The following paper was originally published in the
Proceedings of the USENIX Symposium on Internet Technologies and Systems
Monterey, California, December 1997

Salamander: A Push-based Distribution Substrate for Internet Applications

G. Robert Malan, Farnam Jahanian, and Sushila Subramanian
University of Michigan

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

Salamander: A Push-based Distribution Substrate for Internet Applications *

G. Robert Malan, Farnam Jahanian

Department of EECS
University of Michigan
1301 Beal Ave.
Ann Arbor, Michigan 48109-2122
{rmalan,farnam}@eecs.umich.edu

Sushila Subramanian

School of Information
University of Michigan
550 East University Ave
Ann Arbor, Michigan 48109-1092
sushila@umich.edu

Abstract

The Salamander distribution system is a wide-area network data dissemination substrate that has been used daily for over a year by several groupware and webcasting Internet applications. Specifically, Salamander is designed to support push-based applications and provides a variety of delivery semantics. These semantics range from basic data delivery, used by the Internet Performance Measurement and Analysis (IPMA) project, to collaborative group communication used by the Upper Atmospheric Research Collaboratory (UARC) project. The Salamander substrate is designed to accommodate the large variation in Internet connectivity and client resources through the use of application-specific plug-in modules. These modules provide a means for placing application code throughout the distribution network, thereby allowing the application to respond to network and processor resource constraints near their bottlenecks. The delivery substrate can be tailored by an application for use with a heterogeneous set of clients. For example the IPMA and UARC projects send and receive data from: Java applets and applications; Perl, C and C++ applications; and Unix and Windows 95/NT clients. This paper illustrates the architecture and design of the Salamander system driven by the needs of its set of current applications. The main architectural features described include: the data distribution mechanism, persistent data queries, negotiated push-technology, resource announcement and discovery, and support for Application-level Quality of Service policies.

*This work is supported by the National Science Foundation under the cooperative agreement IRI-92-16848 and a generous gift from the Intel Corporation.

1 Introduction

The availability of ubiquitous network connections in conjunction with significant advances in hardware and software technologies have led to the emergence of a new class of distributed applications. The Salamander data distribution substrate provides the systems support needed for two of these applications: groupware and webcasting. Participants in these wide-area distributed applications vary in their hardware resources, software support and quality of connectivity[11]. In an environment such as the Internet they are connected by network links with highly variable bandwidth, latency, and loss characteristics. In fact, the explosive growth of the Internet and the proliferation of intelligent devices is widening an already large gap between these members. These conditions make it difficult to provide a level of service that is appropriate for every member of a collaboratory or webcasting receiver.

Webcasting applications use push technology to send data from network servers to client machines. In group collaboratories, people from a global pool of participants come together to perform work or take part in meetings without regard to geographic location. A distributed collaboratory provides: (1) human-to-human communications and shared software tools and workspaces; (2) synchronized group access to a network of data and information sources; and (3) remote access and control of instruments for data acquisition. A collaboratory software environment includes tools such as whiteboards, electronic notebooks, chat boxes, multi-party data acquisition and visualization software, synchronized information browsers, and video conferencing to facilitate effective interaction between dispersed participants. A key challenge for the designers of wide-area collaboratories is the creation of scalable distribution

and dissemination mechanisms for the shared data.

The Salamander substrate provides support for both webcasting and groupware applications by providing virtual distribution channels in an attribute-based data space. In a Salamander system, a tree of distribution nodes (servers) can be dynamically constructed to provide points of service into the data space. Clients can connect to this tree to both publish and subscribe to data channels. The data is *pushed* from suppliers to the clients through the distribution tree. Opaque data objects are constructed by clients that are described using text-based attribute lists. Clients provide persistent queries to the Salamander substrate using attribute expressions that represent the data flows they wish to receive, thereby subscribing to a virtual data channel. Salamander connections are first-class objects and are addressable if desired. This addressability allows for feedback from subscribers to data publishers. Additionally, Salamander allows for plug-in modules at any point in the distribution tree for application code to affect the data distribution. These plug-in modules provide the mechanism to support application-level Quality of Service policies.

Although the terminology is relatively new, Push technologies have been around for many years. The seminal push technology is electronic mail . Email has been pushed from publisher to subscribers for decades through mailing lists. Moreover, USENET news[9] has been used to push data objects (articles) based on text attributes (group names). Extending netnews, the SIFT tool[18] has been used to redistribute netnews articles based on text-based user profiles. At a lower level the combination of native IP multicast support and specific Mbone [5] routers provides a mechanism for the push of IP datagrams throughout portions of the Internet. Multicast datagrams are pushed based on a single attribute, namely the IP multicast group address.

Recently, many commercial push-based companies have started: BackWeb, IFusion, InCommon, Intermind, Marimba, NETdelivery, PointCast, and Wayfarer. These commercial ventures promise to manage the complexity of the web by providing data to users by means of subscription; similar to what current mailing lists provide, only more intrusively. In fact, many of these products are really *poll and pull* instead of push. The clients in these systems periodically poll the servers for new data, and then fetch it if it is available, reducing scalability.

The Salamander substrate differs from past technologies in several ways. First, it is not user-centric, but application-centric. Salamander is an underlying substrate that applications can plug into to transparently connect different portions of a wide-area application. This connectivity is achieved through the use of a channel subscription service. Second, Salamander allows for the addition of application plug-in modules along the distribution path to allow for a variety of data modifications and delivery decisions. The remainder of the paper will further describe the Salamander substrate. Section 2 provides background material on Salamander's current applications. Section 3 enumerates the main architectural features. Section 4 provides an overview of the Salamander application programming interface (API) as well as describes its administration and security features. Section 5 gives both a qualitative and quantitative performance evaluation. Finally, in Section 6 we conclude and describe our current and future research interests.

2 Application Domain

The Salamander substrate currently supports two applications with diverse needs: the UARC and IPMA projects. The Upper Atmospheric Research Collaboratory (UARC)[3, 17] is a distributed scientific collaboratory over the Internet. The UARC project is a multi-institution research effort, whose focus is the creation of an experimental testbed for wide-area scientific collaboratory work. The UARC system provides a collaboratory environment in which a geographically dispersed community of space scientists perform real-time experiments at a remote facilities, in locations such as Greenland, Puerto Rico, and Alaska. Essentially, the UARC project enables this group to conduct team science without ever leaving their home institutions. These scientists perform experiments on remote instruments, evaluate their work, and discuss the experimental results in real-time over the Internet. This community of space scientists has extensively used the UARC system for over three years; during the winter months, a UARC campaign – the scientists use the term *campaign* to denote one of their experiments – takes place almost every day. This community has grown to include regular users from such geographically diverse sites as: SRI International in Menlo Park, California; the Southwest Research Institute; the Danish Meteorological Institute; the Universities of Alaska, Maryland, and Michigan; and the Lockheed Palo Alto Research Laboratory.

The UARC system provides a variety of services to its users including shared synchronized displays for instrument data, multiparty chat boxes, a shared annotation database, and a distributed text editor.

However, the primary mechanism for collaboration is the real-time distribution of atmospheric data to the experiment's participants. This data is collected at remote sites such as Kangerlussuaq, Greenland, and is distributed over the Internet to the scientific collaboratory using the Salamander substrate described in this paper. Figure 1 shows several different data feeds displayed during a real-time campaign.

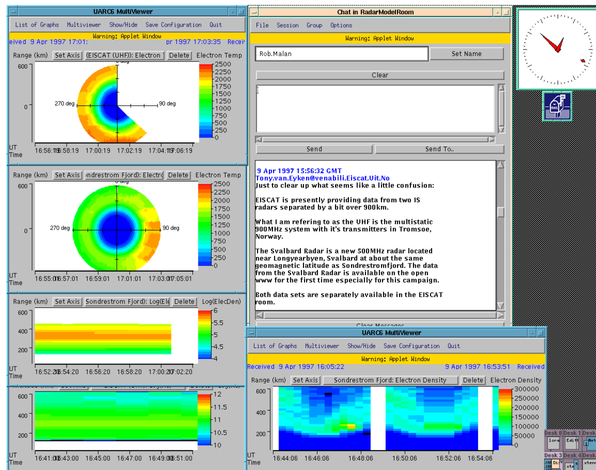

Figure 1: Example screen grab from April 1997 UARC campaign.

The second project that uses Salamander is the Internet Performance Measurement and Analysis (IPMA) project[8], a joint effort of the University of Michigan and Merit Network. The IPMA project collects a variety of network and interdomain performance and routing statistics at Internet Exchange Points (IXPs), internal ISP backbones, and campus LAN/WAN borders. A key objective of the IPMA project has been to develop and deploy tools for real-time measurement, analysis, dissemination and visualization of performance statistics. Two major tools from the IPMA projects are ASExplorer and NetNow. They both use Salamander to webcast the real-time data from the IPMA Web servers to their connected Java applets. ASExplorer is intended to explore real-time autonomous system (AS) routing topology and instability in the Internet. It supports measurement and analysis of interdomain routing statistics, including: route flap, growth of routing tables, network topology, invalid routing announcements, characterization of network growth and stability. An example of the ASExplorer client is shown in Figure 2. NetNow is tool for measuring network loss and latency. The NetNow daemon collects a variety of loss and latency statistics between network peers. The NetNow client is a Java applet which provides a graphical look at real-time conditions across an instrumented network.

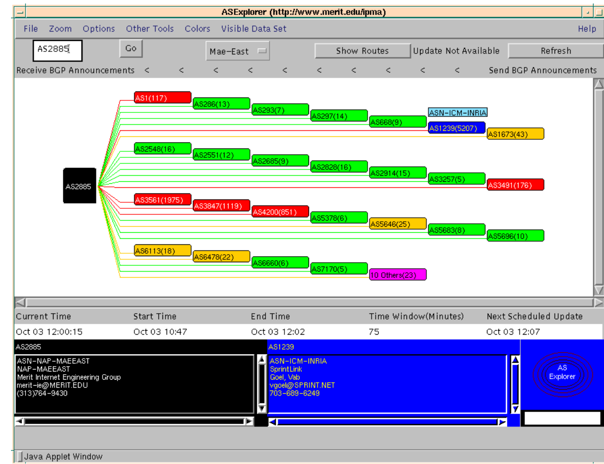

Figure 2: Example screen grab from ASExplorer session.

3 Architecture

The Salamander substrate's architecture can be described in terms of its coarse-grained processes, channel subscription interfaces, and distribution semantics. The key contributions of the architecture are its:

- **Channel Subscription Service**: The Salamander substrate uses a publish/subscribe service that combines the strength of database retrieval with a dynamic distribution mechanism. This service is used to provide a continuous flow of data from publishers to subscribers.

- **Application-level Quality of Service**: Application-level quality of service policies are supported that provide the ability to adapt the delivery channels in response to changes in client and network resources and subscriptions. These policies are supported by the use of application specific plug-in modules that can be used for data flow manipulation.

- **Lightweight Data Persistence**: Salamander employs the use of a caching and archival mechanism to provide the basis for high-level message orderings. A two-tiered cache keeps current data in memory while migrating older data to permanent storage. This storage takes the form of a lightweight temporal database tailored to Salamander's needs.

A Salamander-based system is composed from two basic units: *servers* that act as distribution points and are usually collocated with Web servers; and *clients* that act as both data publishers and subscribers. These units can be connected together in arbitrary topologies to best support a given application (see Figure 3 for an example). The Salamander server is designed from a utilitarian perspective, in that it can stand alone, or like a software backplane can be multiplied to increase scalability. The current version of the server is a POSIX thread implementation on Solaris. Salamander clients can both publish and subscribe to virtual data channels. In both the IPMA and UARC projects, the main data suppliers are written in either Perl or C; whereas the mainstay of the subscribers are Java applets. Applet development has progressed for over a year and a half on the UARC project, and Web browsers are the *de facto* subscriber platform. In the absence of multicast support in Java 1.0.2, and the lack of universal Mbone[5] connectivity, the decision was made to create our own distribution topology using Salamander servers in place of existing Mbone infrastructure. With the advent of Java 1.1 we are beginning to implement the Salamander interface using native multicast support.

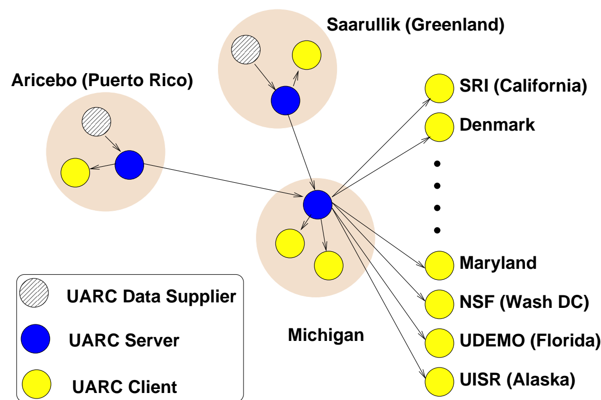

Figure 3: Example UARC campaign topology used during the April 1997 campaign

3.1 Channel Subscription Service

The Salamander substrate provides an abstraction for the distribution of data from publishers to subscribers through its channel subscription interface with both anonymous and negotiated push techniques. The basic idea in *anonymous push* is that publishers package opaque data objects, or Application Data Units as termed in [2], with text-based attribute lists. These attributes can then be used by Salamander to "lookup" destinations for the object. Subscribers place persistent *queries* to the Salamander substrate using lists of attribute expressions that can be used to match both current and future objects published to the Salamander space. Alternatively, this procedure can be thought of as accessing a distributed database where the queries are persistent. These persistent queries are matched by both the objects archived in Salamander's persistent repositories as well as future updates to the database space. These future updates and additions are dynamically matched with outstanding queries that are then pushed to the queries' corresponding clients. The query aspect of Salamander's attribute-based subscription service differs those in traditional nameservice and database systems, in that instead of the queries acting once on a static snapshot of the dataspace, they are dynamic entities that act on both the current state of the system and future updates. Publishers may come and go without affecting the connection between the Salamander database and the subscribers.

Salamander allows for feedback from subscribers to their publishers in the form of *negotiated push*. This is accomplished though the combination of unique endpoint addresses, endpoint namespace registration, and the ability to send unicast messages between Salamander clients. Each Salamander connection is given a unique address that it can insert into a global namespace. Connections manage their entries in this global namespace using the *supply* command. The `supply` command is given an attribute list, similar to the one used in the `query` command, that is paired with its identifier in the namespace. Other clients can then find entries in the namespace by matching the attributes with a *supply query*. Having obtained the identifier of an endpoint, the connection can then send it a *unicast* message. In practice, these sets of commands are used to denote the availability of data or membership in a group. In negotiated push, this mechanism can be used to allow subscribers to ask publishers to begin data distribution, or to modify a supplier's data flows at the source. The UARC application uses this mechanism to turn different data flows on and off. These data flows are computationally expensive, and should only be generated when there is a demand for the data.

A notification service is also provided within the Salamander namespace to allow for propagation of various system events to endpoints. For example, a connection can register a close event with their server that will cause the generation of a *close* notification message to a specified endpoint. In this

way, clients can maintain membership information within their groups.

3.2 Application-Level Quality of Service

The Salamander architecture provides application-level Quality of Service policies to deliver data to clients as best fits their connectivity and processing resources[10]. These policies can rely on either best effort service or utilize network-level QoS guarantees[19] if available. Application specific policies are used to allocate the available bandwidth between a client's subscribed flows, providing a client with an effective throughput based on semantic thresholds that only the application and user can specify. These application-level QoS policies are achieved through the use of plug-in policy modules at points in the distribution tree. Figure 4 shows an example topology with several types of modules.

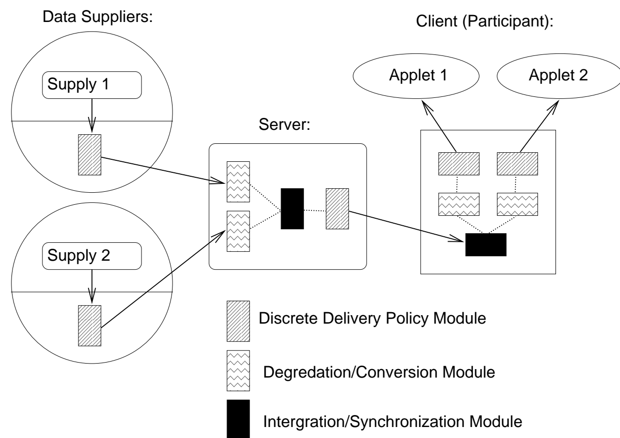

Figure 4: Example collaboratory topology with plug-in modules at suppliers, servers and clients. The architecture allows for the placement of data processing modules at any point in the distributed datapath. This specific example shows the path data takes from two suppliers to two client applets.

To illustrate the use of these modules, we use UARC as an example. Specifically, the UARC application uses discrete delivery, data degradation, and data conversion plug-in modules. By multiplexing the subscribed flows, discrete delivery modules can be used to prioritize, interleave, and discard discrete data objects. We have constructed a flexible interface that allows the client to both: determine its current performance level with respect to the supply, and to gracefully degrade its quality of service from an application-level standpoint so that it best matches the client connection's service level. These quality of service parameters are taken directly from the user in the UARC application. When users discover that they are oversubscribed, they can use the interface to specify priorities among the subscribed flows, and assign drop policies for the individual flows. A *skipover* policy can be used to specify fine-grained drop orders. Current skipover policies consist of both a FIFO *threshold*, and a *drop distance* parameter. The threshold is used to allow for transient congestion; once this threshold of data has accumulated in the proxy, the drop distance parameter is used to determine which data are discarded, and which are delivered.

In addition to discrete delivery policies, the Salamander substrate provides for on-demand data degradation and conversion of data objects. In general, these mechanisms are used to convert one object into another. In order to support real-time collaboration between heterogeneous clients, some mechanism for graceful data degradation must be made available to provide useful data to the slower participants. At the application level, we understand something about the semantics of the shared data. We can exploit this knowledge, and provide a graceful degradation of the data based upon these semantics. The Salamander substrate uses on-demand (lossless and lossy) compression on semantically typed data, tailoring contents to the specific requirements of the clients. Lossless compression techniques are used for those data that cannot be degraded, such as raw text, binary executables, and interpreted program text. Lossy compression techniques are applied to data that can suffer some loss of fidelity without losing their semantic information, examples of which are: still and moving images; audio; and higher level text like postscript or hypertext markup languages. We give the application layer control over this quality by providing an interface that adjusts the fidelity of the real-time data flow on a per-client basis.

We were surprised by our experiences with the UARC project, in discovering that the system bottleneck was not network bandwidth, but was instead the processing power on the clients. The UARC system uses sophisticated Java applets to process raw scientific data as they arrive. During the April 1997 campaign, the Salamander substrate could deliver the UARC data to a large number of clients without difficulty; however, the clients' Java interpreters couldn't keep up with the incoming data rate. Our solution was to put plug-in modules in the distribution tree that could convert the data in-transit to a more manageable format.

Previous work has addressed the variability in client resources. Client resources are addressed in the Odyssey[13] system by sending different versions of the same object from the server depending on the client's resources. In [6], Fox et.al. provide a general proxy architecture for dynamic distillation of data at the server. The use of hierarchically encoded data distributed over several multicast groups is discussed in [12] for the delivery of different qualities of audio and video data. Balachandran et.al target mobile computing in [1] and argue for adding active filters at base stations. Active network proponents [15] argue that Internet routers should be enabled to run arbitrary application-level code to assist in protocol processing. The contribution of our work is the use of client feedback to allow for prioritization among flows; the construction of application and user interfaces for flow modification; and the ability to place modules at any point in the distribution tree.

3.3 Lightweight Temporal Database

Salamander provides data persistence by incorporating a custom lightweight temporal database. This database supports Salamander's needs by: storing a virtual channel's data as a sequence of write-once updates that are primarily based on time; and satisfying requests for data based on temporal ranges within the update stream. A temporal database [14] generally views a single data record as an ordered sequence of temporally bounded updates. In the Salamander database these records correspond to virtual channels. An administrator can determine which sets of virtual channels will be archived by the system. The Salamander system exports only a simple query interface to this database based on ranges of time and attribute lists. By foregoing the complexity of most commercial and research temporal databases, we could build a small and efficient custom database that met our simpler needs.

Salamander's synergy between real-time data dissemination and traditional temporal and relational databases is one of its significant contributions. It is taken for granted that queries on a relational or temporal database act as a static atomic action against a snapshot of a system's dynamic data elements. Our model alters this, by providing support for persistent queries that act over both a snapshot of the data elements present in the database and any modifications (real-time updates) to the database elements that may occur in the future. We plan to further address the impact of this model on database technologies in the future.

In addition to persistent state, Salamander maintains a memory cache used to buffer objects in-transit through the system. Together, the memory buffer and database act as a two-level cache that provides both high-level delivery semantics on the virtual flows and the ability to replay sections of flows from an archive. Salamander's virtual channels in the object space denote implicit groups. The delivery semantics within these groups varies depending on an application's needs. Groups can stay anonymous where the suppliers have no knowledge of the receivers, or they can become more explicit where the suppliers keep a tally of their receivers. A persistent disk-based cache of data objects, in conjunction with the use of application level framing[2], can be used to provide high-level delivery semantics. This includes: FIFO, causal, ordered atomic, etc[7].

4 Salamander Interfaces

There are two interfaces to the Salamander substrate: the application programmer interface (API) and the administration interface. The API has several layers and implementations, depending on the developer's needs. At the lowest level, the substrate provides a simple interface in both C and Java that gives four primitive operations: *send*, *receive*, *connect*, and *disconnect*. These operations are used in conjunction with a *property list* manipulation library to send and receive Salamander data objects. Much of this API can be illustrated by the example application code in Figure 5. This example is a small subroutine that subscribes to a simple virtual channel consisting of a single attribute.

The *connect* operation is demonstrated on line 8 of the listing, where a `connectToSalamanderServer` call is made. While `hostname` is straightforward, the `sskey` parameter requires some explanation. The `sskey` is a simple form of access control to the Salamander substrate. In the current implementation this key is very small, but one can imagine using a more robust key in conjunction with a secure socket implementation to achieve a greater level of confidence. Another security measure, implicit to the client, is an IP access list that restricts the access of data channels and administrative commands. This IP ACL is maintained on a per server basis.

After connecting to the substrate, the example then subscribes to the virtual channel by creating a query and submitted it to the server. This query is constructed in lines 14 through 19. The property list

```
1    void
     subscribeToChannel(char * hostname, unsigned long sskey, char * queryName) {

         plist_t plist;
5        void * dataptr;
         unsigned long data_length;

         if (connectToSalamanderServer(hostname, sskey) != SALAMANDER_OK) {
             fprintf(stderr, "Connection Error.");
10           return;
         }

         /*  Create the query.  */
         plist = createPropertyList();
15       updateProperty(plist, COMMAND_PROPERTY, QUERY_COMMAND);
         updateProperty(plist, NAME_PROPERTY, queryName);
         updateProperty(plist, COOKIE_PROPERTY, "queryCookie");
         sprintf(tmpbuf, "RANGE %d %d", begin, end);
         updateProperty(plist, TIMESTAMP_PROPERTY, tmpbuf);
20
         /*  Send it to the Server.  */
         if (salamanderSendServerData(plist, NULL, 0) != SALAMANDER_OK) {
             fprintf(stderr, "Error making query.");
             return;
25       }
         destroyPropertyList(plist);

         /*  Read the responses as they come.  */
         for (;;) {
30           if (salamanderReadServerData(&plist, &dataptr, &data_length) != SALAMANDER_OK)
                 break;

             handleResponse(plist, dataptr, data_length);
         }
35
         disconnectFromSalamanderServer();
     }
```

Figure 5: Simple C example that connects to a Salamander server and makes a single persistent query.

(plist_t) is the data structure that contains a data object's header and attribute information. Certain attributes are considered *well-known* by the system. An example, is the COMMAND_PROPERTY shown in line 15. The command property tells the substrate what to do with the data object upon receipt. In the example, the command is a query, which is intercepted by the substrate and is processed at the server. The NAME property is the only mandatory query attribute in the current implementation. While any number of attributes can be used to describe a virtual channel, one of them must be the NAME property. The COOKIE property on line 17 is used to match queries with a unique identifier that is returned by the Salamander substrate. This query identifier is added to any object that is returned by a subsequent *receive* operation, and is used to match responses with virtual channels. Finally, on lines 18 and 19, the *well-known* TIMESTAMP property is defined that designates the range of data for which the query corresponds. The *send* operation is carried out on line 22, followed by the destruction of the property list.

After the routine subscribes to the virtual channel, it goes into a loop that reads data objects from the substrate on lines 29 through 34. When the connection is severed, or another error occurs, the code *disconnects* and exits on line 36.

In addition to the base API, Salamander also exports an administrative interface. The substrate can be administered both remotely from a higher level API, and directly on the servers as configuration files. When used remotely, special commands are sent to the server using the base API. Example commands include: list current connections, list active virtual channels, prune a connection, resize a channel's memory buffer, establish a server peering connection, modify debugging level, etc. A server's configuration file is used primarily to establish static server peering relationships, to define default buffer sizes, and specify delivery semantics for specific virtual channels.

5 Performance Evaluation

The performance of the Salamander substrate can be characterized by both empirical and experimental results. Salamander is currently implemented as a multithreaded server on Solaris, and supports a set of complex Java applets that run on a variety of browsers[1,2]. Empirically, the Salamander substrate is used by both the UARC and IPMA projects as their base middleware for daily operation. Salamander has been used for over a year by the UARC scientists and has made a significant impact in the space science research community. The IPMA project has been using Salamander around the clock for over nine months for a variety of tasks. These tasks include: acting as the data collection mechanism for the NetNow probes at the Internet Exchange Points, and as the webcasting delivery mechanism from a central server tree to a collection of Java applets available from its website[8]. The UARC project has gone through several week-long campaigns using Salamander as the data distribution substrate that connects the remote instrument sites to the scientists' Java applets. During these campaigns over sixty scientists have had multiple connections receiving many different types of data. The empirical results from these two projects demonstrate that the Salamander system is both extremely robust and scalable.

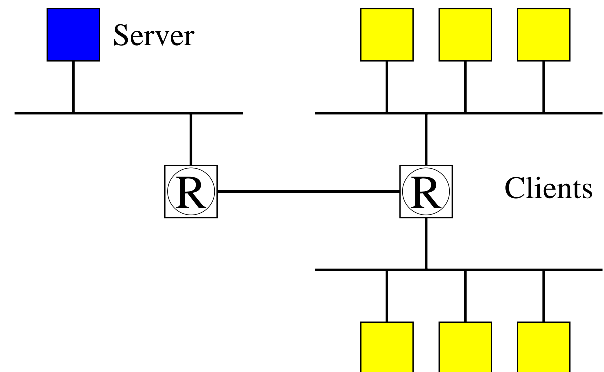

Figure 6: Experimental apparatus for Salamander server performance tests.

A series of experiments were performed on a single Salamander server to help quantify its performance. For all of the following experiments, the same setup was used. This corresponds to the network configuration shown in Figure 6. In this figure, the single Salamander server, shown in the upper left corner, is a SUN Ultrasparc-1 with a 140 MHz processor and 192 Mbytes of memory. A 10 Mbps Ethernet segment connects the server to a router

[1] Although counter-intuitive, cross browser compatibility is not a given. There are significant variations in Java VM implementations between browsers, and it was common during UARC applet development for code that would work under Netscape to break under Microsoft IE or HotJava (and vice versa).

[2] Coincidentally, HotJava continues to be the UARC developers' browser of choice for applet execution.

that is connected by a switching fabric to a second router. This second router has two interfaces that connect to university computing laboratories. Both of these laboratories consist of Ultrasparc-1 workstations connected by 10 Mbps Ethernet LANs.

The performance experiments highlight the scalability of the server in several dimensions: the number of simultaneous connections, and a throughput metric of objects per second. Both of these experiments characterize the substrate's performance in terms of a data object's end-to-end latency from a data supplier to a receiving client. To measure this latency, a timestamp is written into the object's attribute list as it flows down the distribution tree. These timestamps are written to a log file upon receipt at the subscribers. By analyzing the log files offline, the experiment's latency statistics can be extracted. Since the analysis of data in these experiments relies on this distributed timestamp information, a method for synchronizing the clocks on an experiment's hosts was applied. To compensate for the difference in the clocks, a probabilistic clock synchronization technique, similar to the protocols developed by Cristian [4] was used.

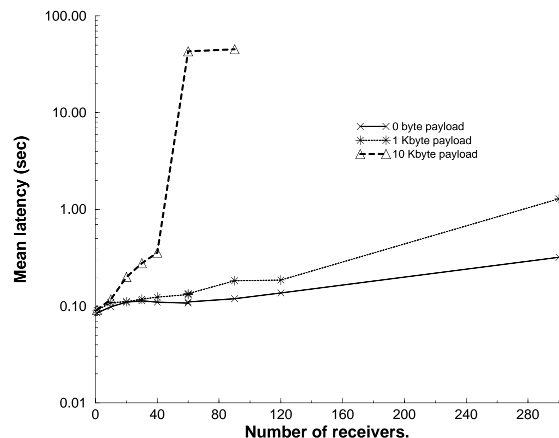

Figure 7: Results from a single supplier Salamander scalability experiment.

Figure 7 shows the results of experiments that evaluate the scalability of a single Salamander server in terms of both number of concurrent receivers and size of the data payload. Three payload sizes were used: 0, 1 Kbyte, and 10 Kbytes. However while the payload varied, each of the object's headers were filled with approximately 100 bytes of testing and channel information. For all of these results, a single data supplier was used that sent an object once per second for a period of five minutes. The horizontal axis represents the number of concurrent receivers during the experiment; whereas the vertical axis shows the mean delivery latency for the objects on a logarithmic scale. These results show that for a small data payload a significant number of clients can be supported. During the execution of these tests the available bandwidth between the server and the laboratories was approximately 400 Kbytes per second. This bandwidth was measured both informally using FTP latency and rigorously with the treno tool[16]. This explains the steep rise in the 10 Kbyte payloads between 40 and 50 receivers, that corresponds to 400 Kbytes and 500 Kbytes per second respectively. At levels of throughput above a link's capacity, Salamander's memory buffer fills and all latencies reach a steady state due to the finite buffer space and drop tail delivery semantics. A virtual channel's buffer space can be specifically tailored to bound this maximal latency of received objects.

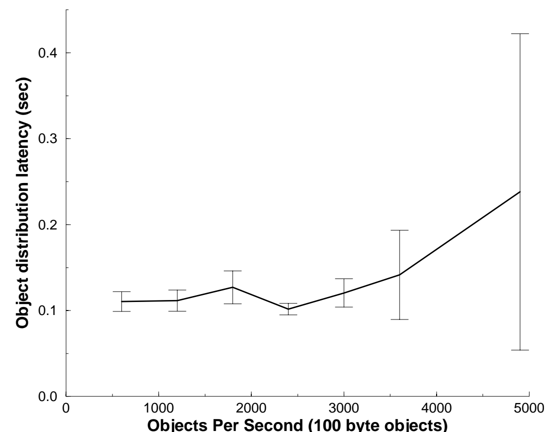

Figure 8: Measured latency of objects through the system when objects per second is varied.

The second set of experiments shows the scalability of a single Salamander server in terms of maximum objects per second (ops) that can be processed. This is similar to a datagram router's packets per second (pps) metric. The results of this set of experiments is shown is Figure 8. The data points in this graph represent the mean and standard deviation of the latency of objects from supplier to client under increasing server load. These points were generated by running experiments that sent an object with a header of 100 bytes and a payload of length zero from a set of senders to a set of receivers. Each

sender sent a single object once per second. For example, in order to generate the 2400 ops data point a series of experiements were run using sets of 40 suppliers and 60 receivers. Unfortunately, the bandwidth between the server and the laboratories never exceeded 500 Kbytes per second for sustained periods while these experiments were undertaken. However, during the 4900 ops experiments the relatively slow 140 MHz processor was approximately 25% idle, leaving room for further scaling. The variation between the data points results from using an active testbed.

Together, these two sets of experiments show that a single server substrate scales with the available network bandwidth. Further improvements in scalability can be made by utilizing a heirarchy of servers in the substrate to offload bandwidth and processing overhead. The empirical results concur, showing Salamander to be both scalable and robust.

6 Conclusion

This paper presented both a functional description of the Salamander distribution substrate's architecture and interfaces; and a quantitative analysis of its performance characteristics. The contributions of the architecture are its combination of channel subscription service with a lightweight temporal database, and the definition and use of an application-level QoS framework.

The Salamander substrate's channel subscription service provides for both an anonymous and a negotiated push of data from a set of suppliers to a set of receivers. The support for anonymous push is straightforward, the suppliers know nothing about its set of receivers. In contrast, negotiated push support enables subscribers and publishers to negotiate the content of their data channels. To provide this functionality, Salamander includes: a registration and matching service, similar to a name service; and a notification service that propagates various system events, including client termination, to Salamander endpoints.

Application-level Quality of Service is defined and supported in the Salamander substrate as a way of tailoring the available resources to best fit the user and application. This is done by utilizing semantic knowledge that only the application has about its data and providing mechanisms for the graceful degradation of its virtual data channels. The specific contribution of our work is the use of client feedback to allow for prioritization among virtual channels; the construction of application and user interfaces for channel modification; and the ability to place channel conversion modules at any point in the distribution tree.

Salamander's incorporation of a lightweight temporal database provides the basis for a powerful synergy between real-time data dissemination and traditional temporal and relational databases. Our model provides support for persistent queries that act over both a snapshot of the data elements present in the database and any modifications (real-time updates) to the database elements that may occur in the future.

Salamander's research contribution is complemented by the utility and robustness of the current implementation. Several Internet applications were described that motivated Salamander's push-based approach to data distribution. These applications, namely the UARC and IPMA project applets, use Salamander around the clock to provide application connectivity throughout the Internet. Through day-to-day use, these applications have shown Salamander's empirical performance to be good. Moreover, the quantitative performance experiments show that a single server scales well for a significant number of connections; the server's bandwidth was the first-order bottleneck in these experiments.

Our current work is focused on enhancing the scalability of the system. Specifically, we are investigating the applicability of scalable routing technologies to Salamander's attribute-based data dissemination. Additionally, we are addressing several system administration aspects of the multiserver substrate, including pairing it with a key distribution mechanism. In concert with these activities, we plan to further address the impact of Salamander's persistent query model on database technologies. Finally, we are in the process of porting the server to both Java and Win32 based platforms, in order to objectively compare the base system in a variety of settings.

7 Availability

Additional information on the Salamander system can be found at the following URL:

`www.eecs.umich.edu/~rmalan/salamander/`

Information that can be found there includes:

- Binaries and source code for the Solaris version of the Salamander server,

- Generic C and Java interface libraries for both data suppliers and clients,
- Status of both a Java and Win32 port of the Salamander server.
- Information about ongoing research efforts based on Salamander.

Acknowledgments

Special thanks go to: Craig Labovitz the indefatigable Salamander evangelist; Jim Wan for his development and debugging aid; Mike Bailey for his constructive criticism; and Michael Willis for his initial efforts. Thanks also go to the entire UARC development team: Peter Knoop, Terry Weymouth, and Mike Burek. The performance experiments would have been unbearable without Don Libes's `expect`. Again we would like to acknowledge the National Science Foundation and Intel for their generous funding that made this work possible.

References

[1] Anand Balachandran, Andrew T. Campbell, and Michael E. Kounavis. Active filters: Delivering scaled media to mobile devices. In *Proceedings of the Seventh International Workshop on Network and Operating System Support for Digital Audio and Video*, May 1997.

[2] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of ACM SIGCOMM '90*, September 1990.

[3] C. R. Clauer et al. A new project to support scientific collaboration electronically. *EOS Transactions on American Geophysical Union*, 75, June 1994.

[4] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[5] H. Eriksson. Mbone: The multicast backbone. *Communications of the ACM*, 37(8), 1994.

[6] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languates and Operating Systems*, October 1996.

[7] Vassos Hadzilacos and Sam Toueg. *Distributed Systems*, chapter 5, pages 293–318. Addison-Wesley, second edition, 1993.

[8] Internet Performance Measurement and Analysis (IPMA) project homepage. http://nic.merit.edu/ipma/.

[9] B. Kantor and P. Lapsley. Network news transfer protocol (nntp). RFC 977, February 1986.

[10] G. Robert Malan and Farnam Jahanian. An application-level quality of service architecture for internet collaboratories. In *Proceedings of the IEEE RTSS Workshop on Resource Allocation Problems in Multimedia Systems*, December 1996.

[11] G. Robert Malan, Farnam Jahanian, and Peter Knoop. Comparison of two middleware data dissemination services in a wide-area distributed system. In *Proceedings of the 17th IEEE International Conference on Distributed Computing Systems*, May 1997.

[12] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *Proceedings of ACM SIGCOMM '96*, August 1996.

[13] Brian D. Noble, Morgan Price, and M. Satyanarayanan. A programming interface for application-aware adaptation in mobile computing. *Computing Systems*, 8(4), 1995.

[14] Abdullah Uz Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard Snodgrass. *Temporal Databases, Theory, Design, and Implementation.* Benjamin/Cummings Publisher Company, 1993.

[15] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *Computer Computer Review*, 26(2), April 1996.

[16] Treno Web Page. http://www.psc.edu/networking/treno_info.html.

[17] Upper Atmospheric Research Collaboratory (UARC) project homepage. http://www.si.umich.edu/UARC/.

[18] Tak W. Yan and Hector Garcia-Molina. SIFT - a tool for wide-area information dissemination. In *Proceedings of USENIX 1995 Technical Conference*, January 1995.

[19] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new resource reservation protocol. *IEEE Network*, September 1993.