Rich Feeds for RESCUE

Barry Demchak
University of California, San Diego – Calit2
bdemchak@ucsd.edu

Ingolf H. Krüger
University of California, San Diego – Calit2
ikrueger@ucsd.edu

ABSTRACT

Effective responses to emergency situations require accessing both information that is known ahead of time and information from emergent sources. Whereas emergency providers have succeeded in formalizing acquisition and distribution of information from pre-existing sources, emergent and unconventional sources remain a challenge, especially in the context of rapid system-of-systems integration.

The Rich Feeds project demonstrates the use of the Rich Services architectural pattern to incorporate data capture facilities into a framework that attends to crosscutting concerns such as authentication, authorization, encryption, and governance. By organizing a system-of-systems, Rich Services fosters the rapid incorporation of novel data sources while promoting scalability, low overall system risk, and fine grained policy definition and evaluation.

In this paper, we demonstrate Rich Feeds' use of Rich Services in accessing multiple data streams during research and disaster drills. Additional opportunities include adding data sources and extraction methods, and increasing flexibility of policy definition and evaluation.

Keywords

Service Oriented Architecture, SOA, Rich Service, ESB, Enterprise Service Bus, COTS, Integration, AJAX

INTRODUCTION

During an emergency, the need for actionable information is often acute, and the kind of information needed is often hard to predict in advance. While emergency providers and the press have demonstrated the use of conventional and prearranged data sources to create a general awareness and enable decision making, much work remains in order to expose and integrate unconventional and emergent data sources for either real time or archival analysis. Furthermore, incorporating such feeds into emergency data systems on short notice (or even under normal circumstances) risks system instability because of the arduous programming mechanics involved in acquiring these feeds, the multiple stakeholder concerns that must be addressed, and the difficulty of scaling such systems so they are performant under high demand.

Rich Feeds is a sub-project of the RESCUE ("Responding to Crises and Unexpected Events") project (Calit2 RESCUE Project, 2007), whose goal is to gather, maintain, leverage, and present information pertinent to an emergency so that emergency response networks and the general public can quickly and accurately understand the situation and the resources available – ultimately to aid in making decisions that save lives, save infrastructure, and reestablish normalcy. Other RESCUE sub-projects include data feeds from the Calit2 Traffic Reporting System (Calit2 RESCUE Traffic, 2007) and the Calit2 Tracked Objects System. (The Traffic Reporting System is a peer-to-peer traffic incident registry, and the Tracked Objects System leverages the cellular phone network to associate position and velocity with objects of interest.)

Rich Feeds is a work in progress that demonstrates how unconventional data feeds (including RESCUE research feeds) and emergent data feeds can be captured, preserved, integrated, and exposed in either real time or after the fact. To achieve this, Rich Feeds overcomes numerous challenges:

- incorporating data from producers not well positioned to support Rich Feeds
- distributing data to consumers without knowing in advance how such data will be analyzed or integrated
- allowing data to be added and accessed subject to policies defined by the producers, consumers, and system operators
- supporting data access by external systems as part of a larger systems framework

To meet these challenges, we based Rich Feeds' design on a Service Oriented Architecture (SOA) pattern called Rich Services (Arrott, M. , Demchak, B., Ermagan, V., Farcas, C., Farcas, E., Krüger, I.H., and Menarini, M., 2007; Demchak, B., Farcas, C., Farcas, E., and Krüger, I. H., 2007), which delivers the benefits of SOA in a system-of-systems framework using a specialized agile development process. Rich Feeds is a hierarchically decomposed system that integrates data producers, data consumers, and data storage and streaming facility into a structure that services crosscutting concerns such as authorization, authentication, and governance flexibly and reliably. Rich Feeds' service oriented architecture allows the addition of new data producers and consumers quickly and with low risk to existing functionality while providing clear paths to high scalability. Its agile development process recognizes the constantly emerging requirements of systems that serve emergency situations and the short timeframes in which stakeholders must be serviced.

This paper describes the challenges involved in using Rich Services and its development process in creating and evolving the Rich Feeds application. It presents the Rich Services architectural pattern and development process as a candidate solution, and it describes and discusses the Rich Feeds prototype that resulted from their use.

The Challenges

Acquiring Data from Producers – Rich Feeds must acquire data from either from other RESCUE sub-projects or from sources that emerge during an emergency. In either case, such data is generally produced and maintained in some form convenient to the producer, and usually without concern for making it available to third parties or applications such as Rich Feeds. Consequently, systems that expose such data must be taken as-is, without expecting accommodations for Rich Feeds. Furthermore, the schema and delivery methods for such data may be changed by the producer based solely on its own needs, and not in coordination with Rich Feeds. The Rich Feeds design meets this challenge by isolating producer systems, and integrating them into a standards-based framework one by one.

Make Data Available to Consumers – Consumers of Rich Feeds data include emergency data systems and researchers seeking new insights. In either case, a consumer must discover the list of feeds available, learn the schema for each feed, and acquire the actual data. Depending on the needs and capabilities of the consumer, Rich Feeds must make the feed list, schema, and data available on either a query basis or an event stream basis. The Rich Feeds design meets this challenge by provided standards-based interfaces (such as HTTP, Web Service, streams, and so on) to match the interface styles likely of interest to consumers.

Enable Policy-driven Access Decisions – Each class of stakeholder has interests that must be addressed and enforced by Rich Feeds. The Rich Feeds design meets this challenge by formulating such interests as policies, which must be applied in particular circumstances so as to enable or constrain services delivered to other stakeholders. Each stakeholder is free to create such policies independently of other stakeholders, and the Rich Feeds design enforces these policies. For example, a Producer may have a policy that its data must be accessible only to particular Consumers, and then only at a particular resolution. A Consumer may have a policy that data it receives must have been produced only by a particular experimenter. An Operator may have a policy that a particular Consumer can obtain a particular data rate, or that all data accesses are logged for auditing and accounting purposes.

Enable Interaction with External Systems – While Rich Feeds must be able to make data available to external systems, it must also cooperate with such systems in workflow and choreography operations. These systems may query and set policy, create new feeds as composites and transformations of existing feeds, and query operational data. The Rich Feeds design addresses this challenge by exposing precise and well-defined interfaces which it services using encapsulated services and components.

Taken together, these challenges imply other important requirements for Rich Feeds:

Authentication and Authorization – In order to enforce most types of policy, the identity of actors must be reliably and unambiguously established. Furthermore, a policy model must be established wherein policies can be defined and evaluated independently of any particular Producer, Consumer, or feed. The model must account for an actor's identity and its capabilities in executing a particular operation in a particular context.

Concerns Clearly Separated – Implicit in all of the challenges listed above is the need to make changes to Rich Feeds quickly, cheaply, reliably, and without introducing processing errors in already-stable code. Similarly, crosscutting concerns such as authentication, authorization, policy management, encryption, and system management must each be independently modifiable and retargetable so as to achieve a high level of flexibility at low maintenance risk. Rich Feeds must be designed to keep separate concerns separated at all levels of the architecture and code.

A SERVICE-ORIENTED APPROACH

Our approach to the design of the Rich Feeds system involves the characterization and analysis of the provider, consumer, and operations services in a Service Oriented Architecture (SOA).

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 by leveraging existing technologies to create standards-based interactions (such as HTTP/SOAP, XML, WSDL, and UDDI) 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.

In Rich Feeds, our use of these terms is more basic and generic – a *service* is defined as a choreography of interactions between entities (Krüger, I. H., 2004), and can be discussed independently of the particular technologies used to implement it (Krüger, I. H., Mathew, R., Meisinger, M., 2006). A service description focuses on well-defined roles played by the entities, and the interactions between them. 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 be a model of either the logical level, the deployment level, or both.

At the logical level, a SOA models roles and interactions independently of how they are implemented or deployed. It 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. Services can be composed of other services, can act as proxies for other services, or can stub out services as needed. Service-oriented analysis can lead to the separation of concerns and the identification and modeling of crosscutting concerns, thus contributing to long term system reliability, extensibility, and maintainability. In the Rich Feeds context, examples of crosscutting concerns include authorization, authentication, encryption, and logging.

At the deployment level, a SOA models services as interactions between loosely coupled components that implement the roles modeled at the logical level. Such components are self-contained, thus encouraging component-level interoperability and adherence to standards. In Rich Feeds, communications between components is carried out via messages – the combination of messaging and standards compliant self-contained components enables a great deal of flexibility in deploying components and managing their interactions. Similar to the logical level, components can be composed of other components, can modify information flow by intercepting messages exchanged between components, and can stub out component processing as needed.

At both the logical level and the deployment level, SOAs provide value by encouraging manageability, scalability, dependability, testability, malleability, interoperability, composition, and incremental development.

Rich Services

We based the Rich Feeds design on a Rich Services architectural pattern and development model. The architectural pattern leverages the composite pattern (Gamma, E., Helm, R., Johnson, R., and Vlissides, J., 1995) and the messaging and routing patterns (Hohpe, G. and Woolf, B., 2004) to create a system of systems. The development model provides a means whereby application requirements can be factored into roles and the services that involve them, thereby leading to both logical level and deployment level SOAs. The Rich Services model creates an efficient and effective process by helping to identify crosscutting concerns early in development, and by allowing agile-style iteration at every stage.

We modeled Rich Feeds as a system composed of three kinds of systems: the Provider systems, the Consumer systems, and the Integration system. The Provider systems are provided and controlled by experimenters or emergent providers – they collect data and expose it. The Consumer systems are provided and controlled by researchers and emergency systems – they analyze and visualize data. The Integration layer consists of a database, interfaces to the Provider and Consumer systems, and processing for crosscutting concerns.

A Rich Service architecture organizes systems-of-systems into a hierarchically decomposed structure that supports both "horizontal" and "vertical" service integration (see Figure 1). 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.

There are three main entities in a Rich Service architecture: Rich Services, Messengers and the Router/Interceptors, and Service/Data Connectors. A Rich Service encapsulates the various application and infrastructure functionalities, which may themselves be decomposed as Rich Services. The Messenger and the Router/Interceptor together form the internal communication infrastructure linking Rich Services. The Service/Data Connector serves as the sole mechanism for interaction between a Rich Service and its environment.

A Rich Service could be a simple functionality block such as a Commercial Off The Shelf (COTS) system (Ermagan, V., Farcas, C., Farcas, E., Krüger, I. H., and Menarini, M., 2007) or a Web service, or it could be hierarchically decomposed. We distinguish between two kinds of Rich Services: *Rich Application Services* (RAS) and *Rich Infrastructure Services* (RIS). RASs interface directly with the Messenger, and provide core application functionality such as data collection and database access. RISs interface directly with the Router/Interceptor, and provide infrastructure and crosscutting functionality such as logging, authentication, and authorization.

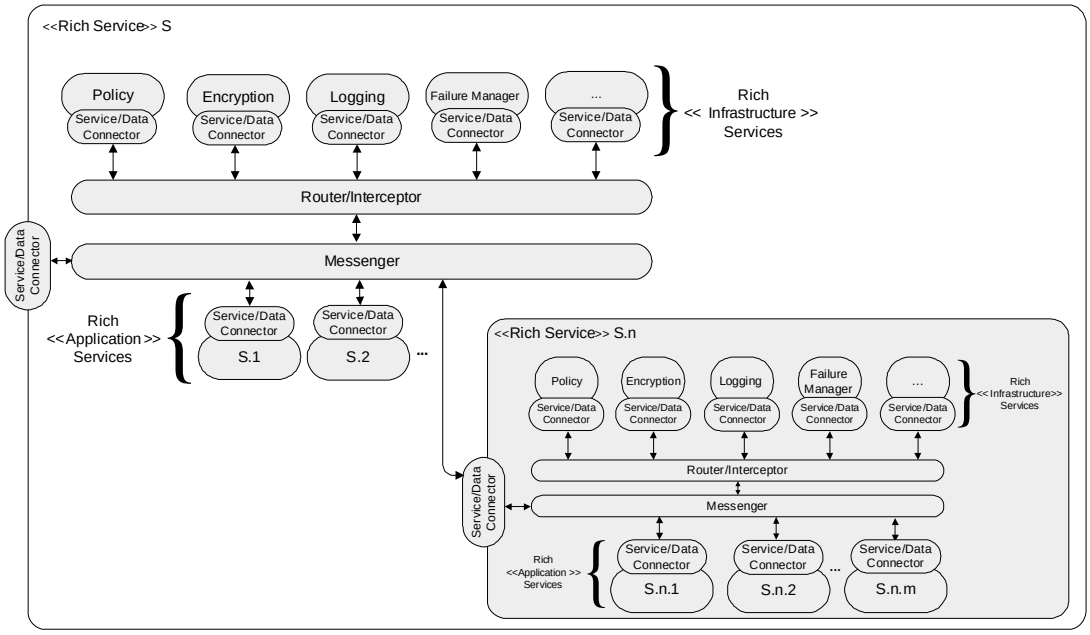

Figure 1. Rich Services Architectural Pattern

The Messenger and Router/Interceptor support horizontal integration by routing messages between Rich Services. The *Messenger* layer is responsible for message transmission between services – by providing the means for asynchronous messaging, the Messenger supports decoupling of services. The *Router/Interceptor* intercepts messages carried by the Messenger and reroutes them. Leveraging the interceptor pattern (Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F., 2000) facilitates dynamic behavior injection based on the interactions among services. This is useful for the injection of policies (such as crosscutting behaviors) governing the integration of a set of horizontally decomposed services.

The Service/Data Connector is the means by which Rich Services are connected to an external communication infrastructure. It encapsulates and hides the internal structure of the connected Rich Service, and exports only the interfaces that the connected Rich Service intends to provide and expose. This encapsulation helps tackle the vertical integration challenge introduced by systems-of-systems by precisely identifying the interfaces contributed by a Rich Service.

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 differences.

Use of Rich Services in Rich Feeds

As the number of stakeholders of a SOA project grows, so typically does the number and complexity of various business concerns (such as governance, security, and policy) and their level of distribution across the architecture. We chose to model Rich Feeds on a Rich Services architecture in anticipation of this complexity – Rich Services provides a framework that decouples these concerns, thereby promoting scalability, extensibility, maintainability, and reliability. Additionally, the flexibility of Rich Services' horizontal and vertical decomposition allows the alignment of the Rich Feeds architecture to the relationships between the stakeholders, thereby promoting understandability and maintainability.

At the logical level, the Rich Feeds architecture models the relationship between the Integration system, the Producer systems, and the Consumer systems. As shown in Figure 2, at the core of the Integration system is a database that stores data acquired from the Producer systems and makes it available to the Consumer systems. While the Producer systems are modeled as services from the perspective of the Integration system, they are actually RASs decomposed into adapters and the data acquisition systems provided by the experimenters and emergency networks. Similarly, while the Consumer systems are modeled as services, they are actually RASs decomposed into adapters and analysis/visualization systems.

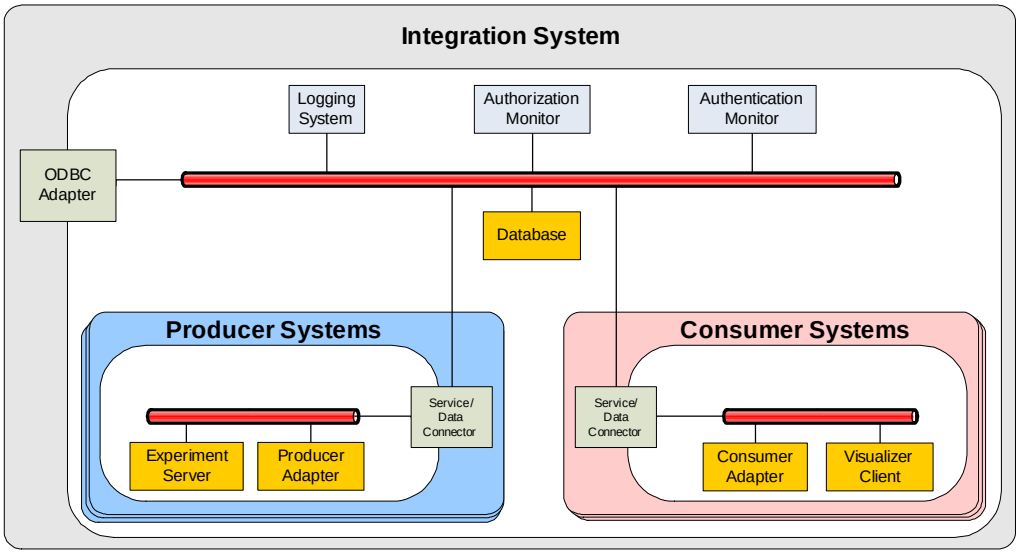

Figure 2. Rich Feeds Conceptual Architecture

Within the Integration system, infrastructure services implement crosscutting concerns such as authentication, authorization, and logging. Regardless of their source, messages sent to the Database service (for storing or retrieving data) are passed amongst infrastructure services according to a routing defined at application configuration time. In this case, messages are intercepted first by the authentication RIS, then by the authorization RIS, and finally by the logging RIS. By separating concerns in this way, the database RAS need not be security-aware, and the RISs need not be database-aware.

The Integration system's Service/Data Connector defines access to Rich Feeds from outside clients, and consists of a standards-based Open Database Connector interface.

At the deployment level, the Rich Feeds architecture demonstrates the relationship between the Integration system and the Producer and Consumer systems, an example of which is shown in Figure 3. The Messenger and Router/Interceptor portion of the Integration system is implemented by ActiveMQ and version 1.4.3 of the Mule Enterprise Service Bus (ESB) (MuleSource, 2007). ActiveMQ transports the messages, and Mule determines the RASs and RISs to which the messages are routed. Under Mule, messages are routed to code modules (called UMOs)

written in Java and called Plain Old Java Objects (POJOs). The MySQL database COTS is interfaced to the ESB using such a POJO. The Producer and Consumer systems are implemented by a combination of a POJO, an Internet-facing application adapter, the Internet itself, and experiment and emergency network servers or client visualizers. The Integration system RISs are implemented as POJOs that intercept messages passed to the database POJO. (The particular routing of messages between RAS and RIS POJOs is set up at Mule configuration time.)

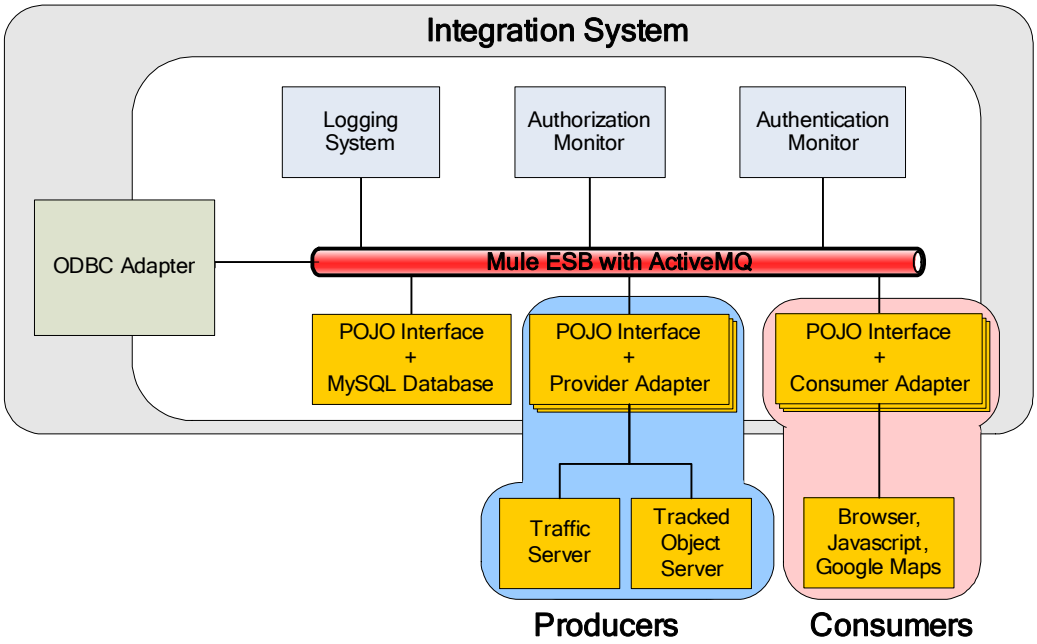

Figure 3. Rich Feeds Deployment Architecture

In this diagram, the Integration System consists of the ODBC Adapter (Microsoft Corporation, 2007), the Database RAS, and RISs for authentication, authorization, and logging. The Database RAS consists of the POJO/Database pair, which accepts messages containing queries and returns messages containing data or an error code. All such messages have an internally defined XML format, which is also processed by the ODBC Adapter and the RISs.

There are two producers: the Calit2 Traffic Reporting System and the Calit2 Tracked Objects System. For each producers, there is a POJO/Adapter pair that queries the provider server for new data, and then sends new data to the POJO/Database pair. Both producers make their feeds available in proprietary formats: RSS for the Traffic system, and a proprietary format for the Tracked Objects system. The POJO/Adapters reformulate the data into the POJO/Database's XML format, and then insert the XML document onto the ESB, destined for the POJO/Database pair.

Similarly, there is one consumer: the Rich Feeds browser-based AJAX application. The AJAX application queries its POJO/Adapter pair to determine the list of available feeds and their schemas, and then to download data. The AJAX application uses HTTP to submit an SQL query, and the POJO/Adapter pair acts as an HTTP server – it reformulates the query into the POJO/Database's XML format and inserts it onto the ESB, destined for the POJO/Database pair.

Along the way, Mule routes the query message to the Authentication Monitor, the Authorization Monitor, and then the Logging System RIS. In the Authentication Monitor, if the message contains valid credentials, the credentials are replaced by a list of credential-specific capabilities, and then the message is reinserted onto the ESB – otherwise, the message is returned to the sender along with an error status. Next, in the Authorization Monitor, if the message's query matches the credentials, the credentials are removed from the message, and then the message is reinserted onto the ESB – otherwise, the message is returned to the sender along with an error status. Next, the message is logged by the Logging System, is reinserted onto the ESB, and then arrives at the POJO/Database RAS.

Development Methodology

Traditional component-oriented software development focuses on defining component blocks and the interactions between them, leaving crosscutting concerns to be layered in toward the end of the design process. This results in retrofitting and backtracking as accommodations to these concerns. In contrast, we used the Rich Services development process (Demchak, et al, 2007) because it focuses on requirements, entities, and interactions, thereby resulting in early discovery and integration of crosscutting concerns. Additionally, this process produces a logical architecture separate from a deployment architecture, and a mapping between them. Consequently, an investment in a logical architecture can be leveraged into a number of deployment options, depending on the availability and configuration of computing resources.

As shown in Figure 4, the Rich Services development process is an iterative process having three phases, each with one or more stage. Each phase or stage can be iterated a number of times in order to achieve an implementation that satisfies an application's requirements. While this approach applies well to initial system design, it applies equally well when requirements are changed, as is often the case when adapting to emergent scenarios – an iteration can begin at any phase or stage, and can be worked through to achieve an appropriate logical or deployment architecture.

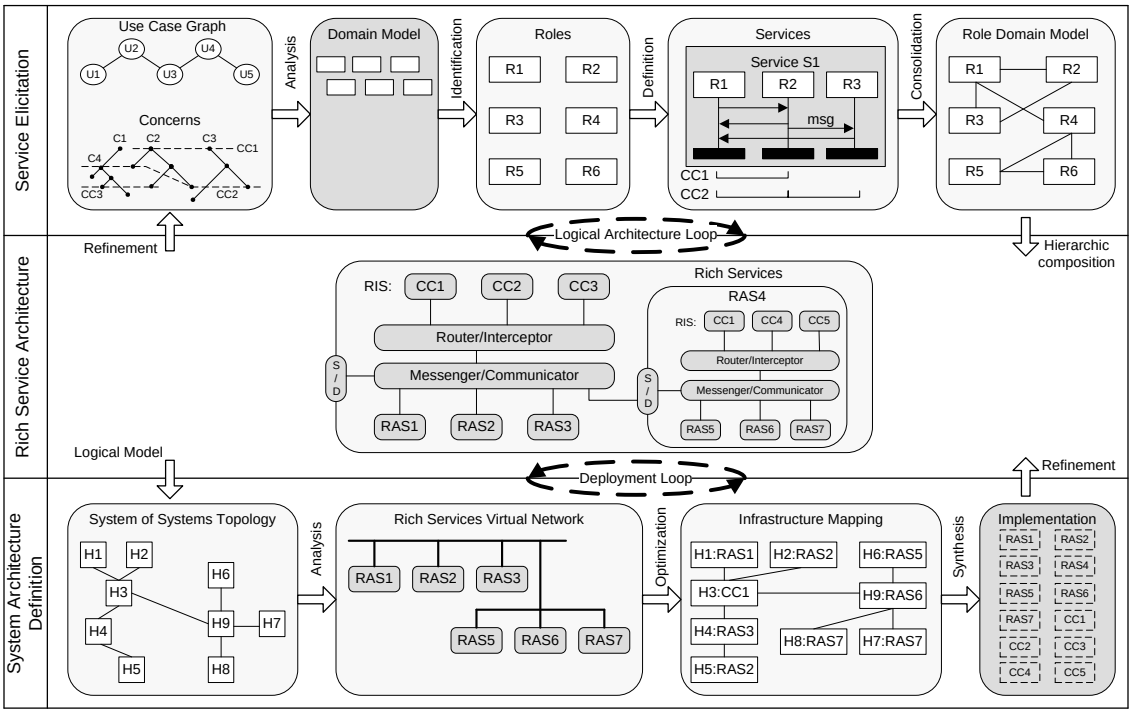

Figure 4. Rich Services Development Process

In the Service Elicitation phase, system requirements are captured and analyzed, resulting in a role domain model and a service repository. The Rich Service Architecture phase defines a hierarchic set of Rich Services as a logical model incorporating the role domain model and the service repository. The System Architecture Definition phase establishes the relationship between the logical model and the deployment model.

A RICH FEEDS PROTOTYPE

We created a working prototype of the Rich Feeds system on the RESCUE server at Calit2 (Calit2 RESCUE Rich Feeds, 2007), and it was used to improve situational awareness during an active shooter drill at the University of California, San Diego (UCSD) and the San Diego firestorms, both in October, 2007. For the active shooter drill, it showed the position and video feed associated with a mobile robot, and for the firestorms, it integrated traffic incident positions with geographically diverse live feed video cameras.

The producers include those shown in Figure 3 in addition to live camera feeds provided by the High Performance Wireless Research and Education Network (HPWREN) (Baker, J., 2007) and the UCSD Police Department. Each

producer contributes data unique to its experiment or data source, with each data element being tagged with a producer -determined timestamp and GPS location. For example, a Traffic System data element includes the time of a traffic incident report, the GPS location of the incident, the identity of the person making the report, and a reference to an audio clip of the actual incident report. A Tracked Objects System data element includes the time of the report, the GPS location of an object, the speed and direction of the object, and the object's name and alias.

The consumer is a browser-based AJAX application (Figure 5) that queries the Rich Feeds server to determine the list of feeds it hosts, and then makes the feeds available for a user to select via a browser form. Along with the feed list, the server provides the schema for each feed, including profiling information that allows the user to create filters on each feed. Periodically, the AJAX application queries the server to acquire data that matches the filters for each feed selected by the user. Any data returned by the server is plotted on a Google Map, with each feed being represented by a different style of marker. By clicking on a marker, the user can drill down on individual data elements to see data element details, including playing audio or video clips associated with the data element. In animation mode, the application plays a movie of markers appearing or moving on the display within a time interval specified by the user. The result is that by spatially juxtaposing multiple feeds or by animating them, the user can see the geographical and temporal relationship between data available from multiple sources.

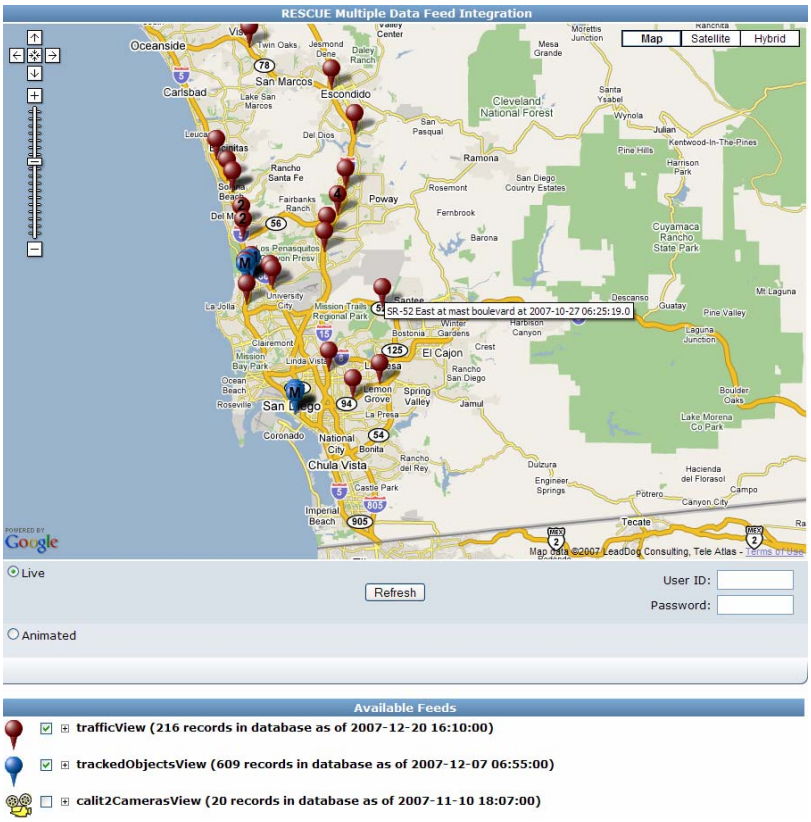

Figure 5. Rich Feeds Browser Showing Two Feeds in San Diego, California

The User ID and Password entered by the user determines the list of feeds returned by the server. The policy defined at the server is to return three feeds if the user enters no credentials. If the user enters police credentials, the server returns the three feeds in addition to the police camera feed.

DISCUSSION

Within a Rich Services framework, there is room for numerous strategic decisions based on business objectives. For example, the architecture shown in Figure 3 is not the only one that can fit the Rich Feeds requirements and achieve the goals of separated concerns and loosely coupled crosscutting processing. We could have chosen a similar architecture, which would have the Producer and Consumer systems accessing the database POJO through the Integration system's Service/Data Connector instead of directly across the ESB. This would have required that the Producer and Consumer systems adhere to the services defined at the Connector, or that services compliant with the Producer and Consumer systems be exposed through the Connector. We chose our architecture for efficiency reasons: our Producer and Consumer modules create non-standards-based messages directly consumable by the database POJO. To match this using a Service/Data Connector path would require that standards-based messages arriving at or leaving from the Connector undergo transformation to and from the internal message format. Alternately, we could have chosen to expose the internal message interface at the Service/Data Connector, and then

implemented adapters outside of the Rich Service. There is no advantage to this if we could avoid publishing these interfaces by encapsulating them within the Integration system.

Much of the functionality of Rich Feeds could be achieved in a database-centric application framework such as Oracle's Application Development Framework (Oracle Corporation, 2007), and we could have designed Rich Feeds along these lines. However, a Rich Services framework was chosen because it offers an end-to-end development process, and architecture-level distribution of services across several computing platforms, support for parallel execution (when appropriate), and flexible exploitation of COTS.

Additionally, the Rich Feeds system could have been implemented purely as an orchestration (Janssen, M, Gortmaker, J., & Wagenaar, R.W., 2006) involving the Producer, Consumer, and database RASs. However, this would have required explicitly orchestrating the crosscutting concerns implemented in the authentication, authorization, and logging RISs. This explicit orchestration would have been more complex and less maintainable than the router-based interceptor method inherent in the Rich Services pattern. It is possible that crosscutting concerns could have been gracefully intermixed with orchestration in an AO4BPEL implementation (Charfi, A. and Mezini, M., 2007), but that possibility was not explored.

OPPORTUNITIES

Under the RESCUE award, we continue to evolve Rich Feeds in order to enable more flexible access to realtime and archival data. We have opportunities to incorporate Consumers such as Google Earth, Yahoo Pipes, and Microsoft Office. To support Yahoo Pipes, we would add a streaming interface to the Integration system's Service/Data Connector. To support Microsoft Office, we would realize the ODBC interface in the Service/Data Connector.

We hope to further develop the authentication and authorization RISs so as to demonstrate more flexible routing policies within the Mule ESB and ActiveMQ systems. Currently, for the sake of demonstration of policy evaluation within the RISs, the password and capabilities databases exist and are hard-coded within the authorization and authentication RISs themselves, and then are used within Rich Feeds' existing policy model. We hope to move them to a separate database, thereby leveraging separate policy definition facilities that can be employed quickly and easily during an emergency.

Currently, no data is being exchanged under encryption. Opportunities exist to secure links between the Integration system Rich Service and the Consumer and Procedure Rich Services.

RESULTS AND CONCLUSION

The Rich Feeds system was initially demonstrated and used in the Emergency Operations Center (EOC) at an active shooter drill in October, 2007 at the University of California, San Diego. In addition to showing the location of various assets, it demonstrated a live camera feed generated from a remote control robot traversing the crime scene. The system was generally well received.

The success of the Rich Feeds system is a demonstration that a Rich Services architecture can be used to integrate data from diverse sources in order to enable discovery by diverse stakeholders either in real time or archivally.

Equally important is the ability to evolve Rich Feeds quickly, reliably, and without risking code breakage. To that end, we demonstrated the creation of Producer and Consumer components for a new feed in less than two hours, and without breaking existing Producer and Consumer functionality. (We added the HPWREN camera feeds in order to support discovery during the San Diego Firestorm of October, 2007.)

Additionally, while early versions of Rich Feeds did not implement authentication and authorization, we added these features in less than a day. Besides creating the infrastructure services for these concerns, only the message routing and the AJAX application were changed – there was no impact on any of the data producers or on any other Rich Services.

We take these experiences as corroboration of our premise that the separation of concerns possible in a Rich Services architecture enables rapid and reliable integration of new data sources, which can be reprised on demand during emergencies in order to achieve situational awareness. The Rich Services development process allowed the new domain and crosscutting requirements to be leveraged into the existing logical architecture smoothly and expeditiously, while giving high confidence that existing capabilities would not be compromised.

While the existing AJAX interface enables the location- and time-based integration of data, numerous other emergency and research scenarios remain to be explored. The Rich Services architecture and development process provide Rich Feeds with a solid basis on which to expand access to more data sources, improve accessibility, address more complex stakeholder concerns, and to do so quickly and reliably.

ACKNOWLEDGMENTS

Our work was partially supported by the NSF within the projects "RESCUE" (award #03311690), "Responsphere" (award #0403433), and "ASOSA: Automotive Service-Oriented Software and Systems Engineering" (award #CCF0702791), as well as by funds from the California Institute for Telecommunications and Information Technology (Calit2). We are also grateful for the helpful comments of our anonymous reviewers.

REFERENCES

1. Arrott, M. , Demchak, B., Ermagan, V., Farcas, C., Farcas, E., Krüger, I.H., and Menarini, M. (2007) Rich Services: The Integration Piece of the SOA Puzzle. *Proceedings of the IEEE International Conference on Web Services (ICWS)*, Salt Lake City, Utah.
2. Baker, J. (2002) The High Performance Wireless Research and Education Network: An Overview. http://hpwren.ucsd.edu/info/images/baker.doc.
3. Calit2 RESCUE Project (2007) http://www.itr-rescue.org.
4. Calit2 RESCUE Rich Feeds (2007) *http://rescue.calit2.net*.
5. Calit2 RESCUE Traffic (2007) *http://traffic.calit2.net*.
6. Charfi, A. and Mezini, M. (2007) AO4BPEL: An Aspect-oriented Extension to BPEL. Springer Netherlands. World Wide Web Vol. 10, No. 3, pp. 309-344.
7. Demchak, B., Farcas, C., Farcas, E., and Krüger, I. H. (2007) The Treasure Map for Rich Services. *Proceedings of the 2007 IEEE International Conference on Information Reuse and Integration (IRI)*, Las Vegas, NV.
8. Ermagan, V., Farcas, C., Farcas, E., Krüger, I. H., and Menarini, M. (2007) A Service-Oriented Blueprint for COTS Integration: the Hidden Part of the Iceberg. *Proceedings of the ICSE workshop on Incorporating COTS Software into Software Systems: Tools and Techniques (ICSE '07)*, Minneapolis, MN.
9. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.
10. Hohpe, G. and Woolf, B. (2004) Enterprise Integration Patterns: Designing, Building, and Deploying. Addison-Wesley Professional.
11. Janssen, M., Gortmaker, J., & Wagenaar, R.W. (2006) Web Service Orchestration in Public Administration: Challenges, Roles and Growth Stages. Information Systems Management Vol. 23, No. 2, pp. 44-55.
12. Krüger, I. H. (2004) Service Specification with MSCs and Roles. Proceedings of the IASTED International Conference on Software Engineering (IASTED SE'04), Innsbruck, Austria.
13. Krüger, I. H., Mathew, R., Meisinger, M. (2006) Efficient Exploration of Service-Oriented Architectures Using Aspects. *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*, Shanghai, China.
14. Microsoft Corporation (2007) *http://support.microsoft.com/kb/110093*.
15. MuleSource (2007) *http://mule.mulesource.org/wiki/display/MULE/Home*.
16. Oracle Corporation (2007) *http://www.oracle.com/technology/products/adf/index.html*.
17. Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F. (2000) Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. Wiley & Sons.