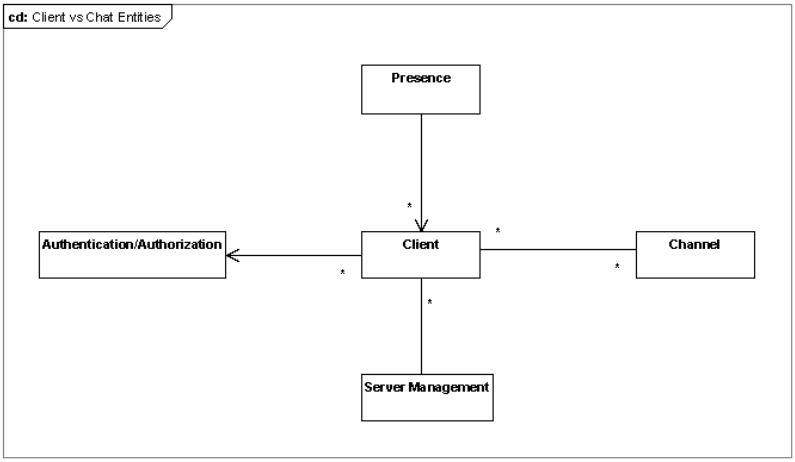


**Figure 14. High Level, Client-centric IRC Model**

This view shows how the major logical entities in the IRC domain relate to a client (e.g., a user). The highlights include:

- The presence system tracks many clients.
- A client can be a member of many channels (e.g., chat rooms), and a channel can service many clients.
- A server can have many clients.
- The authentication system (if there is one) can service many users.

A message-centric view focuses on a number of important relationships:

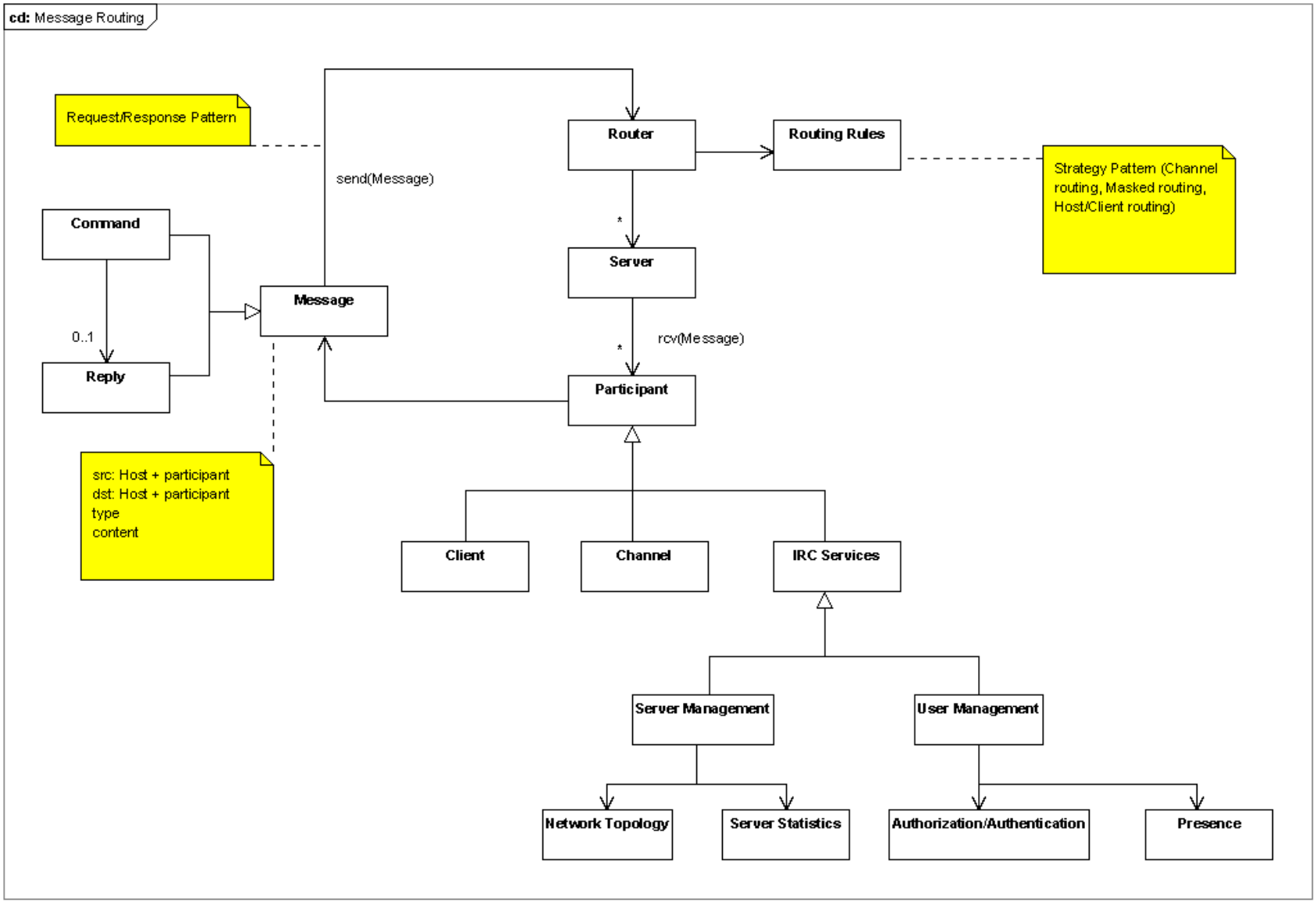


**Figure 15. Message-centric IRC Model**

This view shows how messages are routed through the system, and identifies important behavioral patterns relating to logical entities. The highlights include:

- A message may be a command or a reply, and if it is a reply, it is a reply to a command.
- A participant may be a client, a channel, or IRC services, and it can send and receive messages.
- Messages go to a router entity, which uses routing rules to send the message to another server or to a participant in the current server. Note that a router is a logical entity that is built into the IRC server code, though this type of diagram would not show that.

Note that an IRC system uses messages to convey chat room conversations between users, and it also uses messages to convey information between users and servers, and between servers and servers. So, messages are the basis for chat, for chat and server management, and for server synchronization.

**XMPP**

The XMPP system is an evolution of the open source Jabber system maintained by the community at www.xmpp.org. It is described in a collection of RFCs [15][16] and extensions [17][18].

An XMPP system is fundamentally a network of servers linked by channels that bear a stream of XML [19] documents called stanzas, and which bear XMPP messages.

The XMPP specification defines concerns common to all XMPP applications and extensions, including:

- negotiating a secure channel between a client and a server or between two servers
- exchanging messages between users
- managing subscriptions for presence information
- exchanging presence information

While XMPP defines stanzas for each of these activities, it also sets basic ground rules for using XML to create new kinds of stanzas for implementing additional services. The http://www.xmpp.org/extensions/ web site contains a repository for extension specifications and advisories, currently numbering 143. Consequently, “XMPP” connotes a family of chat systems which share a common messaging infrastructure and conventions. This paper describes an XMPP system having the XMPP Service Discovery extension [17] and the Multi-User Chat extension [18].

A simplified view focuses on the relationship between major XMPP logical entities (grouped by color):

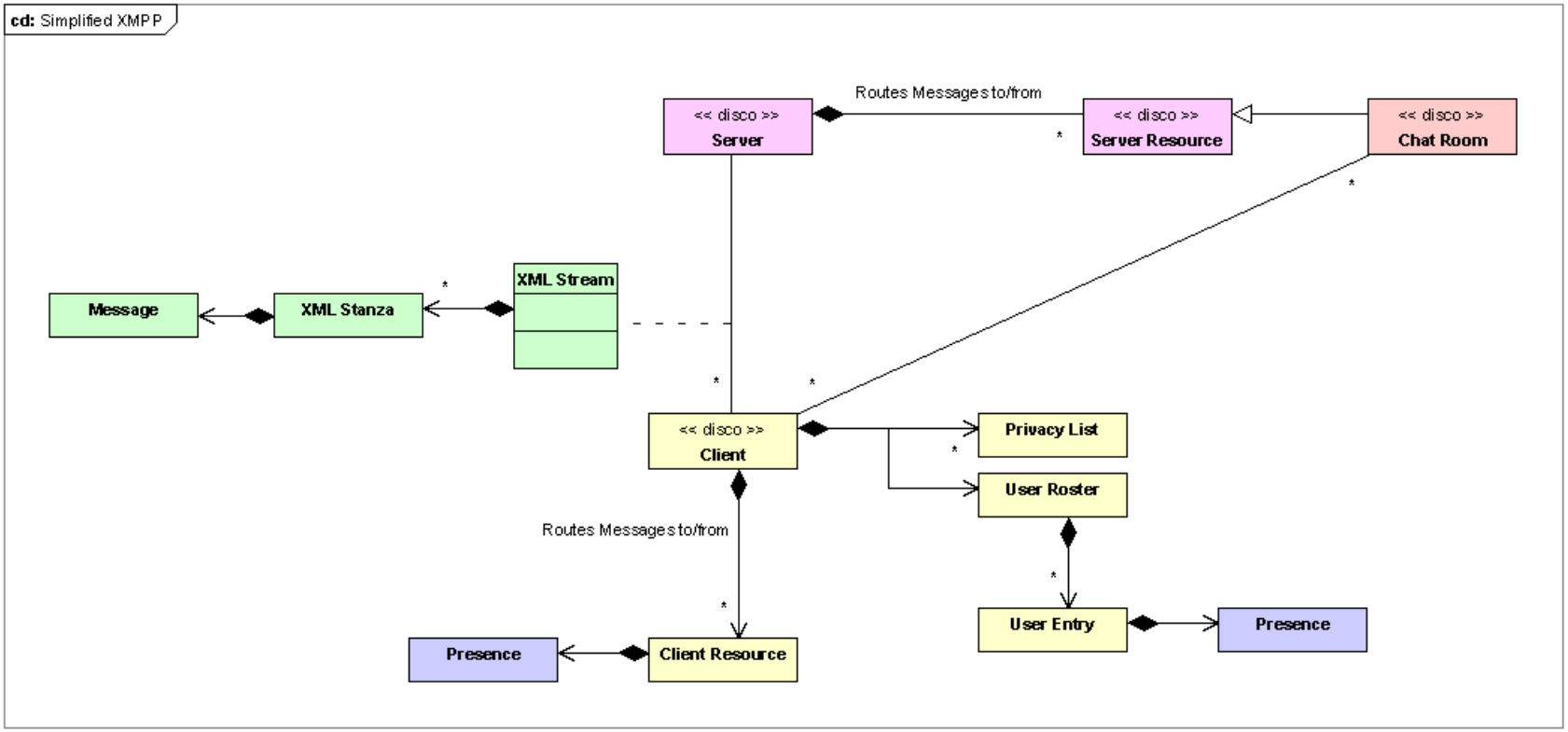


**Figure 16. Simplified XMPP Model**

This view shows how messages are routed through the system. The highlights include:

- A client and a server have a number of resource entities. An important client resource is the client itself, and an important server resource is a chat room.
- Messages are routed between resources via clients and servers.
- A client is associated with a single server, and a server can have multiple clients.
- The conversation between a client and a server is an XML stream consisting of XML stanzas, which contain messages.
- A client can be a member of several chat rooms, and a chat room can have several clients.
- A client has a number of lists, including a user roster, which identifies subscriptions to other clients' presence information. Each client has a presence status, too.
- Resources are discoverable using the XMPP Service Discovery extension, so it is possible to discover which servers, clients, chat rooms, and chat room clients the XMPP system contains. (The `<<disco>>` stereotype indicates which logical entities participate in the discovery protocol.)

A more complex view focuses on the numerous details of server relationships, client roles, the presence system, and chat room management:

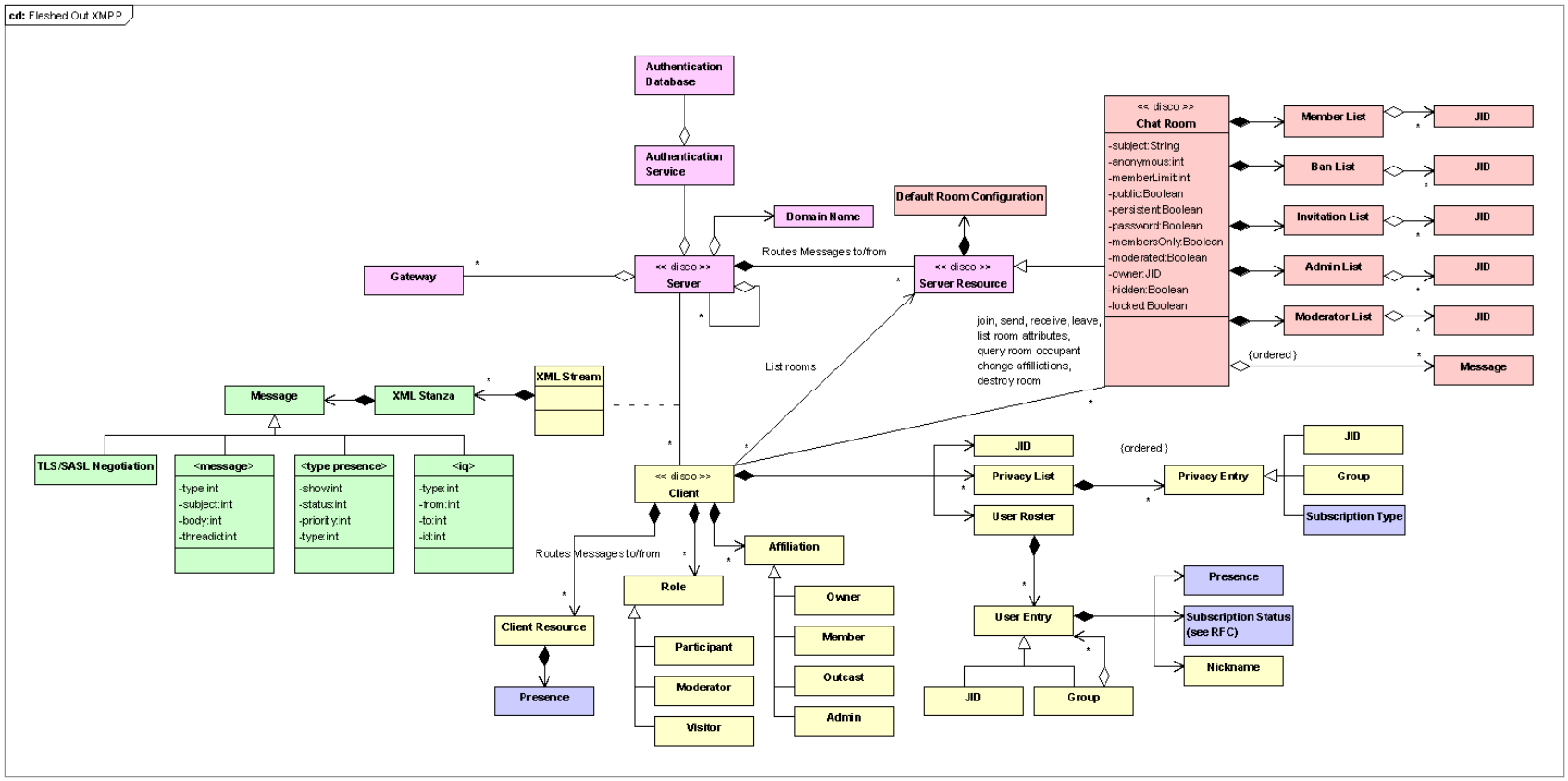


**Figure 17. Elaborated XMPP Model**

The highlights are:

- A chat room can have a number of client lists and a number of options (expressed as attributes). Clients can be banned and/or specifically invited, and can be administrators or moderators.
- A client can have visitor or moderator roles for a chat room. It can also be a chat room's owner, administrator, or an outcast.
- There are a number of different kinds of messages, including encryption negotiation, text content, presence, and resource interaction.
- There can be a number of servers, and they route messages between each other or through a gateway.

**SIP/SIMPLE**

The SIP/SIMPLE system is a layered system consisting of the Session Initiation Protocol (SIP) [25] and the SIP for Instant Messaging and Presence Leveraging Extensions (SIMPLE) [26][27].

SIP is the lower layer – it is an Internet-based signaling protocol for creating, modifying, and terminating communication sessions between users. It is defined across a number of RFCs, and it forms the basis of a number of commercial telephony products, including Voice-over-IP (VOIP) systems.

Fundamentally, SIP defines protocols that allow users to create sessions and negotiate the types of media to be involved in the session. A typical SIP exchange makes use of elements called proxy servers to provide a rich set of negotiation-time Internet telephony features, including:

- Internet telephone calls
- multimedia conferencing and distribution
- complex and dynamic routing
- user authentication and service authorization
- various call routing policies

Once the negotiation is complete, SIP delegates actual media stream processing to distribution elements called mixers.

Using SIP as a basis, a number of RFCs define value added services such as presence management [27][28][30][31][32][34][35][38] and instant messaging and chat [26][27][29][33][37]. For the sake of discussion, we incorporate all of these RFCs and their supporting RFCs into our working definition of SIMPLE, recognizing that SIMPLE is not rigorously defined in any RFC, nor has it been adopted as a standard by any major standards organization.

Note that a SIP-based system can accomplish instant messaging and chat using existing SIP facilities without relying on SIMPLE – text messages are just a simple media form, and once the endpoints are known, normal SIP signaling, routing, and mixing are adequate for transporting them. Accordingly, a chat room can be implemented as a conference call. SIMPLE provides value by simplifying and tailoring its processing and semantics to the needs of text-based (or object-based) exchanges, including providing a rich presence system.

A simplified view focuses on the relationship between major SIP/SIMPLE logical entities (grouped by color):

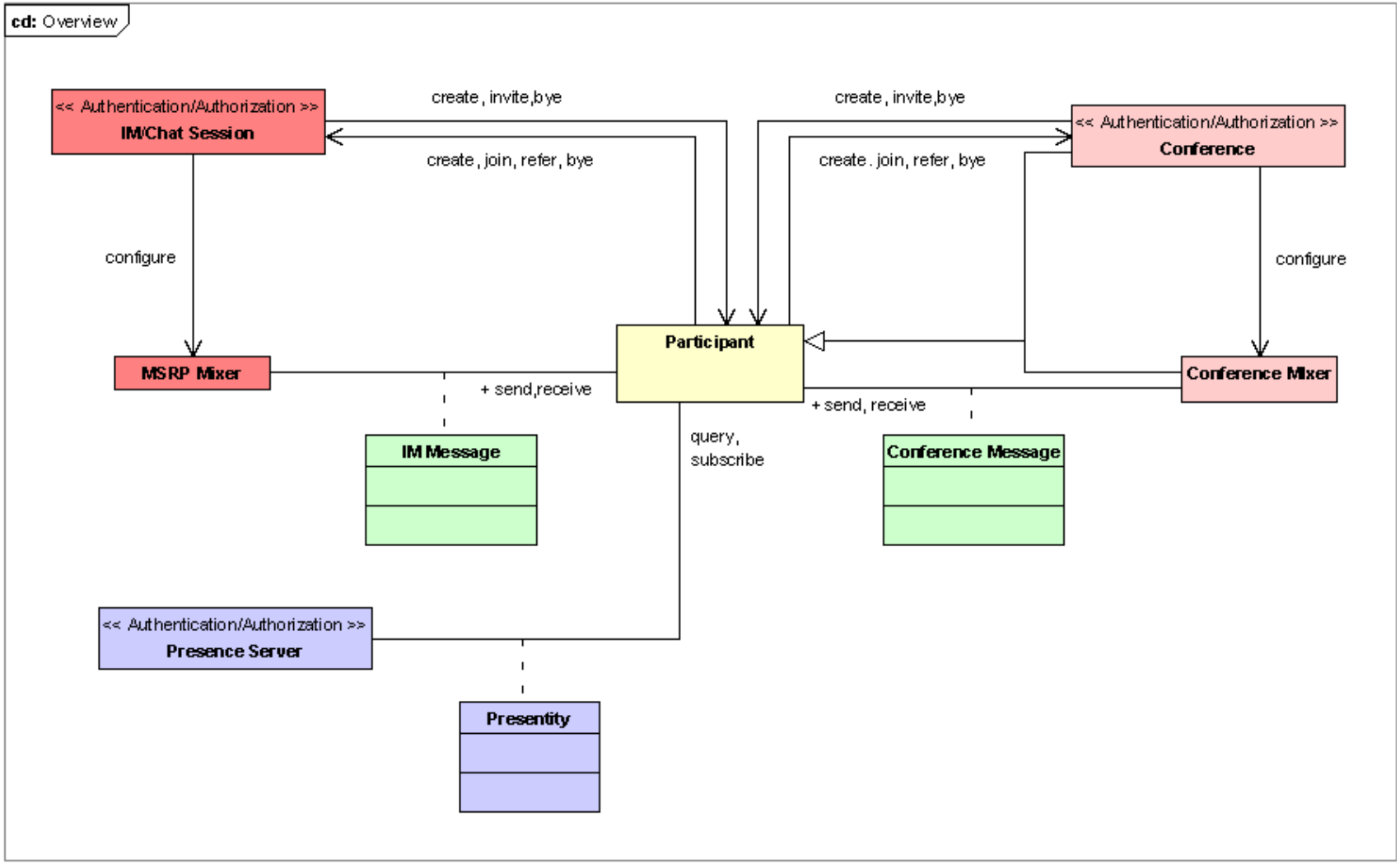


Figure 18. SIP/SIMPLE Overview

This view shows how messages and presence information are routed through the SIMPLE system, assuming an underlying SIP layer. Note that the Conference and Conference Mixer (in light red) are SIP entities, not SIMPLE entities. They are modeled in this view (and other views) because the SIMPLE RFC lacks the necessary precision and completeness, and reference to SIP-only implementations can clarify the intent of the SIMPLE RFC.

The highlights include:

- The Presence Server keeps track of Participants, including the Participants' SIP address(es), device(s), and device status(es). Participant information is modeled as a Presentity, which can be queried (or subscribed to) by other Participants.

- By interacting with an IM/Chat Session, a Participant can join or administer a chat room. The Participant can also initiate and participate in an IM session.

- Once a chat room or IM session is created, Participants exchange messages though the MSRP Mixer, which accepts messages and routes them according to the configuration supplied by the IM/Chat Session.

- One could create a chat room using standard SIP facilities (e.g., a Conference and Conference Mixer) in a manner similar to a SIMPLE conference. SIP conferences can be composed of subconferences.

- The Presence Server, IM/Chat Session, and Conference server can each require that the Participant be authenticated and authorized, though SIP/SIMPLE does not specify the details of this.

A view of the underlying SIP system reveals the relationship between the users, devices, proxies, and messaging:

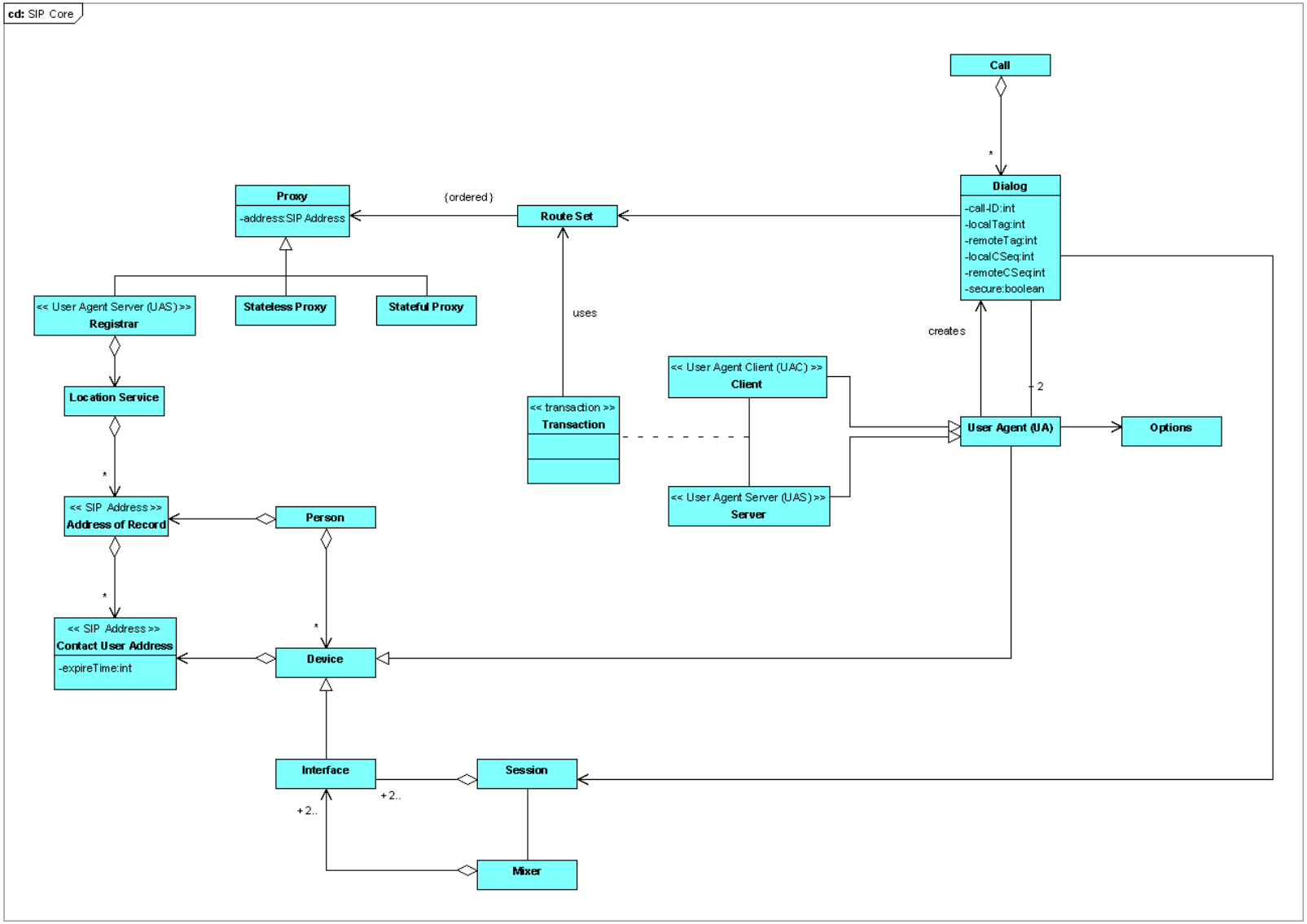


Figure 19. SIP Core

The highlights include:

- The central communication channel is the Dialog, which is an exchange of signaling messages between two User Agents (e.g., Participants at the SIMPLE level).
- The exchange may go through a number of proxies, thus establishing a routing for signaling purposes.
- One type of proxy is a Registrar, which can supply the address(es) associated with a user. The user can have several devices, each separately addressed.
- Media is exchanged between devices through a Mixer, which is set up as a consequence of signaling messages exchanged by User Agents.

In a SIP system, an address can take on a number of forms:

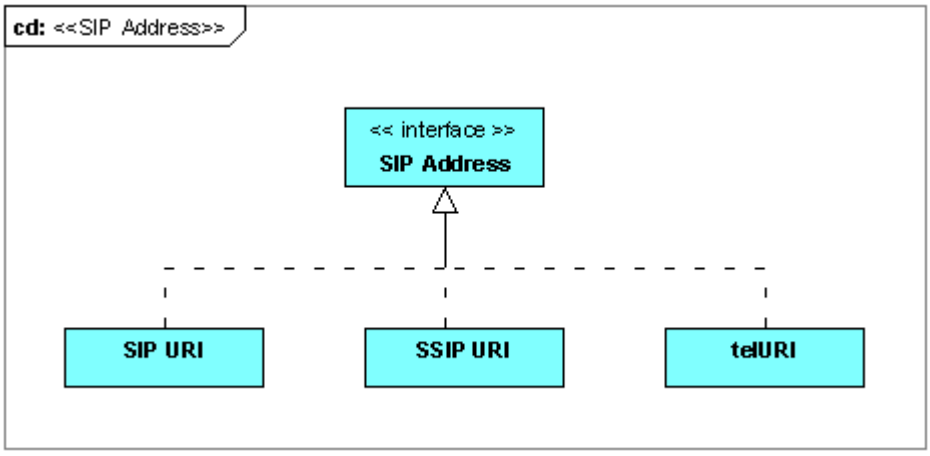


**Figure 20. SIP Address**

A SIP URI has the familiar format of a user name coupled with a domain name, and the domain name is resolved using normal DNS-style lookups. A SSIP URI is a URI that implies a secure channel (similar to the HTTPS namespace). A telURI is a phone number.

A detailed view reveals the mechanics of a signaling exchange:

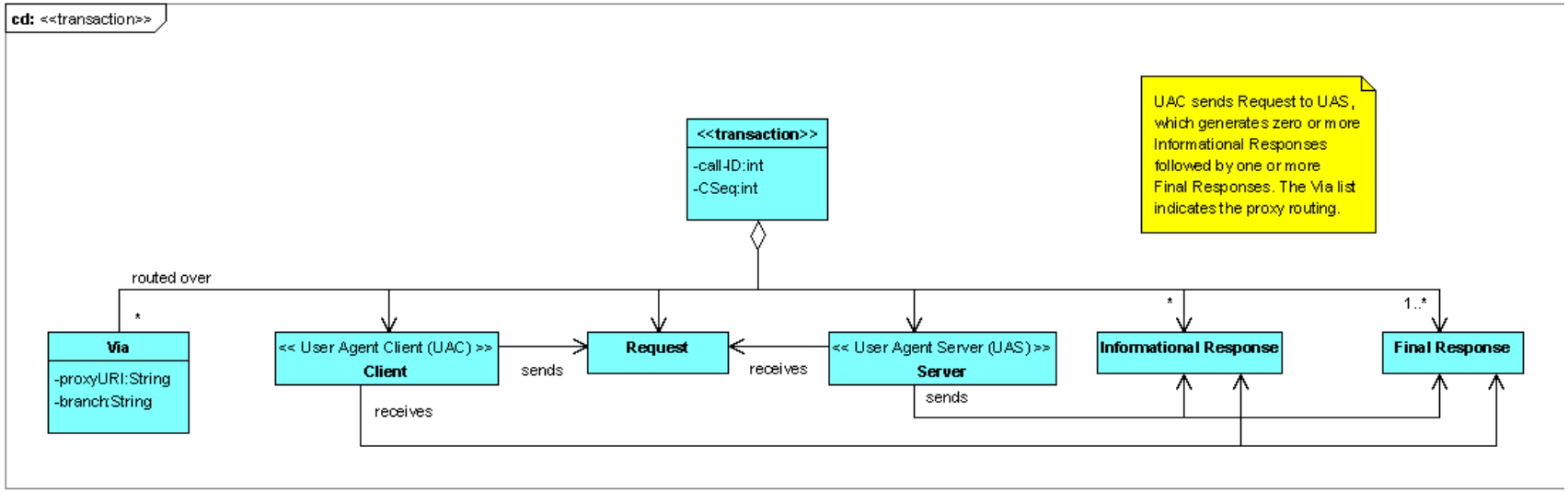


**Figure 21. SIP Transaction**

The highlights include:

- A client sends a request to a server, and the server replies by sending some number of informational responses followed by some number of final responses.
- The transaction carries its own routing.
- Requests and responses have a form similar to the HTTP protocol.

A view of a Presentity shows that it consists of a number of different kinds of information on a single person. (In fact, various RFPs not presented here optionally further enrich Presentity information.)

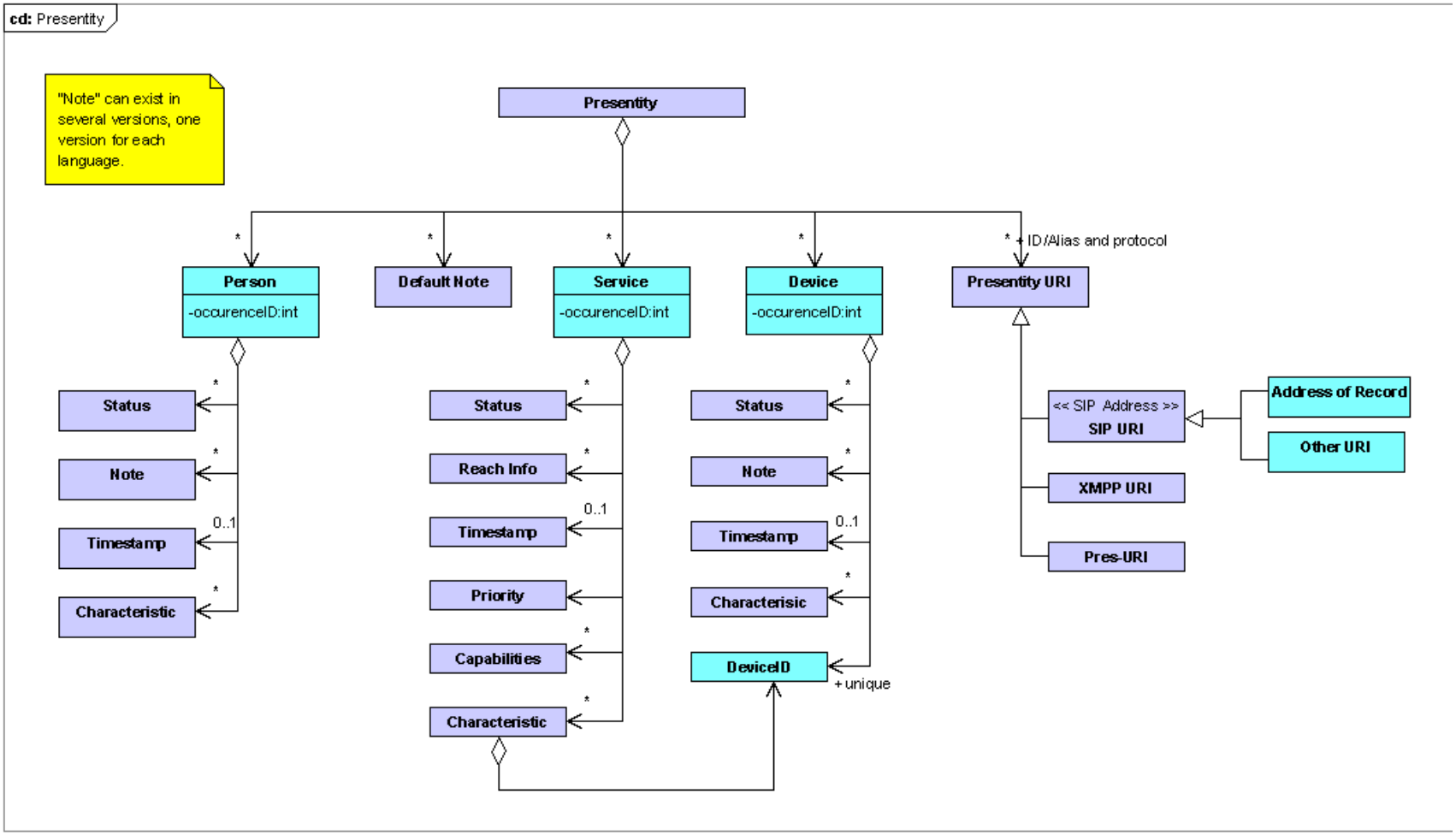


Figure 22. SIP/SIMPLE Presentity

The highlights include:

- A Presentity maintains all of the information pertaining to a person and his/her devices.
- A Person can have several different aliases.
- A Status is a state that can change.
- A Characteristic is an inherent attribute.
- A Timestamp indicates which is the latest version of Presentity information.
- A Person can have many devices, each having status and characteristic information.
- A Service is intimately tied to a device, and represents the ability of the device to perform some function. A Service can be reached though a SIP address, and when multiple services are available for a Person, a priority can be assigned.

A view of the Presence System shows the relationship between a Participant, Presentities, and the Presence System.

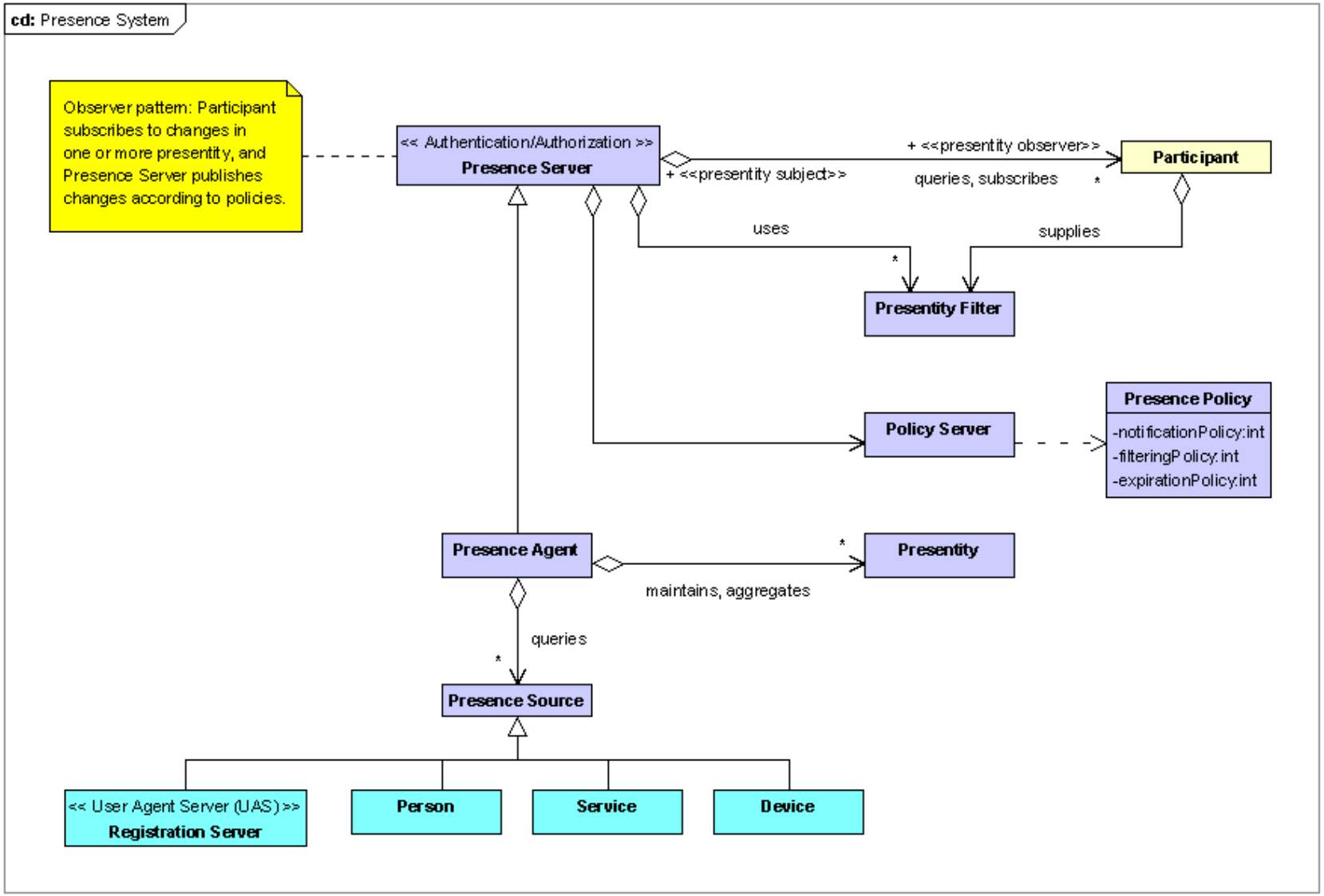


**Figure 23. SIP/SIMPLE Presence System**

The highlights include:

- A Presence Agent (often deployed as a controller for the devices belonging to a single person) acquires Presentity information from a number of sources, including devices, the person himself, or even by eavesdropping on Registration Server transactions.

- A Presence Agent publishes Presentity information to the Presence Server, which may then forward it to any number of interested Participants. As such, the Presence Server and Participant implement an Observer pattern.

- An interested Participant subscribes to Presentity information at the Presence Server, and it can provide a filter which allows the Presence Server to return only a subset of the Presentity information.

- A Presence Server observes preset policies governing the frequency of Presentity updates, filtering capabilities, Presentity expiration, and so on. As such, it implements the Strategy pattern.

A view of the SIMPLE layer shows the relationship between the Participant (i.e., user), the IM/Chat system, and the Mixer:

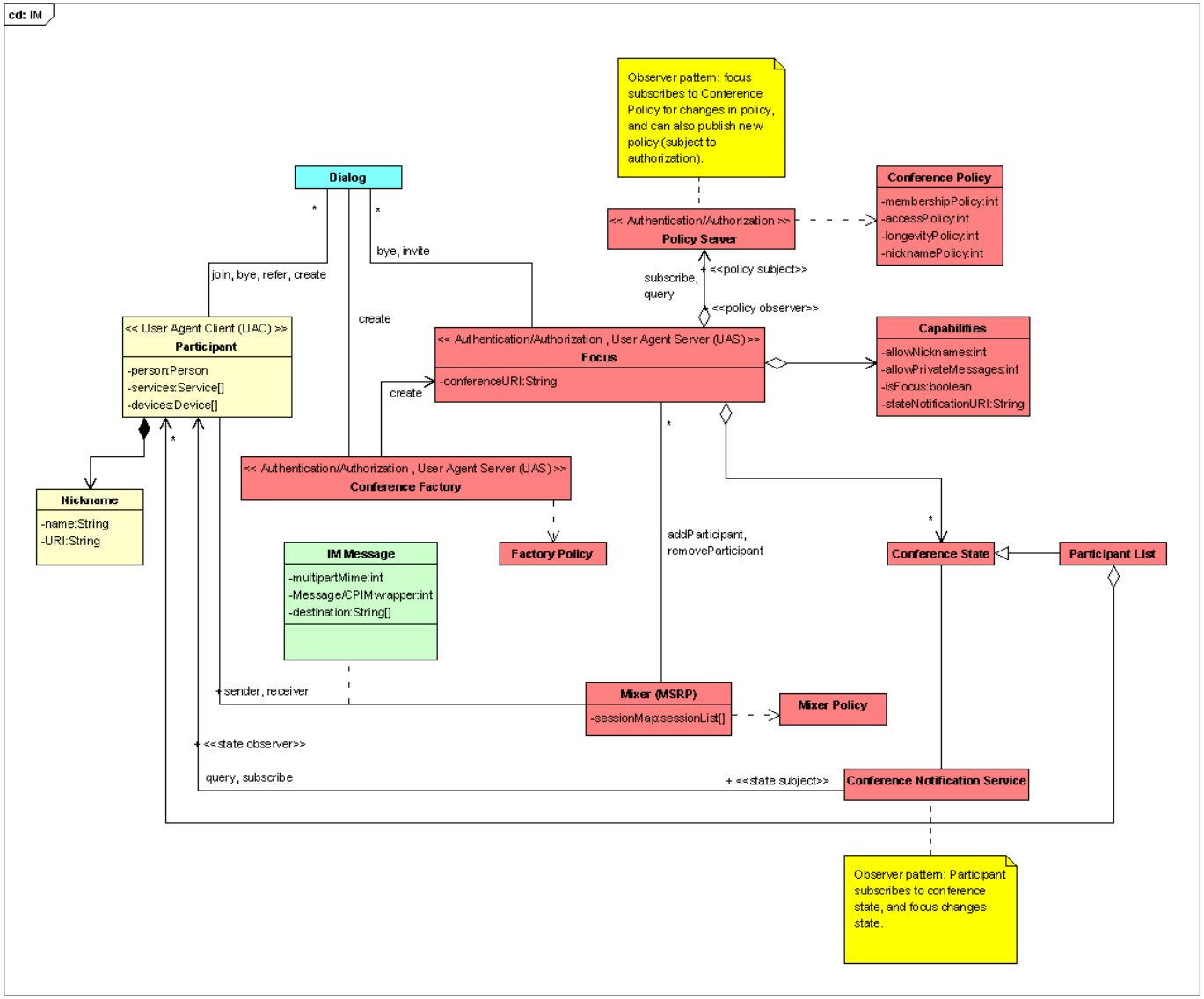


**Figure 24. SIP/SIMPLE IM/Chat System**

The highlights include:

- A Participant exchanges signaling information with the IM/Chat system by creating and using a SIP Dialog.

- A Participant can have a nickname and a number of devices, device services, and aliases. The nickname identifies the Participant in an IM or chat, and may have a SIP address by which the conversants can identify the Participant's true identity.

- A Participant creates or joins an IM or chat by addressing the Conference Factory, which then creates a Focus.

- A Focus is a controller that orchestrates the IM or chat. Its primary responsibility is to set up the MSRP Mixer so that messages from Participants are routed to the appropriate Participants. In this respect, the Mixer is the embodiment of an Observer pattern, though the subscription and observer functions are programmed by the Focus, acting as a proxy for the Participants.

- The Participants exchange messages through the Mixer, and the messages are MIME-formatted, which allows them to be composite in nature. The CPIM wrapper encodes security/signature information, and the message destination address allows a Participant to address all Participants or a smaller subset of Participants (as a sidebar).

- The Focus maintains a list of IM/Chat Participants, which a Participant can query or subscribe to. The Participant and Conference Notification Service cooperate in an Observer pattern.

- The Conference Factory, Mixer, and Focus each use the Strategy pattern to determine their constraints and characteristics. Additionally, the Focus subscribes to its Policy Server to become aware of any policy changes as they occur – another example of an Observer pattern.

Note that the SIMPLE RFP [26] is ambiguous in a number of respects, and numerous details of SIMPLE relationships and operating characteristics are unclear. The RFP references an RFP for SIP-based conferencing [37] as an authority, though it doesn't indicate in which respects.

A view of a SIP-based conferencing model shows structural similarities to SIMPLE, and contains features that probably exist in SIMPLE, too:

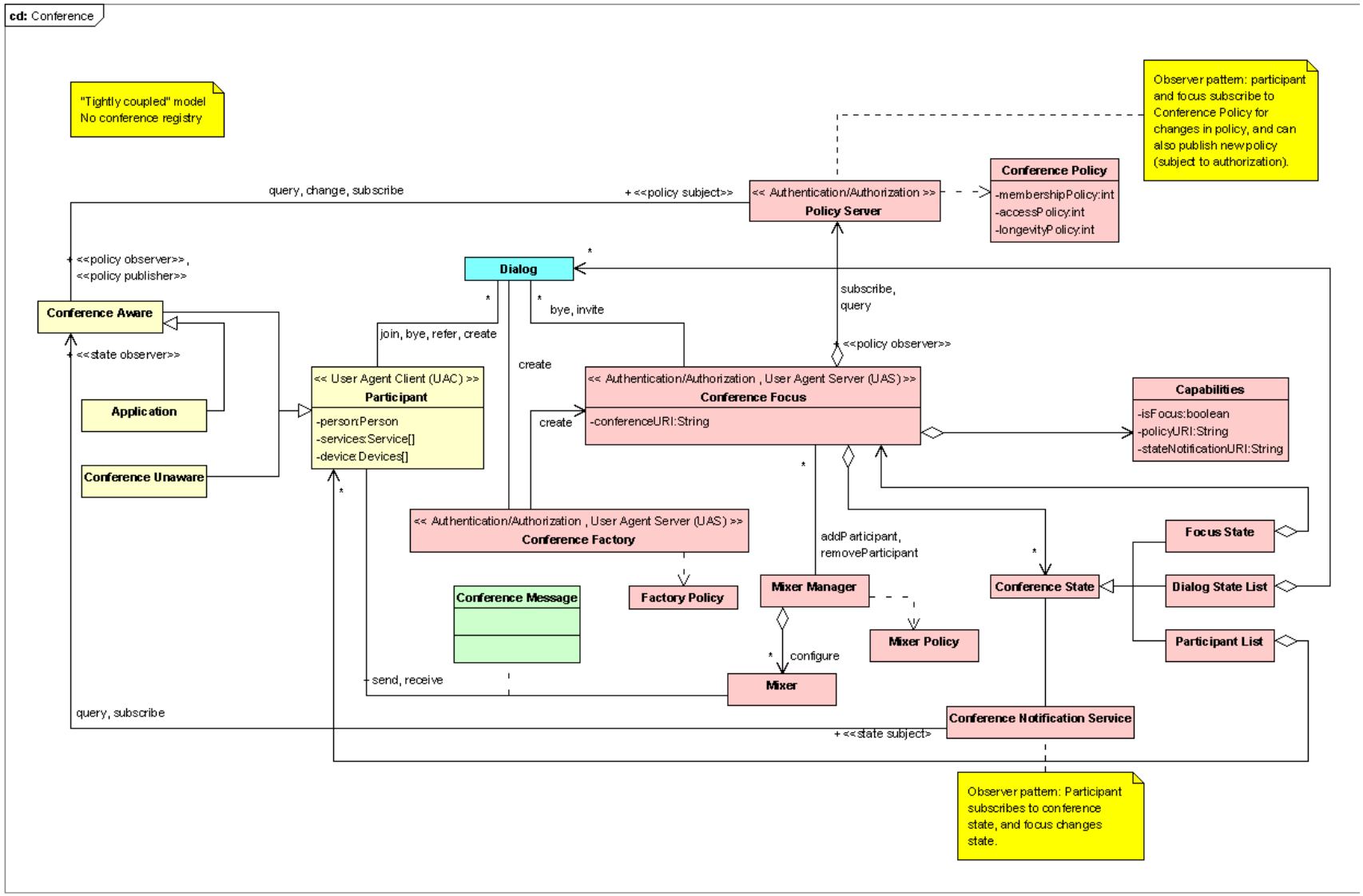


**Figure 25. SIP Conferencing System**

The highlights include:

- A Participant can be an application, or it can be completely unaware that it is participating in a conference. Conference-aware Participants can query the Conference Policy Server and the state of the Conference Focus.

- The Conference Policy Server can be affected by a Participant, presumably for administrative purposes.

- A Participant can not only query (or subscribe to) a conference Participant list, but can do the same for basic Conference and SIP status, too.

- A Mixer is complex in that it can actually consist of a network of Mixers, configured according to some policy.

## 5.6. Elaboration of Benefits and Problems

Each of the chat systems was built to solve different concerns, and they, therefore, have fundamentally different architectures. In this section, each chat system is described relative to its native ability to deliver the fundamental features of a Composable Chat system, namely authentication, authorization, presence, directory, and Rich Messaging.

**IRC**

A dominant feature of the IRC system is the creation of a distributed database (called the Global State Database) implemented by having the IRC servers constantly synchronize information such as presence, chat room lists, chat room memberships, and related information. Consequently, IRC maintains high availability, gently degrading its services in the face of network disruption. Because of this near-realtime synchronization, the scalability of an IRC system is limited as a function of the speed of server processors, network communication speed, and the frequency of network outages. Additionally, IRC does not emphasize secure communications, authentication, and expandability, though it does not prohibit them. Consequently, some IRC systems can be expected to contain these features, while others may not – and those that do may provide them in an ad-hoc manner.

Function authorization checks in IRC are limited to a few key cases: role-based command sets (e.g., enabling certain commands for system operators, chat room creators, and chat room operators) and chat room ban lists. There is no facility that associates and administers authorization at either the individual level or the group level across a wide range of chat system activities. To add extensive authorization checking and centralized administration would require invasive server-level modifications.

The presence system in IRC is provided as explicit queries from a client to its server, where the server returns the Global State Database's version of a target user's status. Because of server-to-server propagation delays, the status returned may be stale. In any case, querying for user status represents a load both on servers and on network bandwidth. The load is compounded as a result of IRC client software allowing its user to create a list of target users, then querying the server frequently for their presence status. Changing the presence system to a more efficient event-oriented system would require invasive server- and client-level modifications.

The identity system in IRC identifies users by self-defined nicknames, and prohibits more than one user with the same nickname on the system at the same time. There is no centralized directory system. (Because of network partitioning issues, this presents a significant opportunity for unauthorized and meddlesome acts.) Overall, the identity system itself places a practical limitation on the scalability of an IRC system.

The chat room system (called channels) operates similarly to the presence system in that chat room attributes and resources are replicated across all servers in an IRC network. Additionally, a message sent to a chat room on one server is replicated to all servers, and each server distributes the message to users that are connected to that server and are members of the chat room. The chat room system suffers the same scalability and propagation delay issues as the presence system, and suffers the same potential for unauthorized and meddlesome acts as the identity system.

The messaging system supports only text messages, and the support is further limited by client-based software which also supports only text messages.

The scaling, authentication, authorization, directory, presence, and messaging issues make IRC unsuitable as a general enterprise chat system, but it can be incorporated as a component of such a system as described in Section 6.

**XMPP**

The dominant features of the XMPP system are its data link encryption, reliance on XML messaging, provisions for SAML-based [20] authentication, point-to-point communications, and a server-mediated presence system. The data link encryption and SAML-based authentication lends an XMPP system a degree of security, depending on how it uses these features. The XML messaging provides explicit paths to feature extensibility, as demonstrated by the numerous XMPP extensions available. The point-to-point communications enables scalability, subject to load balancing deployments of services and network traffic. Because of the numerous choices of authentication mechanisms, extensions, and deployments, wide variability in deployed XMPP systems can be expected.

As in IRC, authorization checking in XMPP systems is limited to a few key role-based cases: user-defined privacy lists, chat room ban lists, and so on.

The presence system in XMPP is provided as a combination of the Proxy and Observer patterns. For each user, the user's server maintains the user's presence status. The server also maintains a list of users that have subscribed to the user's presence status. When the user's presence status changes, the server notifies each user in the subscription list. Such notifications are subject to blocking filters and other rules, which are defined and enforced at the server level. The server-centric event-driven nature of the presence system contributes to XMPP's scalability and efficient use of bandwidth and processing, though it incurs presence status and subscription list maintenance on the servers. Consequently, if a server is lost, all of the information pertaining to the users connected to it is lost, too. Additionally, to benefit from a particular server maintaining its subscription list, a user must always connect to the same server.

The identity system in XMPP identifies a user by a JID, which incorporates the user's server and nickname, and is validated and certified by the XMPP authentication system. Such a system implies the existence of a directory, but the directory exists only for authentication purposes. Regardless, as a result of the JID system, there is little risk of name collision as an XMPP system scales up.

The chat room system operates similarly to the presence system in that a chat room's attributes and resources are located on a single server, and several different servers can host their own chat rooms. It derives the same scalability benefits as the presence and identity systems.

While the messaging system supports only text messages, the messaging system is also extensible, and therefore provides the rudiments of support for Rich Messaging. However, the client-based software only supports text messages.

Under XMPP, chat rooms, users, and other services are implemented as *resources* addressable under a regular namespace designed to uniquely address each resource, and the namespace carries enough information to route messages point-to-point efficiently. Additionally, resources can be considered a base class on which enumeration queries and other methods are defined. Consequently, access to all types of resources carries a polymorphic character that fosters regular access, which in turn contributes to scalability, composition, and the addition of new services.

One consequence of XMPP's resource architecture is the discoverability of all XMPP resources. Should XMPP itself be used as a general enterprise chat system, all users would have access to all resources in the chat system (especially considering XMPP's lack of authorization checks). However, in a large system, this would be overwhelming in its sheer size, and unfettered resource access would constitute a security breach. For practical reasons, a large XMPP system would not make a suitable enterprise chat system on its own. Instead, it can be incorporated as a component of a Composable Chat system, which would enforce administrative and organizational properties on access to external resources.

**SIP/SIMPLE**

The SIP/SIMPLE system consists of the SIP telephony system extended by a number of modules, including a presence module and the SIMPLE IM/chat module. SIP is concerned with creating a secure, robust, feature-rich Internet-based signaling and distribution network that is modularly extensible. To these ends, SIP contributes a mapping between users and addressable devices, a scaleable and routable namespace, signaling over a network of proxies, and distribution over a network of mixers. SIP also leverages TLS and SRTP to provide secure point-to-point communications when appropriate. Presence modules are concerned with providing a rich and extensible presence and presence distribution model. The SIMPLE IM/chat modules are concerned with providing a scalable IM and chat facility that incorporates a rich messaging model.

Function authorization checks in SIP/SIMPLE appear to be based on a combination of authentication and policy evaluation. While the SIP standards allude to the authentication of users, they are generally silent on the details, and they do not address the authentication of devices and proxies (except for providing for an SSL-class encryption). User-oriented policy evaluation is defined to occur in presence servers, conference factories, and conference focuses. However, the nature of the policies, how they apply to individual users, and how they might result in function authorization are not addressed. Furthermore, the user model is silent on the concept of groups and roles. While it is possible to create SIP/SIMPLE implementations that account for these issues, such implementations are unlikely to be interoperable in the absence of further standards work. For a particular SIP/SIMPLE implementation, it is plausible that such facilities could be provided via proxies introduced onto the signaling network – this is an area for future research.

The presence system in SIP/SIMPLE comprises a core presence definition and a query-enabled,

event-driven distribution system. The basic presence definition incorporates the status of any and all devices identified with the user. Multiple standards define additional presence features that can be added into the basic presence definition, and a filtering mechanism is identified (but not defined) in order to allow a user to receive only desired subsets of the available presence information. While policy mechanisms exist to define operational characteristics of the presence server, there appears to be no standards regarding user-defined or policy-based limitation of presence dissemination or receipt. While such mechanisms could be added, intrusive changes to both SIP software and client software would be necessary. Additionally, because presence extensions carry no semantic information and lack feature set query mechanisms, client software may not be interoperable between SIP-based implementations.

The identity system in SIP/SIMPLE is limited to the assurance that no two users in the same chat room are allowed to have the same nickname, along with a recommendation that nickname uniqueness be enforced across multiple chat rooms residing on the same server. Even so, no mechanism is defined for the enforcement of either of these two policies. Additionally, while the presence and SIMPLE specifications recommend user authentication, neither provide any authentication mechanism, nor do they reference any centralized identity repository. While authentication facilities could be provided via proxies introduced onto the signaling network, intrusive changes would be required in existing client software. To the extent that authorization mechanisms are provided in existing SIP/SIMPLE systems, client software may be non-interoperable due to a lack of standards.

The SIP/SIMPLE specification indicates that in order to join a chat room (sometimes called conference), a user must know the chat room's SIP address. Alternately, it proposes that when a conference factory creates a conference, a broadcast may be made to advertise the conference's address. Though implementation of a conference repository may be possible via additional proxies introduced onto the signaling network, intrusive changes would be required in existing client software. To the extent that repository mechanisms are provided in existing SIP/SIMPLE systems, client software may be non-interoperable due to a lack of standards.

While the SIP core and its interfaces are reasonably well defined, the presence and SIMPLE module definitions appear to be in flux and are underspecified. Consequently, SIP's modularity encourages the existence of multiple IM, chat, and client implementations that may not be stable, common, or interoperable. Additionally, while the SIP core and extensions regularly posit policy servers and authentication/authorization processing for key control points, there appears to be no SIP standards defining them. Consequently, different implementations are likely to be non-interoperable. From a chat perspective, this results in silos of SIP-based chat instead of an interoperable chat fabric.

## 5.7. Coverage of Chat System Models in Conceptual Chat Model

As discussed in Section 5.3, the objective of modeling a conceptual chat system is to decouple the concepts behind chat from the specific deployment choices and constraints found in concrete chat systems. While the conceptual model focuses on the relationship between conversations, participants, messages, and presence, the concrete models serve particular implementations by weaving those entities into specific implementation protocols. Given the different purposes of the two types of models, they must be compared to verify that the conceptual model has not missed any concept essential to its purpose.

**IRC**

The conceptual model does not cover IRC's server/router design and the server management functionality. Additionally, the nuances of specific IRC implementations are not conceptually modeled.

**XMPP**

The conceptual model does not cover XMPP's XML stream and JID addressing scheme. While XMPP's participant exclusion system (e.g., chat room ban lists, outcast roles, and visitor roles) is not directly represented in the conceptual model, it is represented as policies defined and applied at the conversation level. Additionally, the ramifications of various XMPP extensions are not modeled.

**SIP/SIMPLE**

The conceptual model does not cover SIP/SIMPLE's bifurcated command/control and data paths and the decomposition of a presentity into relationships between persons, services, and devices.

None of the differences between the conceptual model and the concrete models conflict with or invalidate our use of the conceptual model for defining a Composable Chat system. While the transport protocols and novel features implemented within a particular chat system are critical to the function of that chat system, the mission of the Composable Chat System is to abstract the essential functionality of each chat system, and then provide an integration layer that bridges chat systems while preserving chat system semantics across the bridge. Section 3.2 discusses the ramifications of this approach.

## 5.8. Defining the Service Interactions between Roles

Given a conceptual chat model, we hypothesize a Composable Chat system consisting of the conceptual model composed upon itself, thereby creating a chat system of chat systems organized as a two tier hierarchy: the enterprise layer and the chat system layers.

This two tier model fulfills the assumptions pertaining to the Composable Chat system defined in Section 2.2. Each chat system is self-contained and serves its members without modification or disruption. Additionally, it allows non-native members and chat rooms to be accessed by native members, and publishes its chat rooms for discovery by other chat systems.

However, in order to accommodate the requirement that the chat system itself not be modified, the chat system layer is bifurcated into a chat interface layer and the chat system layer – the actual chat system resides in the chat system layer.

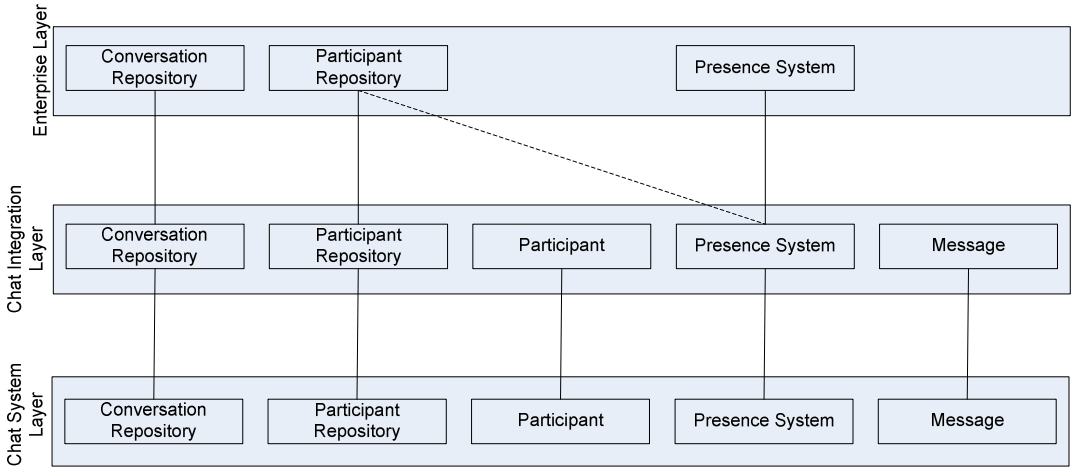


**Figure 26. Composable Chat Layers and Roles**

The roles provided by the enterprise layer include the enterprise-wide conversation, participant, and presence system repositories. They are capable of storing and disseminating the aggregation of information on conversations and participants located throughout the Composable Chat system.

The roles provided by the chat integration layer corresponded to the main entities of the conceptual chat model's core domain: conversation, conversation repository, participant, presence system, and the message. The conversation repository, participant repository, and presence system roles interact with corresponding enterprise roles to publish conversation and participant information pertaining to the associated chat system, and to receive such information pertaining to other chat systems.

The roles provided by the chat system layer correspond to the functionality of the actual chat system. They interact with the chat system integration layer roles to publish actual conversation and participant information, and to receive such information pertaining to other chat systems.

Essentially, the chat integration layer roles act as proxies and facades, leveraging the capabilities of the actual chat systems to present a consistent face to the enterprise layer. Because actual chat systems vary in their capabilities and service entry points, the interactions between the chat integration layer roles and the chat system layer roles are highly dependent on the actual chat system. For example, the IRC presence system is passive – presence information is not distributed unless asked for by a client. The interaction that supports this would be a query from the chat system to the integration layer presence system roles. The XMPP presence system is active – changing presence information is published to subscribers. The interaction that supports this would be the Observer pattern.

Note that the semantics of the interactions between the enterprise layer and the chat integration layer may be either all inclusive of chat system capabilities or may reflect the lowest common denominator of chat system capabilities. As discussed in Section 3.2, determination of this strategy is a strategic design decision; a SOA-based design can accommodate the entire spectrum of possibilities, as well as tractable migration paths between them.

Consistent with the conceptual model, the chat integration layer roles are mirrors of the

corresponding enterprise roles, subject to filtering imposed by the authorization system. (For example, the chat integration layer presence system role reflects the enterprise layer presence system, with the restriction that it presents only individuals that are authorized to use the chat system. To achieve this filtering, the capability lists for individuals must be consulted.)

Note that an important notation in describing the interactions between roles is a Message Sequence Chart (MSC) [2]. Ideally, all services would be modeled by drawing MSCs so as to accurately capture them and to stimulate thought leading to the discovery of yet other interactions. For the purposes of this paper, it is sufficient to merely mention this technique.

### 5.9. Composing the Rich Services Hierarchy

The Rich Services hierarchy is a direct consequence of the interactions between roles. Encapsulating each hierarchy layer (defined in Section 5.8) into a separate Rich Service accomplishes these interactions, and confers additional benefits:

- Rich Services messages traveling across a Rich Service message bus are exposed for intercept and additional processing by Rich Interface Services (RISs). Processing for crosscutting concerns is thus easily customized to the needs of the corresponding hierarchical layer.
- Interactions between Rich Services can be mediated through a Service/Data connector, which can add additional value customized to the link between the Rich Services.

To create this hierarchy, the roles identified in each hierarchical layer are modeled as Rich Application Services (RASs) and are connected by a common message, routing system, and Service/Data connector.

By examining the use cases as applied to the various roles and then tying them back to the original requirements, it is possible to clearly identify crosscutting concerns pertaining to the use cases and associate them with particular interactions between particular roles [24]. Each crosscutting concern is modeled as a RIS, and the message routing system is configured to include the RIS at the appropriate points in the interaction between RASs.

An example of Rich Service composition as applied to Composable Chat is presented in Section 6. The layered hierarchy is modeled as three Rich Services, with RISs applied to two of the Rich Services.

Note that the Rich Service hierarchy models the logical relationships between roles and services, but not, *per se*, the deployment relationships. A Rich Service model can be mapped to a number of deployment scenarios as described in [1]. The mapping between the logical relationships and the physical deployment can be very simple – each Rich Service can be deployed as an Enterprise Service Bus [21] running on its own server, as services interacting over multiple servers, or even as traditionally linked sub-systems.

## 6. Integration Architecture

Building the Composable Chat system requires integrating existing chat systems with policy-driven functionality at each layer according to the structure identified in Section **Error! Reference source not found.** and subject to the assumptions in Section 2.2. At the integration layers, value-added functionality includes authentication services, authorization, presence management, and directory services, all subject to crosscutting concerns including logging,

auditing and information assurance, bandwidth management among others. The Rich Services architecture described in Section 4.2 provides a hierarchically decomposed structure that supports this three-layer structure.

Considering that the Composable Chat system is implemented as a Rich Service, it can support interactions with other system components such as workflow orchestration and choreography services. Such interactions would flow through the Composable Chat Rich Service's Service/Data connector. The following sections describe the Composable Chat Rich Service, which is essentially the Enterprise Integration Layer Rich Service (Figure 27) combined with a Service/Data connector (not shown).

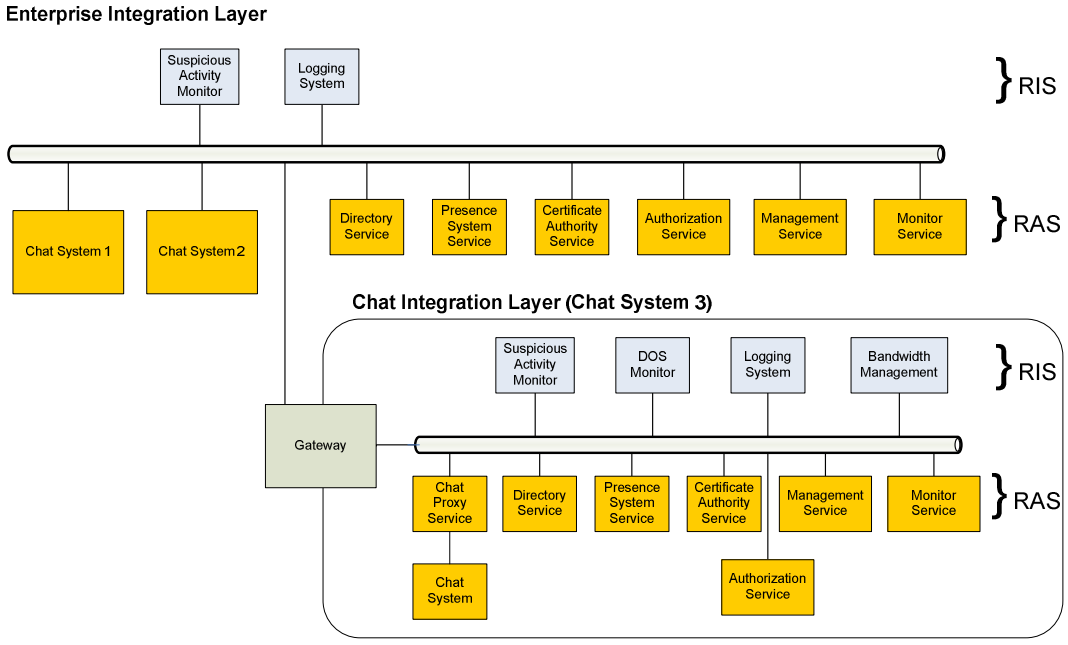


**Figure 27. Composable Chat Integration Model**

## 6.1. The Enterprise Integration Layer Rich Service

The enterprise integration layer is a Rich Service consisting of enterprise services and the collection of chat system integration layers, themselves all implemented as Rich Application Services (RASs). Crosscutting services such as logging and suspicious activity monitoring (and possibly others) are implemented as Rich Infrastructure Services (RISs). Interaction with and amongst the RASs is carried out via messaging over the common bus, subject to interception and processing by the RISs as defined by the layer's routing policy.

**The Chat System RASs** provide chat services to a command and whatever other individuals are authorized to use them. Each chat system interacts with the Directory Service, Presence System Service, Certificate Authority Service, and the Authorization Service to provide communications and resource access to individuals both inside and outside of the chat system. Their function is described in Section 6.2.

**The Directory Service RAS** provides the concept of identity, which establishes a one-to-one mapping between a person and his or her information. Such information could include anything

the enterprise deems important, which would include a list of the capabilities the user is authorized to execute and a list of chat system roles played by the individual. (Maintaining a list of chat system roles enables an individual to assume one persona on one chat system, and another persona on another chat system.)

Note that this RAS can implement a directory repository itself, or it can cooperate with an enterprise directory (per Core Enterprise Services) to implement this role. This RAS does not initiate interactions with any of the other RASs, but instead remains available for them to interact with.

Note that another likely function of the RAS would be to use the identity repository to represent the mapping between a user's chat-specific persona and his or her true identity.

Note that for the purposes of this illustration, the directory service also maintains information on chat rooms, with both similarities (e.g., a list of capabilities and chat system-specific aliases) and differences (e.g., chat room attributes).

This service provides a publish/subscribe-style service whereby changes to directory service information can be reflected to interested parties such as Chat Integration Layer Rich Services.

**The Presence System Service RAS** maintains a correlation between an identity and an individual's online status. It interacts with chat systems to keep the status updated and to deliver new status as an individual's online status changes. It provides a publish/subscribe-style service whereby changes to an individual's presence status can be reflected to interested parties such as Chat Integration Layer Rich Services.

**The Certificate Authority Service RAS** verifies that a certificate presented by a user is valid. This RAS does not initiate interactions with any of the other RASs, but instead remains available for them to interact with. During authentication, the authenticating service would interact with this RAS and the Directory Service RAS.

**The Authorization Service RAS** determines whether an individual has the credentials to perform a particular action on a given resource. Chat systems interact with it to determine whether a user has access to a chat room, to personal communications, or to other functionality as indicated by the capability list provided for the user by the Directory Service RAS.

**The Management Service RAS** allows the maintenance of the user capability list maintained by the Directory Service RAS. It interacts with client-side maintenance programs that assist in this task, and provides input to the Directory Service RAS.

**The Monitor Service RAS** enables administrators to determine the status of the Composable Chat system. The service is defined to present the status of resources at the enterprise integration layer, and to aggregate the status of resources presented by chat integration layer's RASs (using their Monitor Service RASs via their Service/Data connector). Additionally, it interacts with other RASs to provide additional status, and it interacts with a client-side monitor program, which queries and controls it.

**The Suspicious Activity Monitor RIS** is an example of a RIS that monitors messages traveling between RAS and RIS components on the message bus. By whatever means, it determines what message patterns may indicate a breach, and may quash errant messages. It interacts with the Monitor Service RAS to provide historical information.

**The Logging System RIS** is an example of a RIS that logs some subset of messages traveling between RAS and RIS components on the message bus. The logging policy can be set administratively, and the log can be retrieved by the Monitor Service RAS.

Note that any number of RISs can be defined for a Rich Service, and can implement any number of crosscutting services. The particular messages intercepted by a RIS are determined by the Rich Service's routing policy, which can be either static or dynamic. For this particular Rich Service, the routing policy is that all messages pass through all RISs.

Note that both the Presence System Service RAS and the Directory Service RAS provide interactions in a publish/subscribe-style so as to maintain near-realtime information flow to roles needing timely information. Any other RAS could adopt this pattern should its client roles have a need.

## 6.2. The Chat Integration Layer Rich Service

The function of the Chat Integration Layer Rich Service is to integrate an actual chat system into the Composable Chat system by both interacting with the Enterprise Integration Layer Rich Service and with the chat system itself. As such, it creates a functional bridge between the functions of the chat system and the functions of the Composable Chat System.

**The Gateway** component performs the role of the Service/Data connector. To the Enterprise Integration Layer Rich Service, it is a proxy for the Chat Integration Layer Rich Service, and performs all interactions with that service. To the Chat Integration Layer Rich Service, it is a proxy for the Enterprise Integration Layer Rich Service, and performs all interactions with that service.

**The Chat Proxy Service RAS** interacts with the actual chat system to achieve authentication, authorization, presence, and Rich Messaging effects. The form and substance of the interaction depends on the capabilities of the actual chat system and the service interfaces the chat system exposes.

**The Directory Service RAS** interacts with the chat proxy service to perform services similar to the Enterprise Integration Layer's directory service, except that for a given user, it adds the list of capabilities defined for the user at the level of the that owns the actual chat system. In this way, user-level policy is effectively provisioned both at the enterprise level and at the local level.

The directory service RAS can be implemented in a number of ways, depending on the circumstances of the actual chat system. For example, a simple directory service might be defined to store its capability lists in the enterprise directory, and could then map all interactions into interactions with the Enterprise Integration Layer's directory service. This implementation would be appropriate for chat systems having a reliable, high bandwidth connection to the Enterprise Integration Layer Rich Service. More likely, it would conserve bandwidth or mitigate the effects of communications dropouts by caching interactions with the Enterprise Integration Layer's directory service, or simply replicating portions of the enterprise identity repository. Should the directory service RAS choose to either strategy, it would interact with the Enterprise Integration Layer's directory service to receive directory change notifications in a publish/subscribe style.

**The Presence System Service RAS** interacts with the chat proxy service to perform services similar to the Enterprise Integration Layer's presence system service. For each of the chat system's authorized users, it maintains a presence status. Users internal to the chat system are

automatically considered to be authorized users, and users external to the chat system are considered to be authorized by virtue of a capability list at either the Chat Integration Layer or at the Enterprise Integration layer (i.e., a system-level policy or a command-level policy).

For users internal to the actual chat system, it acquires the presence status by either occasionally querying the chat proxy service, or receiving a publish/subscribe-style change notification from that service, depending on the capabilities of the underlying chat system. It then notifies the Enterprise Integration Layer's presence service of the change. For users external to the chat system, it acquires and caches the current presence status via a publish/subscribe-style change notification from the Enterprise Integration Layer's presence service. Depending on the capabilities of the chat system, this service either notifies the chat proxy service of the change, or caches the status until the chat proxy service requests it.

**The Certificate Authority Service RAS** interacts with the chat proxy service to provide certificate validation services for authentication purposes. It may provide pass-through services to the Enterprise Integration Layer's certificate authority service, or it may cache or replicate enterprise-level decisions, or may adopt other locally motivated policies.

**The Authorization Service RAS** interacts with the chat proxy service to determine whether a particular individual is authorized to perform a given action on a particular resource. It may apply any number of strategies in making this determination, including making a local decision based on the subject, action, and capability list presented to it. Additionally, it could use completely different criteria, based on local policy, or it could defer the decision to the Enterprise Integration Layer's authorization service.

**The Management Service RAS**, **Monitor Service RAS**, **Logging System RIS**, and **Suspicious Activity Monitor RIS** all behave similarly to their counterparts in the Enterprise Integration Layer Rich Service, except they implement policies and algorithms appropriate to the particular the chat system.

**The Bandwidth Management RIS** service is an example of a RIS having a Chat Integration Layer function but no Enterprise Integration Layer counterpart. This service decimates outbound content according to particular policies when the outgoing bandwidth is insufficient to support that content. Such a determination would be independent of any other chat system activity, and would therefore qualify as a crosscutting concern implemented as a RIS.

Similarly, the **DOS Monitor RIS** service is an example of a RIS that determines when a denial of service attack is underway, then quashes all messages related to the attack. The determination of such an event would be Chat Integration Layer dependent.

Note that for the sake of this discussion, a Chat Integration Layer Rich Service is assumed to be trusted by the Enterprise Integration Layer services, which means that the Chat Integration Layer Rich Service authenticates members internal to the chat system, and performs no attacks on the Enterprise Integration Layer services. If this assumption is relaxed, additional protective services must be added at the Enterprise Integration Layer to isolate the Chat Integration Layer.

### 6.3. The Chat System Rich Service

The Chat System Rich Service consists of the Chat Proxy Service and the actual chat system. In fact, this Rich Service is best represented by a separate message bus with a collection of RASs and a Service/Data connector. For the sake of brevity, Figure 27 shows only the Chat Proxy

Service RAS, which contains all of the functionality and semantics of a Service/Data connector and RASs, and interfaces directly with the actual chat system.

As described in Section 6.2, the Chat Proxy Service RAS interfaces with Chat Integration Layer RASs and the actual chat system. It projects entities internal to the chat system into the Composable Chat system via interactions with the Chat Integration Layer RASs. Similarly, it projects entities external to the chat system into the chat system via service interfaces presented by the chat system.

When a chat system supports near-realtime updates, the chat proxy service interacts with Chat Integration Layer RASs to implement them. For example, the presence system for XMPP publishes and receives presence status changes in near-realtime. Consequently, when the chat proxy service receives a presence status change notice from an XMPP server, it interfaces with the Chat Integration Layer's Presence System RAS to make the Composable Chat system aware of it.

When a chat system does not support near-realtime updates, the chat proxy interacts in a query/response exchange with Chat Integration Layer RASs. For example, the presence system for IRC is passive, and does not report presence status unless queried by a client. Consequently, the chat proxy maintains the Composable Chat system's view of the presence status by polling an IRC for the presence status of all internal chat members.

In either case, the chat proxy service interacts with the Chat Integration Layer's Presence System RAS to subscribe to presence status change notifications for users external to the chat system. When it receives such a notification, it forwards it to a chat system server for immediate propagation through the chat system.

The chat proxy service adopts similar strategies for reflecting authentication, chat room, and other kinds of information to the chat system and the Chat Integration Layer.

## 6.4. Deployment Architecture

Note that while the Rich Service integration architecture models the logical relationships between roles and services, it *can* also model the deployment relationships. A Rich Service model can be mapped to a number of deployment scenarios as described in [1]. The mapping between the logical relationships and the physical deployment can be very simple – each Rich Service can be deployed as an Enterprise Service Bus [21] running on its own server, as services interacting over multiple servers, or even as traditionally linked sub-systems.

For a Composable Chat system, a plausible deployment would be to create ESB implementations for each of the Rich Services:

- The bulk of the Chat System Rich Service would be deployed as an ESB on one server, with a network link to the actual chat system. For each chat system, there would be one such server, and it would run a version of the Rich Service appropriate for the chat system. The Chat System Rich Service would be collocated with the chat system server.
- For each chat system, there would also be a Chat Integration Layer Rich Service configured to compliment the Chat System Rich Service and running on the same physical server.
- For the Enterprise Integration Layer Rich Service, there would be a single server (or server complex) in a centralized location with WAN service to each of the Chat

Integration Layer Rich Service servers.

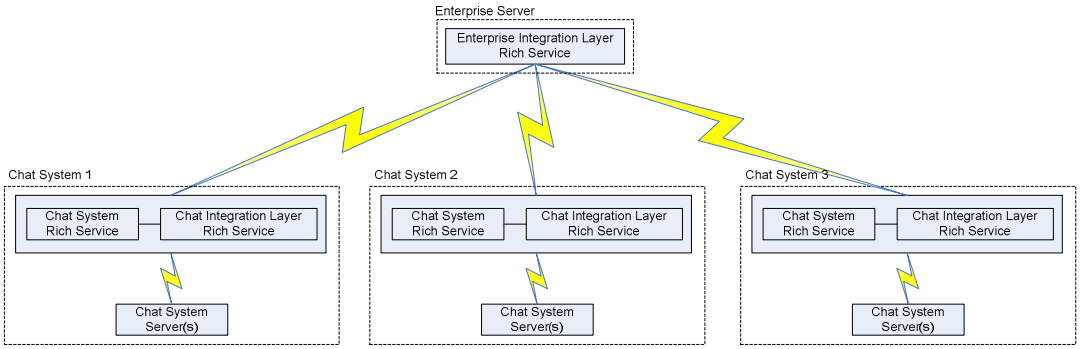


**Figure 28. Composable Chat Deployment Model**

## 6.5. The Rich Messaging Support

The support for Rich Messages exists in two locations: the Chat System Rich Service and the chat client.

At the Chat System Rich Service, the chat proxy service must filter a Rich Message according to the privileges of the user receiving the message. For each portion of the message, the service must determine whether the portion can be forwarded to a user, and then create and forward a Rich Message consisting of only the authorized portions. For outbound messages, the service must determine whether the sender has privileges to send each portion of the outbound message, then edit out any unauthorized portions. To make these determinations, the service must interact with the Chat Integration Layer Authorization Service RAS.

A normal chat client is capable of displaying text messages – displaying Rich Messages requires that message contents be distributed to a number of message viewers, including the normal chat client. To do this, a Rich Message proxy is introduced at the client, and the proxy distributes each piece of the Rich Message to an appropriate viewer.

As a practical matter, the proxy runs on the client and intercepts inbound chat traffic before the chat client can receive it. Additionally, the viewers run on the clients. This arrangement supposes that such proxies and viewers can be deployed to clients in a chat system. In cases where this is not possible, the chat proxy service would filter out all non-text portions of a Rich Message before forwarding it to the client. Developing the means to sense this situation requires more work, and is beyond the scope of this paper.

# 7. Example: Integration of XMPP and IRC

An example of the application of the Composable Chat integration architecture (Section 6) can be found in a demonstration of a simplified propagation of presence status between two different kinds of chat systems: XPPP and IRC. While these chat systems share a similar concept of presence, they maintain and disseminate presence information in very different ways, using very different protocols. Additionally, each chat system has the problem of acquiring the presence status of users external to itself and who are authorized to use the chat system.

This example demonstrates the use of all three layers of the architecture, involving both RASs

and RISs.

For the sake of simplicity, assume that there are two chat systems, $C_1$ and $C_2$, with $C_1$ being an XMPP system and $C_2$ being an IRC system. Assume also that each chat system has its own members, and that some members of each chat system are authorized to access the other chat system. (From the point of view of the other chat systems, the latter would be considered to be external users.) This example shows how changes in members' presence on each system propagate across the infrastructure and appear on the other system. Additionally, the example shows how the changes are logged on both systems and at the enterprise layer.

For the sake of simplicity, we extract only the integration architecture components relevant to this example:

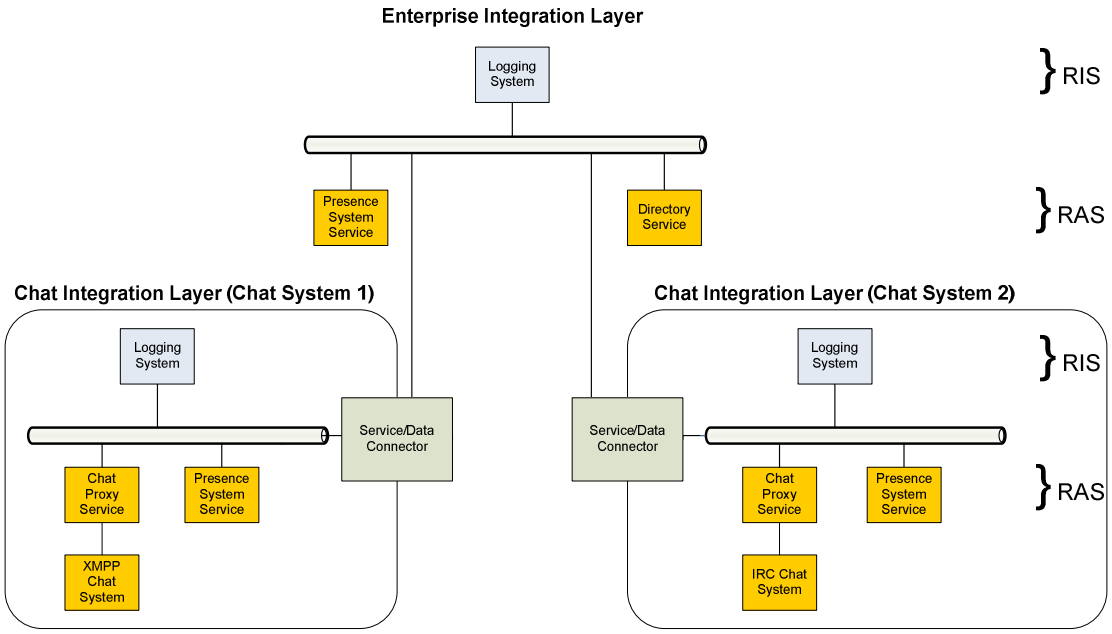


**Figure 29. Sample Integration of XMPP and IRC Presence System**

The solution to acquiring the presence status for all internal users is solved differently, depending on the particular chat system. For this example, the Chat Proxy Service for XMPP would subscribe to the presence status for all XMPP chat system members – the Chat Proxy Service would discover that a member's presence had changed when it received a notification from the XMPP presence system. In contrast, the Chat Proxy Service for IRC would periodically poll an IRC server for status of all users online, and then compare the result to the last list it received – the Chat Proxy Service would know that a member's presence had changed when the new status did not match the old status – with a similar calculation to determine when a user had logged on or off of a server.

In either case, the presence status changes would be forwarded to the Chat Integration Layer's Presence System Service, which would forward it to the Enterprise Integration Layer's Presence System Service. For this example, the Enterprise Integration Layer's Presence System Service is defined to implement the Observer pattern, and it creates a different observer list for each individual logged onto a chat system anywhere in the enterprise. (There are alternate

implementations, though this implementation suffices for this example.) For a given individual, the observers are defined to be the list of chat systems that the individual is authorized to use, as determined by the Enterprise Layer's Presence System Service via a query to the Enterprise Layer's Directory Service.

Therefore, when the Enterprise Layer's Presence System Service receives notification of an individual's presence status change, it publishes the change to the Chat Integration Layer's Presence System Service for each of the observer chat systems.

Upon receipt of an external user's presence status change, the Chat Integration Layer's Presence System Service propagates the status change to the Chat Proxy Service. In this case, the Chat Proxy Services for XMPP and for IRC operate in the same way: both proxies maintain a connection on their chat server for each present and authorized external user (thereby impersonating the external user). Accordingly, the Chat Proxy Service propagates the status along the connection representing the external user.

To complete the picture, the Enterprise Integration Layer's Presence System Service notifies chat systems on the observer list when they are added to the observer list or removed from it. A chat system could be added because the external user logged onto her native chat system or because she was just granted privileges to access that chat system. A chat system could be removed because the external user logged off her native chat system or because she lost her privileges for that chat system. As a result, the bookkeeping at the Chat Integration Layer's Presence System Service is relatively simple.

Finally, for the sake of simplicity, the Logging System RIS for each Rich Service is defined to log all messages that appear on the message bus. Therefore, the message routing tables for all Rich Services cause all messages to route to their Logging System RIS first, and then on to its original destination. An alternative definition, though, might be for the Chat Integration Layer to log only conversation messages (and not presence system messages), and the Enterprise System Layer to log all incoming presence system messages, but not outgoing presence system messages. The alternative definition would require alternative routing tables for the affected Rich Services.

MSCs showing the common interactions are:

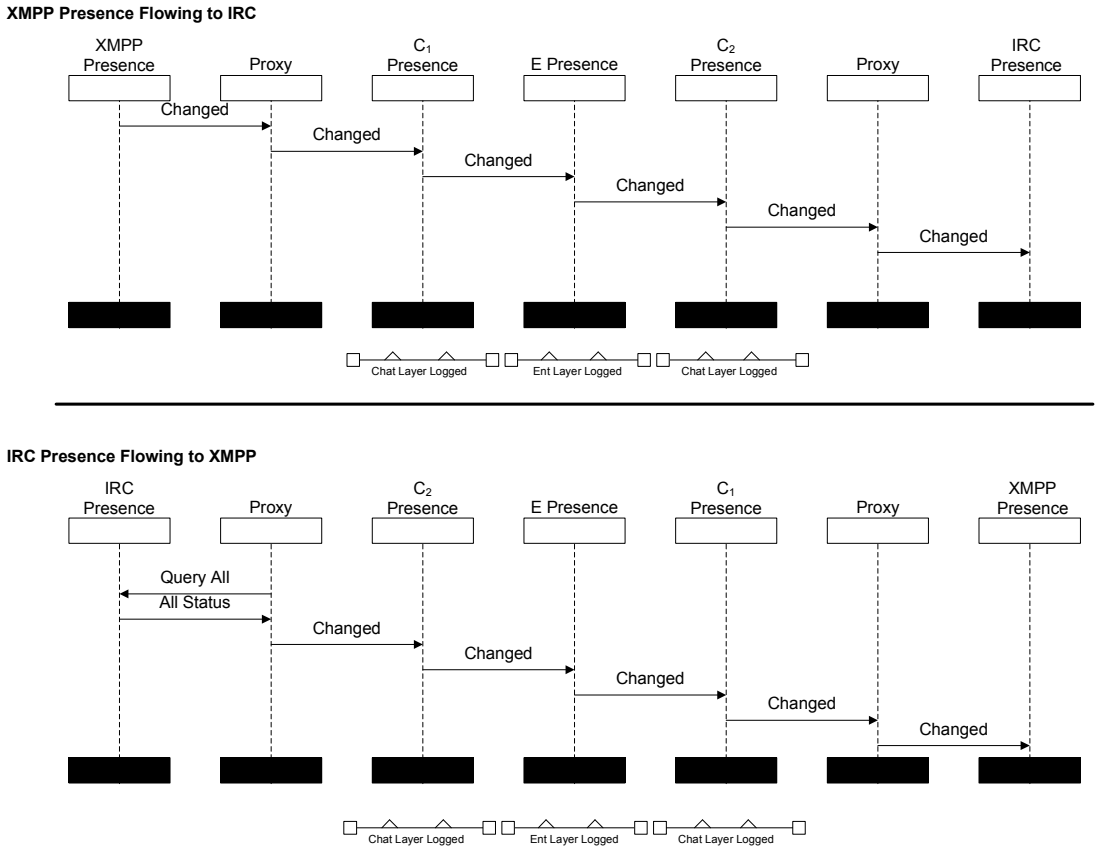


**Figure 30. Message Sequence Charts for Common Interactions**

## 8. Summary and Conclusions

Enterprise Chat is one of the key technologies currently in use to enable rapid communication, decision making, and situational awareness in complex, distributed systems with humans in the loop. In realistic enterprise scenarios, a variety of different chat systems have emerged among the various sub-organizations making up the overall enterprise. Providing an integrated Enterprise chat system under these conditions is a challenging systems-of-systems integration problem.

In this paper, we defined a Composable Chat system based on a Rich Services SOA pattern that enables a service-based hierarchy incorporating existing chat systems into a framework that leverages services (such as authorization, authentication, directory, and presence management) – and does so without risking such systems' existing functionality. To accomplish this, we modeled a conceptual chat system and existing chat systems to discover the roles and interactions that fundamentally define a Composable Chat system. We then identified the roles that could be played by enterprise-level services, and modeled their interaction with corresponding chat system-level roles. The result was a hierarchy that defined a system of systems, incorporating an enterprise service layer and local chat system integration layers. Finally, we organized the hierarchy according to a Rich Service architectural pattern and articulated its benefits, including practical means for organizing and providing additional value-added services and crosscutting processing, subject to policies defined across multiple administrative and organizational domains.

In addition, we presented means by which Rich Messaging could be incorporated into a

Composable Chat system while attending to policy-based distribution rules.

Large enterprises, massively multi-player online games (MMOGs), public safety, and defense are just a short list of application domains that can benefit from a Composable Chat system design along the lines carried out in this case study. Interestingly, the notion of Enterprise Chat itself has the flavor of a much broader range of enterprise integration technologies and approaches – consequently, the fit of emerging enterprise integration technologies, such as Enterprise Service Bus (ESB) technologies, is evident. Our case study identified the Rich Services architecture blueprint as a fitting choice in the conceptualization of the Composable Chat system. In particular, Rich Services bring crosscutting concerns, such as security, policy/governance, and failure management forward, and prescribe a place in the system architecture at which these concerns can be addressed methodologically, as well as from a deployment point of view.

The process we followed in eliciting the core requirements of a Composable Chat system proved useful. In particular, we were able to first conceptualize the notion of chat, independent of the idiosyncrasies of individual chat standards and implementations – this is a prerequisite for finding a common set of features at the interface between disparate standards and systems. The conceptual chat model was then juxtaposed to similar models we have developed for the concrete chat standards and technologies. This juxtaposition provided deeper insight into the capabilities of the existing technologies, and indicated a pathway for decomposing the individual chat capabilities into Rich Infrastructure and Application Services.

Furthermore, the use of both the conceptual chat model, and the Rich Services blueprint allowed us to identify at which places in the architecture changes would be necessary to accommodate advanced features, such as Rich Messaging support.

We believe that the Composable Chat system design offers important insights into accomplishing the integration goals described in Section **Error! Reference source not found.**, and points the way toward a more robust and capable enterprise communication capability. In particular, the Composable Chat system design is sufficiently developed at the conceptual and operational levels; given the short distance between the conceptual and the deployment architecture in the Rich Service blueprint, a working prototype is within reach of a follow-on project.

## 9. Acknowledgements

We are grateful to LorRaine Duffy and B. Scott Jaffa of SPAWAR for their sponsorship of this project and for their kind assistance in helping us to understand the military chat domain. We are also grateful to Frederic Doucet, a PhD student on the SAINT/S3EL team at UCSD, for his help in modeling the IRC, XMPP, and abstract chat systems.

## 10. References

[1] M. Arrott, B. Demchak, V. Ermagan, C. Farcas, E. Farcas, I. H. Krüger, and M. Menarini. *Rich Services: The Integration Piece of the SOA Puzzle*. In Proceedings of the IEEE International Conference on Web Services (ICWS), Salt Lake City, Utah. July 2007.

[2] I. H. Krüger. *Service Specification with MSCs and Roles.* In Proceedings of the IASTED International Conference on Software Engineering (IASTED SE'04), Innsbruck, Austria. 2004.

[3] I. H. Krüger, R. Mathew, M. Meisinger. *Efficient Exploration of Service-Oriented Architectures Using Aspects.* In Proceedings of the 28th International Conference on Software Engineering (ICSE '06), Shanghai, China. May 2006.

[4] V. Ermagan, C. Farcas, E. Farcas, I. H. Krüger, and M. Menarini. *A Service-Oriented Blueprint for COTS Integration: the Hidden Part of the Iceberg.* In Proceedings of the ICSE workshop on Incorporating COTS Software into Software Systems: Tools and Techniques (ICSE '07), Minneapolis, MN. May 2007.

[5] I. H. Krüger. *CSE210: Service-Oriented Software and Systems Engineering & Model-Driven Development.* Lecture at University of California, San Diego. April 21, 2005.

[6] T. Andrews et al. *Business Process Execution Language for Web Services (BPEL4WS) 1.1.* http://www.ibm.com /developerworks/webservices/library/ws-bpel, May 2003.

[7] N. Kavantzas. *Web Services Choreography Description Language Version 1.0.* W3C, 2005.

[8] D. Bryson, D. Winkowski, M. Krutsch, C. Smith, J. Jacobsen, and M. Huss. *XEP-0204: Collaborative Data Objects*. http://www.xmpp.org/extensions/xep-0204.html, Version 0.1, January 2007.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1995.

[10] C. Kalt, *Internet Relay Chat: Architecture.* RFC 2810, April 2000.

[11] C. Kalt, *Internet Relay Chat: Channel Management*. RFC 2811, April 2000.

[12] C. Kalt, *Internet Relay Chat: Client Protocol*. RFC 2812, April 2000.

[13] C. Kalt, *Internet Relay Chat: Server Protocol*. RFC 2813, April 2000.

[14] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition).* Addison-Wesley Professional, 2003.

[15] P. Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Core*. RFC 3920. October 2004.

[16] P. Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*. RFC 3921, October 2004.

[17] J. Hildebrand, P. Millard, R. Eatmon, and P. Saint-Andre. *XEP-0030: Service Discovery*. http://www.xmpp.org/extensions/xep-0030.html, Version 2.3, February 2007.

[18] P. Saint-Andre. *XEP-0045: Multi-User Chat*. http://www.xmpp.org/extensions/xep-0045.html, Version 1.22, April 2007.

[19] World Wide Web Consortium (W3C). *Extensible Markup Language (XML)*. http://www.w3.org/XML/. May 2007.

[20] S. Cantor, J. Kemp, R Philpott, and E. Maler. Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) 2.0 – Errata Composite. http://www.oasis-open.org/committees/download.php/22385/sstc-saml-core-errata-2.0-wd-04-diff.pdf, February 2007.

[21] http://mule.mulesource.org/wiki/display/MULE/Home.

[22] E. Evans. *Domain-Driven Design.* Addison-Wesley, 2003.

[23] http://freemind.sourceforge.net/wiki/index.php/Main_Page. Version 0.8.0, June 2005.

[24] B. Demchak, C. Farcas, E. Farcas, and I. H. Krüger. *The Treasure Map for Rich Services.* In Procedings of the 2007 IEEE International Conference on Information Reuse and Integration (IRI), Las Vegas, NV. August 2007.

[25] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, E. Schooler. *SIP: Session Initiation Protocol*. RFC 3261, June 2002.

[26] A. Niemi, M. Garcia-Martin. *Multi-party Instant Message (IM) Sessions Using the Message Session Relay Protocol (MSRP)*. draft-ietf-simple-chat-00. June 2007.

[27] J. Rosenberg. *SIMPLE made Simple: An Overview of the IETF Specifications for Instant Messaging and Presence using the Session Initiation Protocol (SIP)*. draft-ietf-simple-simple-00. July 2007.

[28] A. B. Roach. *Session Initiation Protocol (SIP)-Specific Event Notification*. RFC 3265. June 2002.

[29] B. Campbell, J. Rosenberg, H. Schulzrinne, C. Huitema, D. Gurle. *Session Initiation Protocol (SIP) Extension for Instant Messaging*. RFC 3428. December 2002.

[30] J. Rosenberg. *A Presence Event Package for the Session Initiation Protocol (SIP)*. RFC 3856. August 2004.

[31] J. Rosenberg. *A Watcher Information Event Template-Package for the Session Initiation Protocol (SIP)*. RFC 3857. August 2004.

[32] J. Peterson. *Common Profile for Presence (CPP)*. RFC 3859. August 2004.

[33] J. Peterson. *Common Profile for Instant Messaging (CPIM)*. RFC 3860. August 2004.

[34] H. Sugano, S. Fujimoto, G. Klyne, A. Bateman, W. Carr, J. *Peterson. Presence Information Data Format (PDIF)*. RFC 3863. August 2004.

[35] A. Neimi. *Session Initiation Protocol (SIP) Extension for Event State Publication*. RFC 3903. October 2004.

[36] H. Schulzrinne. *The tel URI for Telephone Numbers.* RFC 2806. December 2004.

[37] J. Rosenberg. *A Framework for Conferencing with the Session Initiation Protocol (SIP)*. RFC 4353. February 2006.

[38] J. Rosenberg. *A Data Model for Presence*. RFC 4479. July 2006.