The SIMBA User Alert Service Architecture for Dependable Alert Delivery

Yi-Min Wang
Paramvir Bahl
Wilf Russell

March 26, 2001

Technical Report
MSR-TR-2000-117

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

To appear in *Proc. IEEE International Conference on Dependable Systems and Networks (DSN, formerly FTCS)*, July 2001.

The SIMBA User Alert Service Architecture for Dependable Alert Delivery

Yi-Min Wang Paramvir Bahl Wilf Russell

Microsoft Research, Redmond WA

Abstract

Alerts refer to the delivery of user-subscribed information to the user. As the number of alert services and the types of information delivery devices increase, a new model that allows users to manage alert delivery and avoid alert overflow is needed. The unique dependability challenge in the management of alerts is in the proper use of redundancy to achieve timeliness and reliability without being unduly intrusive or cumbersome.

We describe the design, implementation, and user experience of an alert service architecture, called SIMBA. SIMBA utilizes Instant Messaging with acknowledgements as the universal, reliable alert delivery channel, with emails being the fallback channel. All alerts that a user subscribes to are first directed to the user's MyAlertBuddy, which allows centralized delivery preference customization and acts as a personal alert router to protect the privacy of user addresses. Delivery modes, each of which involves multiple user addresses to accommodate communication failures, are supported as an abstraction for specifying personalized dependability levels. A working implementation of the SIMBA system, which integrates five different types of alert services, is described. Challenges and techniques in maintaining a highly available MyAlertBuddy to avoid single-point of failure are discussed. The concept of exception-handling automation is introduced for enhancing the robustness of applications that drive third-party communication client software through automation interfaces.

1. Introduction

The explosive growth of the Web has created a gigantic networked data store that contains a wealth of information for immediate access by anyone with an Internet connection. However, such high availability of potentially useful information has also created information overflow problem for individuals. One way to alleviate the problem is to switch the data access model from information polling and navigation to eventing (or alerting). Instead of periodically browsing through all Web sites with potentially useful information, each individual user specifies the kinds of information that she is interested in from each of the Web sites. In subscribing to such events, the user also specifies a "callback address", e.g., an email address, to which the notification should be sent when the event subscription has a match. Many Web sites already provide such eventing services, usually under the category of "*alerts*". For example, *Amazon.com* offers to send alerts to users when a particular artist's new CD album comes out. The company *Alerts.com* maintains all kinds of alert services for other Web information sites. General Web portal sites such as *Yahoo!* (http://alerts.yahoo.com/) and *MSN Mobile* (http://mobile.msn.com/) provide alert services for stock quotes, weather, sports, lottery, career, real estate, etc. We analyzed a recent one-week usage log from a commercial portal site, and it showed that on average around 225 thousands of people received around 778 thousands of alerts every day from that site.

Several other types of alerts are also emerging. *On-line communities*, such as *MSN Web Communities* (http://communities.msn.com/) and *Groups@AOL* (http://community.aol.com/), allow users from different parts of the world who share similar interests to create virtual communities. Members of a community can share photos, activity calendars, etc. in a password-protected private area. It would be very useful if on-line community members can subscribe to alerts triggered by changes made to any community contents. Another example is the *Aladdin home networking system* [9], which integrates diverse devices and sensors at home and connects them to the Internet. Aladdin generates an alert when any critical sensor fires. *Wireless user-location tracking systems* (e.g. RADAR [1]) are yet another example. Such systems can provide alert services that notify authorized users of location changes of the people being tracked. Finally, *desktop assistant software* can send alerts to a user's cell phone when it detects important reminders or incoming emails while the user is away from the desktop.

The current model of alert subscription and delivery has several dependability-related problems. First, most of the alerts today are delivered as email messages, which are not suitable for delivering time-critical, high-importance alerts. Second, alert users usually require different timeliness and reliability levels for different categories of alerts. Most of today's alert services do not provide customizability at this finer granularity. Third, the above requirements may change over time. Since alerts from multiple sources may belong to the same category, having to visit multiple Web sites to modify or disable alert delivery mechanisms is a cumbersome task and greatly impacts the usability of alert services. Finally, to receive alerts as *SMS* (*Short Message Service)* messages on a cell phone, the user needs to supply the SMS email address. Since the SMS address typically contains the

corresponding cell phone number, providing that information to multiple alert services creates serious privacy concerns.

In this paper, we describe the design, implementation, and user experience of an alert service architecture, called *SIMBA*, to address the above issues. Throughout the paper, we use the term ***dependability*** to refer to the overall user experience of using alert services; more specifically, we focus on ***the capability of delivering alerts in a timely and reliable fashion without being unduly intrusive or cumbersome***. The motivation for this unusual definition of dependability is as follows. Clearly, the timeliness and reliability of alert delivery can be enhanced through heavy use of redundancy; for example, each alert can be delivered as *N* duplicated emails and *N* duplicated SMS messages. However, when considering dependable services that involve human end-users, heavy redundancy often makes the services too irritating and cumbersome to use. The challenge is to take into account the "irritability factor" and figure out the proper trade-off between timeliness/reliability and usability in order to make the overall user experience comfortable. Since such assessment of dependability is clearly a subjective matter, our goal is to provide a framework that simplifies the user task of configuring personalized dependability levels and associating them with the delivery of various alerts based on each individual user's experience.

The overall contribution of SIMBA is to study the dependability issues and solutions in the context of a new-style distributed, Web-based service. Specific contributions include:

- To support delivery of time-critical, high-importance alerts, SIMBA utilizes Instant Messaging (IM) with user acknowledgements for end-to-end synchronous, reliable delivery. Like emails, instant messaging is becoming another standard way of communication over the Internet for people to exchange short, fast messages. SIMBA extends the use of instant messaging to communications between potentially non-human entities.

- To support personalized dependability requirements for alert delivery, SIMBA introduces the concept of *delivery modes.* Each delivery mode involves potentially multiple addresses to accommodate communication delays and failures. The user defines a set of personalized delivery modes, each of which corresponds to a personalized dependability level.

- To support easy customizability, SIMBA introduces *MyAlertBuddy (MAB*) as a level of indirection between alert services and users. Each user has a MAB, which is always online for receiving and acknowledging IM-alerts. The MAB uses an email address as its fallback mechanism. All alerts to a user are first directed to the user's MAB, which then determines the best way at that time to route the alerts to the user. The best way is based on her static and dynamic preference of dependability. Since only the addresses of the MAB are revealed to the various alert sources, the privacy of user addresses is greatly improved.

- In terms of implementation, we incorporate extensive fault-tolerance mechanisms into *MyAlertBuddy* to avoid single-point of failure. In particular, we introduce the concept of exception-handling automation and demonstrate how it enhances the robustness of applications that drive the IM and email communication client software through automation interfaces.

The rest of this paper is organized as follows. Section 2 describes the alert services that have been connected to SIMBA. The diversity of these alert sources was the main motivation for our dependability study. Section 3 describes the dependability issues associated with the current model of user alert service, and the approach taken by SIMBA to address these issues. Section 4 presents the design and implementation of SIMBA, with emphasis on exception-handling automation and its use in maintaining a highly available *MyAlertBuddy*. Section 5 presents experimental results. Section 6 surveys related work. Section 7 summarizes the paper.

2. Alert Services

In this section, we describe a number of alert services that we consider, as illustrated in Figure 1. We classify them into five different types, and give a brief description of each type with sample scenarios that demonstrate the benefits they provide. In today's world, for each interesting alert service, the user visits the service's alert subscription page and enters subscriptions based on categories, keywords, etc. The user also supplies a preferred email or SMS address, to which the alerts should be sent.

2.1. Information Alert Services

Information alert services including the five shaded boxes in Figure 1 and many other general news and special-purpose alert services are currently supported in SIMBA. For example, *eBay* can send alerts to users when they are outbid or when an auction is about to be closed. *CNN Sports Illustrated* can send an alert when a particular news column is updated. Most of them follow the traditional delivery mechanism of sending alerts as emails.

For Web sites that provide interesting information but do not yet support alert services, we use an *alert proxy* to generate alerts for them. For each Web site, the user specifies the URL, the polling frequency, the starting and ending keywords enclosing the interesting block of information. The alert proxy periodically polls the site and generates an alert when the interesting block changes. For example, an alert proxy was constructed to monitor the year 2000 presidential election results and configured to

send an alert whenever the Florida recount updated the number of votes.

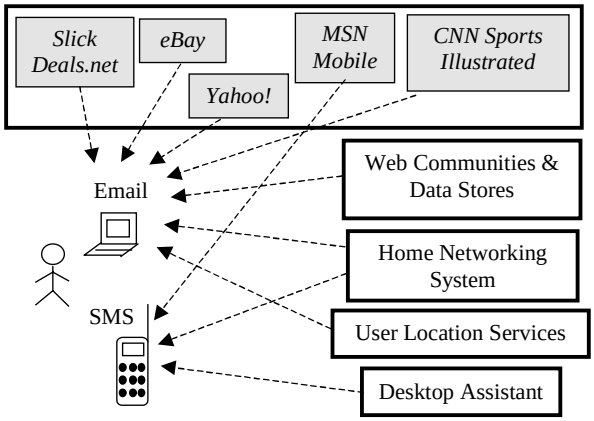

Figure 1. Current Model of User Alert Service: user enters alert subscriptions as well as alert delivery preference at each alert service.

2.2. Web Store Alert Services

In contrast with information alert services that generally provide information interesting to the public, Web store alert services notify users when changes are made to their private data or shared community data stored on the Web. For example, some of today's credit card companies provide confirmation alerts when they cash customers' payment checks. In the future, users are expected to store more and more data on the Web in a device-independent format so that they can get access to the data at any time, from any place, and on any device. As a result, we speculate that alert services that report data changes by, for example, authorized agents will become popular.

To allow timely delivery of certain alerts that a user may be eagerly waiting for on a particular day, we use the alert proxy to periodically monitor the community sites and send alerts upon detecting changes. For example, when a new photo is added to the shared community photo album, interested members can receive an alert containing the URL, which they can click to see the picture.

2.3. Home Networking Systems

The Aladdin system provides a distributed system infrastructure and a programming toolkit for building dependable home networking applications [9]. Aladdin integrates diverse devices and sensors attached to heterogeneous in-home networks including powerline, phoneline, RF (Radio Frequency) and IR (InfraRed), and connects them to the Internet through a home gateway machine. In addition to supporting secure, email-based remote home automation, Aladdin generates alerts when any critical sensor fires or when any critical device fails. For example, flooding in the basement would generate a "Basement Water Sensor ON" alert; garage door sensors running out of battery would trigger a "Garage Door Sensor Broken" alert.

To minimize the potential problem of message loss and delay, Aladdin by default sends all alerts as two emails and two cell phone SMS messages. However, such heavy use of redundancy has not worked well. For critical alerts, there is still no guarantee that any of the four messages can reach the user in time. For less critical alerts, four messages per alert are irritating and cumbersome. As we will discuss later, moving Aladdin to the SIMBA architecture with Instant Messaging and *MyAlertBuddy* greatly improves its usability and dependability.

2.4. User Location Services

Tracking of mobile users has long been considered a necessary service for building location-aware applications [1]. The WISH system, developed at Microsoft Research, is a location-determination system that addresses privacy concerns by leaving the control of location information dissemination solely with the user. The WISH client software, running on the user's handheld device, extracts from its RF wireless network card the identity of the Access Point (AP) the device is connected to and the strength of the signals received from the AP. It then sends that information along with the user's name and activity status to a WISH server. The WISH server maintains an RF signal propagation model and a table that maps each AP to a physical location. Using the information provided by the client, the WISH system is able to determine the user's real-time location to within a few meters. A confidence percentage is associated with each estimate.

The WISH location alert service provides a Web-based interface for people to request location tracking of wireless users. A user of the alert service specifies the name of the person to track and the address for alert delivery. An alert can be generated when the tracked person enters a building, moves to a different part of the building, and/or leaves the building.

2.5. Desktop Assistants

At work, people are spending an increasing percentage of their time using the email/calendar software. In addition to sending and receiving emails, the software also serves as a personal alert service that generates time reminder alerts on a user's screen. Ideally, if the user is not there to see the reminders pop up, the important ones should be routed to a device that can get the user's attention.

We have built a SIMBA Desktop Assistant that runs on a user's primary machine and remains inactive until the idle time of interactive activities exceeds a user-specified threshold and the software determines that the user has not processed emails from other places. Currently, the Assistant software generates alerts when high-importance emails come in and when high-importance reminders pop

up. Since the user is likely to be away from any machine, all alerts are generated as SMS messages.

3. Dependability Issues and the SIMBA Approach

The current model of user alert services as illustrated in Figure 1 suffers from several dependability-related issues. We discuss them in this section and present the approach we adopted in SIMBA to address these issues.

3.1. End-to-end Reliable Delivery of Time-critical Alerts

It is well understood that email delivery is not guaranteed to be reliable, and the unpredictable delivery time can range from seconds to days. Our experience with the cell phone SMS delivery time with a large carrier shows a similar range of unpredictability. This is clearly not acceptable for delivering time-sensitive alerts. For example, alerts from Web bargain sites such as http://slickdeals.net/ should preferably be delivered before the hot deals or electronic coupons expire. Weather Advisory Alerts from http://www.alerts.com/ should reach users before the severe weather affects their travel or commute. Date reminders from *MSN Calendar* (http://calendar.msn.com/) must be delivered in time for users to catch their appointments. Critical sensor events from Aladdin home networking system such as water leakage, power outage, door opening, etc. must be delivered as fast and as reliably as possible for users to prevent damages. Finally, the location information of a wireless user is only useful if it can be delivered before the user changes the location significantly.

To support timely delivery of critical alerts, SIMBA makes heavy use of Instant Messaging (IM) as an alert delivery mechanism. Following emails and Web browsing, IM is becoming another Internet communication standard, supported by most of the large Internet Service Providers and used by tens of millions of people worldwide. Instant Messaging allows each user to specify a list of buddies (or contacts), get notified when any of them logs in, and initiate synchronous, text-based message exchange. As IM moves beyond the desktops and starts to appear on cell phones and PDAs, it will become a ubiquitous communication channel. SIMBA extends the use of IM to communications between potentially non-human entities, essentially turning it into a general application-level communication protocol.

To guarantee end-to-end, reliable delivery of alerts, SIMBA relies on application-level acknowledgements tagged with IM message sequence numbers. Note that Instant Messaging services do provide presence and activity status information of a user's buddies. So one may be tempted to simply use such information to determine whether synchronous, reliable communication can be performed successfully. In SIMBA, we decided against such an approach because the user may be separated from the devices and because such information is always potentially stale. Only an explicit acknowledgement from the user can confirm end-to-end reliable delivery of any alert.

3.2. Delivery Modes

One disadvantage of using Instant Messaging for alert delivery is that the delivery would fail if the subscribing user is not logged on to the IM service. A natural way to solve this problem is to use emails as a fallback mechanism when IM delivery fails. In other words, Instant Messaging can be used as the primary delivery mechanism to synchronously and reliably deliver alerts whenever possible, while emails are used as a fallback mechanism for asynchronous, store-and-forward type of delivery.

In the SIMBA architecture, we generalize the above notion to the concept of ***delivery modes***. Each user supplies multiple addresses for receiving alerts. These may typically include the user's work and personal IM addresses, cell phone SMS addresses, and email addresses. Each delivery mode contains one primary *delivery block*, optionally followed by one or more ordered backup delivery blocks. Each delivery block consists of one more *delivery actions.* Each action involves exactly one address, specifies whether an acknowledgement is required and, if so, how long the send operation should wait for the acknowledgement. Alert delivery is first carried out using the actions in the primary block. If any of the actions fails, the block is considered to have failed and the remaining blocks are tried in sequence until one of them succeeds or the list is exhausted. An action may fail either because the associated address has been temporarily disabled, or because the send operation to that address returns an error (e.g., network disconnection, server unavailability, IM user not logged on, third-party communication client software hanging), or because the required acknowledgement is not received within the specified amount of time.

For each alert category, the user specifies a delivery mode, instead of a single IM or email address, as the alert delivery mechanism. For example, for extremely urgent alerts (e.g. laundry room flooding) from a home networking system, the user may specify a delivery mode that includes actions involving all available addresses in the primary delivery block to ensure that the alerts reach the user as soon as possible. For time-critical but less urgent alerts (e.g., date reminders), the user may specify a primary block that uses IM and requires an acknowledgement, followed by a backup block that contains one action through SMS and another one through emails. For less time-critical alerts (e.g. weather, lottery, and stock quotes), the user may stay with the traditional delivery mode that uses only email delivery.

The concept of delivery modes is essentially a simplified version of rule-based communication

redirection. It had been pointed out that, in the area of general person-to-person communications, the design of user interfaces for rule-based communication filtering has been a notoriously hard issue [6]. With potentially multiple rules in action, it is difficult for users to understand exactly how a message containing certain contents will be routed. The issue is simplified as we focus on only alerts because alert services typically separate *alert subscription* (which performs content filtering) and *alert delivery* (which performs communication routing) into two steps. The proposal of delivery modes is an attempt to provide a simple abstraction for users to reason about the second step, while accommodating unavoidable potential communication failures.

3.3. *MyAlertBuddy*: Decoupling Alert Subscription from Alert Delivery

Even with added IM capability for dependable delivery and the use of delivery modes to allow flexible specification of delivery mechanisms, the current alert service model still suffers from another dependability-related problem: it makes dynamic customization difficult. In short, the current model *maps a service to a delivery mechanism at the service site*, while the ideal model from a user's perspective would be to *map each personal alert category to a delivery mechanism at a central, personalized site*.

Specifically, today's Web portal sites usually provide alerts of multiple categories, some of which require more delivery dependability than the others. But the users are usually forced to specify a single delivery mechanism for all alerts from the same service. In addition, alerts from multiple services may naturally belong to the same category. For example, a user may consider that stock quote alerts from *Yahoo!*, financial news from the *Wall Street Journal*, and news column alerts from *CBS MarketWatch* all belong to her personal "Investment" alert category and should share the same delivery mechanism. Today's service model requires the user to specify the delivery mechanism at each of these services, making subsequent changes a cumbersome task. Suppose the user specifies the SMS address at all three services. If one day the user needs to make timely investment decisions and would like to temporarily switch the delivery mechanism for all "Investment" alerts from SMS to IM, she would need to visit all three services to make the change. In another scenario, the same efforts would be required if the user travels to an area where her cell phone doesn't work and so needs to switch the delivery mode from SMS to emails. In yet another scenario, she may need to disable these alerts during certain hours to avoid distractions; some of these services may not provide consistent or even compatible interfaces for such customization.

To simplify dynamic customization of alert categories and delivery in response to changes in either user preference or the availability of certain delivery mechanisms, we introduce *MyAlertBuddy* in the SIMBA service architecture, as shown in Figure 2. Each user has such an alert buddy, operating between all alert services and the user. *MyAlertBuddy* has its own IM and email addresses, and is always logged on to the IM and email servers. When subscribing to any alert service, the user supplies the buddy's IM and email addresses, instead of her own, for alert delivery. All alerts for a user are first sent to the user's *MyAlertBuddy*, which then performs personalized alert routing.

This architecture has several advantages. It protects the user's privacy because the user's cell phone numbers and IM addresses are not revealed to any alert source. Receiving unwanted alerts at those addresses would have been extremely intrusive. For alert services that support SIMBA delivery modes (see Section 4.2), most alerts would be instantly delivered to and acknowledged by *MyAlertBuddy*, greatly enhancing the chance of dependable delivery of critical alerts. To existing alert services that support only email delivery, *MyAlertBuddy* looks just like any other regular human user. To the user, *MyAlertBuddy* serves as a highly customizable, private alert source. The user registers all her addresses with the buddy and defines several personalized delivery modes, each identified by a friendly name. The user also defines a set of personal alert categories and specifies how the native alerts from other sources should be mapped to these personal categories. For each personal category, the user assigns a delivery mode that matches the dependability requirement.

Later on, whenever the user needs to change the delivery modes of some alerts, she only needs to update *MyAlertBuddy*. When the user's cell phone runs out of battery power or when the carrier does not cover the area of the user's location, she only needs to ask *MyAlertBuddy* to temporarily disable her SMS address. Any delivery block that contains an SMS action will automatically fail and fall back to the next backup block. Enabling and disabling of some categories of alerts and specifying delivery time constraints can also be conveniently and consistently performed with the alert buddy. Effectively, *MyAlertBuddy* serves as a *personal alert aggregator* that absorbs alerts from diverse sources and re-classifies them based on the user's preferences; a *personal alert filter* that temporarily blocks unwanted alerts, which might have been useful before and may be useful in the future; and a *personal alert router* that knows at any given time what the user's preferred way of receiving a particular type of alert is.

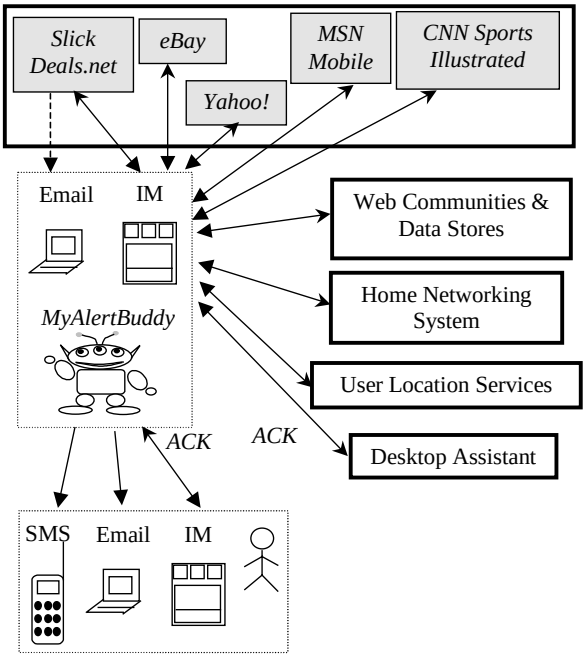

Figure 2. The SIMBA Model of User Alert Service: user enters alert subscriptions at each service and asks all alerts to be delivered to *MyAlertBuddy*.

4. Design and Implementation of SIMBA

In this section, we describe the implementation of the SIMBA library and *MyAlertBuddy*. The library is used by both *MyAlertBuddy* and some of the alert sources. Currently, *MyAlertBuddy* runs on a desktop PC owned by the user.

4.1. The SIMBA Library

Figure 3 illustrates the main components of the SIMBA library and how they interact with other third-party software. The library is divided into two layers:

(1) Subscription Layer: this layer provides APIs for users to register their addresses, personal alert categories, and personal delivery modes. It provides a subscription API for mapping a category name to a user with a particular delivery mode. Each category can have multiple subscribers, each of which can specify a different delivery mode.

Both user addresses and delivery modes are expressed in XML (eXtensible Markup Language) to allow extensibility for accommodating new communication addresses. An XML document for user addresses consists of a list of all of a user's addresses for alert delivery. Each address is associated with a communication type (e.g., "IM", "SMS", and "EM") and identified by a friendly name such as "MSN IM", "Work email", etc. An XML document for a delivery mode contains one or more communication blocks, each of which contains one or more actions. Each action maps to the friendly name of an address. Figure 4 shows a sample delivery mode document with two communication blocks. When a native alert arrives, the subscriptions of the matching category are identified and the corresponding delivery mode XML documents are parsed. Only actions that map to enabled addresses at that time are performed.

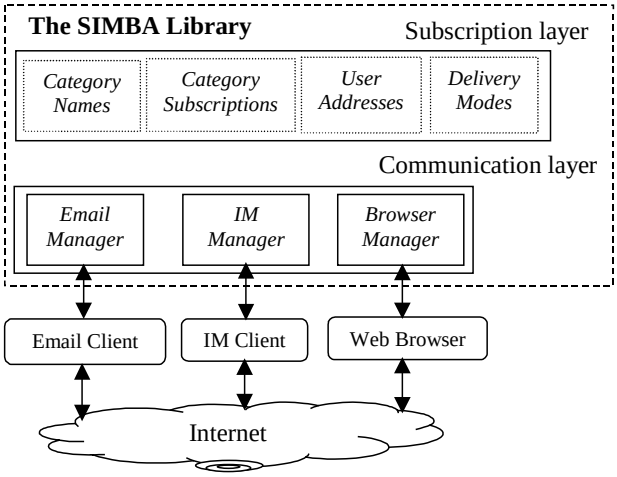

Figure 3. Components of the SIMBA library.

```
<comm_mode>
    <comm_block>
        <comm_action>
            <name>MSN_IM</name>
            <ack_mode>yes</ack_mode>
            <ack_time>20</ack_time>
        </comm_action>
    </comm_block>
    <comm_block>
        <comm_action>
            <name>AT&T_Text_Messaging</name>
            <ack_mode>no</ack_mode>
        </comm_action>
        <comm_action>
            <name>Work_email</name>
            <ack_mode>no</ack_mode>
        </comm_action>
    </comm_block>
</comm_mode>
```

Figure 4. Sample XML Delivery Mode Document.

(2) Communication Layer: this layer provides APIs for programmatically performing Internet communications that are usually performed by humans. It currently consists of three Communication Manager components: the Email Manager, Browser Manager, and IM Manager. These Manager components encapsulate all interactions with their respective third-party, GUI-centric communication client software. (The current implementation uses Microsoft Outlook, Internet Explorer, MSN Messenger.)

4.1.1. Exception-Handling Automation

To allow *MyAlertBuddy* to be interposed between the services and the users in a fully compatible fashion, it is

desirable for *MyAlertBuddy* to use exactly the same email and IM client software that human users use to send and receive alerts. Some of these GUI-centric communication client software packages support *automation interfaces* for exactly this purpose: they allow programmatic access to virtually all the operations that can be performed by human users so that programmers no longer need to use low-level networking APIs to send raw data and implement Internet communication protocols. All the Communication Manager components use automation interfaces to drive the client software.

However, our experience in using automation interfaces to build highly available daemon processes reveals that the current design of automation interfaces suffers from a crucial dependability problem: while they are designed to model the normal use of software by human beings, they do not model and simulate human operations in case of exceptions. Specifically, when a human user is using the software and observes that it is not working properly, the user usually performs simple diagnosis by clicking around in the GUI. Sometimes that forces the software to reestablish a clean connection with a remote server and be able to resume correct operation. Other times the software is badly hung and the only thing the user can do is to kill and restart the software. In addition, sometimes either the software itself or other parts of the system may pop up a dialog box to report problems. The user has to click a button on the dialog box to allow the software to continue operation.

To guarantee continuous operation, it is essential for any daemon processes built upon automation interfaces to be able to programmatically perform all the above exception-handling operations. In SIMBA, we encapsulate these operations in the Communication Managers, which provide the three APIs described below to support *exception-handling automation* for their associated communication client software.

- **Sanity Checking API**: the API starts by checking if the process of the client software is still running and if the pointers to the client software are still valid. Then it performs a series of application-specific checks. For example, the IM Manager checks if the IM client software is still logged on to the server. If it has been logged out due to, for example, server recovery or network disconnection, it will be re-logged in. The IM Manager also checks to see if it can launch IM sessions, obtain the status of the buddies, etc.

- **Shutdown/Restart API**: this API encapsulates the details needed to shut down and restart the associated client software. When an application invokes the sanity checking API to learn of any unfixable anomaly and calls the restart API, the Manager terminates the currently running instance of the client software, restarts another instance, and refreshes all its pointers to point to the new instance.

- **Dialog-box Handling API**: the third API deals with dialog boxes. In GUI applications, dialog boxes are commonly used as a way to report information or warnings to the user, and to receive user input on the preferred course of action. Such dialog boxes should never pop up when the software is driven by a program through automation interfaces because the program cannot interact with the boxes, which then stay on the screen forever and prevent the entire application from making progress. Unfortunately, the email and IM client software used in SIMBA do pop up some dialog boxes that cannot be closed through any automation interfaces. What makes the situation even more complicated is that other parts of the system can pop up dialog boxes that are out of the control of the client software.

A comprehensive solution to the above problem requires careful redesigns of both the applications and the OS to support so-called headless operations. In SIMBA, we adopted a temporary fix that has proved to be effective in solving the hanging dialog box problem. Each Communication Manager maintains a "monkey thread", whose only job is to look for dialog boxes with matching captions and "click" on the appropriate buttons by sending mouse-button-down and then mouse-button-up messages. In each Manager, some of the caption-button pairs are system-generic, while the rest are specific to the associated client software. To handle dialog boxes that are specific to each operating environment, each Manager provides an API for specifying additional caption-button pairs.

4.2. *MyAlertBuddy*

To better integrate the alert services into the SIMBA architecture, we modified the information alert proxy, web store alert proxy, Aladdin home gateway server, WISH alert server, and the desktop assistant to use the "IM-with-acknowledgement followed by email" delivery mode of the SIMBA library to deliver alerts to *MyAlertBuddy*. Upon receiving an alert, the operations and functionality of *MyAlertBuddy* can be divided into four parts.

Alert classification: *MyAlertBuddy* first invokes the *Alert Classifier* to extract category information from the alert. In advance, the user customizes the classifier by specifying the list of accepted alert sources, and how to extract category-related keywords from the alerts. For example, the keywords in alerts from *Yahoo!* and *Alerts.com* appear as part of the email sender name, while the keywords in *MSN Mobile* alerts and desktop assistant alerts reside in the email subject field. As part of the alert classification procedure, *MyAlertBuddy* also helps the user maintain a list of all the subscribed alert services, and the information about how to unsubscribe them.

Alert aggregation: The user can also specify the mappings from those keywords to a set of personalized alert category names. For example, alert aggregation can be achieved by mapping all of "Stocks", "Financial news",

and "Earnings reports" to a single category called "Investment".

Alert filtering: *MyAlertBuddy* can also provide alert filtering through selective sub-categorization. For example, since the Aladdin system does not support content-based event subscriptions, all state changes of any sensor declared as critical will trigger alerts to be sent to *MyAlertBuddy*. By mapping "Sensor ON" and "Sensor OFF" to two different subcategories, the user can treat one of them as more urgent than the other and assign different delivery modes to them.

Alert routing: Once an incoming alert is assigned a personalized category, *MyAlertBuddy* enumerates through all subscriptions of that category, parses each delivery mode, and invokes appropriate Email and IM Manager APIs to deliver the alert. Although *MyAlertBuddy* provides primarily a personalized service, it supports multiple subscribers per category to allow alert sharing.

4.2.1. Achieving High Availability

The most critical dependability task in the SIMBA architecture is to maintain a highly available *MyAlertBuddy*. We next describe the various failure scenarios that *MyAlertBuddy* must recover from and how we apply several fault-tolerance techniques including pessimistic logging, watchdog, self-stabilization, and software rejuvenation to achieve that goal.

- **Pessimistic Logging**

It is possible that, after *MyAlertBuddy* receives and acknowledges an IM alert and before it finishes processing the alert, *MyAlertBuddy* may crash or get terminated due to some anomaly. Since the sender has received the acknowledgement and will not resend the alert, the alert would be lost in such a case. To solve this problem, we use pessimistic logging [3]: upon receiving an IM, *MyAlertBuddy* instructs the SIMBA library to save a copy to a log file before sending the acknowledgement. After processing the IM, *MyAlertBuddy* marks the saved copy as "Processed". Every time *MyAlertBuddy* is restarted, it first checks the log file for unprocessed IMs before accepting new alerts. Another problem is that duplicated alert deliveries may occur if *MyAlertBuddy* fails after sending out an alert and before marking the corresponding received IM as "Processed". We use timestamps to allow the user to detect and discard duplicates.

- **Watchdog**

MyAlertBuddy is always launched by a watchdog process called *Master Daemon Controller (MDC)*, which monitors *MyAlertBuddy* and restarts it upon detecting its termination. The MDC also periodically invokes a non-blocking *AreYouWorking()* function call and restarts *MyAlertBuddy* if it is hung and fails to respond to the call. The *AreYouWorking()* function is implemented as follows. Upon starting, *MyAlertBuddy* makes a call to *MDCInitialize()* with the function pointer to *AreYouWorking()* as a parameter. The call *MDCInitialize()* creates an MDC client thread, which communicates with the MDC using Windows NT event synchronization objects. The MDC invokes the checking by signaling the event to trigger the client thread to call *AreYouWorking()* inside *MyAlertBuddy*. If the call successfully returns, the client thread signals another event as a reply. If the reply event is not signaled within a time limit, the MDC terminates and restarts *MyAlertBuddy*. If the number of failed restarts exceeds a threshold, the MDC reboots the machine.

- **Self-Stabilization**

MyAlertBuddy interacts with third-party software that in turn interacts with remote email and IM servers. Problems occurring at any of these involved entities may cause unexpected behaviors. Since it is very difficult to anticipate all possible failures and to detect and recover them on the spot, *MyAlertBuddy* incorporates self-stabilization mechanisms that periodically check system invariants and correct violations. Inside the *AreYouWorking()* callback made by the MDC, *MyAlertBuddy* checks the health of the process and the threads by monitoring the timestamps of their progress and unusual system resource consumption caused by, for example, memory leaks in rarely executed branch of code or in third-party software. On a higher frequency, *MyAlertBuddy* periodically invokes the APIs provided by the Email and IM Managers to check the sanity of the communication client software and the availability of their corresponding servers. It also checks for violations of other application-specific invariants including unprocessed emails and IMs due to potential loss of new-email and new-IM events, unprocessed dialog boxes, etc. Currently, the *AreYouWorking()* callback is invoked every three minutes, the sanity checking APIs are invoked every minute, and unprocessed dialog boxes are checked every 20 seconds.

- **Software Rejuvenation**

Rejuvenation is a technique that gracefully terminates an application and immediately restarts it at a clean internal state [5] It has been recognized as a useful technique for increasing the availability of continuously-running service applications. We perform three kinds of rejuvenation tasks in *MyAlertBuddy*: (1) whenever *MyAlertBuddy* catches an exception that cannot be handled or any of the self-stabilization checks reveals invariant violations that cannot be rectified, *MyAlertBuddy* gracefully terminates and gets restarted by the MDC. (2) Every night at 11:30PM, *MyAlertBuddy* requests an orderly shutdown of all the communication client software and terminates itself. (3) To facilitate remote administration of *MyAlertBuddy*, SIMBA allows users to

send IMs or emails with special keywords to explicitly trigger rejuvenation.

5. Experimental Results

The experimental setting illustrated in Figure 5 was used to measure the performance and effectiveness of SIMBA. The one-way IM delivery time from any of the alert sources to *MyAlertBuddy* is typically less than one second. With pessimistic logging, the alert source receives an acknowledgement in about 1.5 seconds. An alert proxy was set up to monitor the Florida recount numbers and the availability of the PlayStation2 game consoles by polling the associated Web sites. When the proxy detected a change, it sent out an alert, which on average took 2.5 seconds to route through *MyAlertBuddy* to reach the user.

The Soft-State Store (SSS) server [9] is a daemon process that maintains a store of soft-state variables, each of which is associated with a required refresh frequency and the maximum number of allowed missing refreshes before the variable is timed out. Clients of SSS can define data types, create variables, read/write variables, and subscribe to events relating to changes in the types or variables. The SSS is used in the Aladdin system as described in the following scenario: the kid returned home from school and used a remote control to disarm the security system. The RF signal was received by a powerline transceiver and converted into a powerline signal. A powerline monitor process running on a PC picked up the signal and converted it into an update on the local SSS server, which replicated the update to other PCs through a multicast over the phoneline Ethernet. The SSS server running on the home gateway machine fired an event to the Aladdin home server, which then sent out an IM alert. From the time the button on the remote control was pushed to the time an IM popped up on the user's screen, the end-to-end delivery took an average of 11 seconds.

In the wireless location service scenario, the WISH client software running on the user's laptop periodically sends the location information to the WISH server. The server updates the Soft-State Store, in which each user is represented by a soft-state variable. If another person has entered an alert subscription corresponding to that update, the WISH Alert Service sends an alert through SIMBA. From the time the laptop sends out the information wirelessly to the time the subscriber gets notified by an IM alert, the average end-to-end delivery time was measured to be 5 seconds.

We also measured the elapsed time for *MyAlertBuddy* to perform rejuvenation. The entire process started with *MyAlertBuddy* shutting down the IM and email client software before it itself exited, and ended when *MyAlertBuddy* restarted the software and successfully logged on to the two servers. Depending on the server response time, rejuvenation time ranged from 8 seconds to 40 seconds. Since rejuvenation is performed either at night when there is little alert traffic or when an exception that cannot be handled occurs, we have found that the short period of unavailability due to rejuvenation is acceptable. Table 1 summarizes all performance numbers.

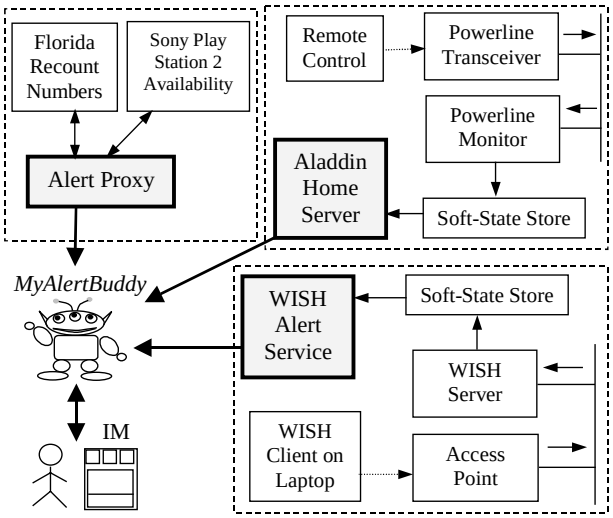

Figure 5. Service Configurations for SIMBA Performance Measurements.

Table 1. Performance Numbers

	Elapsed Time (in seconds)
IM with acknowledgement	1.5
Alert proxy to user	2.5
Home remote control to user	11
Tracked laptop to subscriber	5.0
Rejuvenation	Between 8 and 40

To measure the effectiveness of the fault-tolerance techniques described previously, we have instrumented both the SIMBA library and the *MyAlertBuddy* to log all recovery actions. Preliminary results from the log file show that, within a one-month period of time, there were five extended IM downtimes lasting from 4 to 103 minutes. These could be due to actual IM service unavailability, corporate proxy server unavailability, network connection problems, etc. In addition, there were nine instances where *MyAlertBuddy* was logged out and simple re-logon attempts worked. In another nine instances, the hanging IM client had to be killed and restarted in order to re-log in. There were 36 restarts of *MyAlertBuddy* by the MDC. Most of them were triggered by IM exceptions caused by the use of an earlier version of undocumented interfaces. The fault-tolerance mechanisms effectively recovered *MyAlertBuddy* from all failures

except three: one failure was caused by a rare power outage in the office; another two were caused by previously unknown dialog boxes. UPS and dialog-box handling APIs were then used to fix the problems.

In summary, the experimental results show that SIMBA successfully leverages IM to provide efficient alert routing and delivery, and the fault-tolerance techniques for maintaining a highly available *MyAlertBuddy* are crucial and effective.

6. Related Work

Several recent efforts have been proposed to address the issue of user mobility for general person-to-person communications. Universal Inbox [7] proposed a scalable and extensible architecture for supporting user mobility across devices attached to heterogeneous networks. For each type of network, an Access Point is provided to bridge network-specific sessions to generic sessions to the Internet core. Name translation, preference-based redirection, etc. are provided as reusable components inside the Internet infrastructure. In contrast, the Mobile People Architecture (MPA) [6] provides similar functionalities through a Personal Proxy, which protects the privacy of the user's reachability information and is deployable without requiring modifications to existing networking infrastructures.

In contrast with the above efforts that mostly deal with the functional aspects of general inter-person communications, SIMBA focuses on a specific type of one-way communications, namely, the delivery of alerts, and places emphases on dependability issues. As pointed out in Section 3.2, the focus on alerts has greatly simplified the hard issue of providing usable user interfaces for rule-based communication redirection. The abstraction of delivery modes has proved to be useful in providing overall dependable delivery of alerts. The SIMBA *MyAlertBuddy* plays a similar role as the MPA Personal Proxy. Although both MPA and SIMBA allow personal-level routing without modifications to existing networking infrastructures, they introduce a single point of failure in their respective architectures. The fault-tolerance techniques that we have adopted to successfully maintain a highly available *MyAlertBuddy* can also be applied to constructing a robust Personal Proxy.

The PRIORITIES system [4] adopted decision-theoretic approach to attention-sensitive alerting. It provides automatic assessment of the expected criticality of email messages, and makes context-sensitive decisions on whether and how to alert users about the messages. Such techniques are essentially orthogonal to SIMBA, which focuses on the service architecture for dependable, mechanical delivery of alerts.

Finally, there has been a significant amount of work on event notification services [2][8], which are mostly orthogonal to our work on alert delivery services: while the former focuses on enabling distributed applications to generate events/alerts at the user-alert sources, the latter focuses on taking those alerts and delivering them to the multiple devices of end users.

7. Summary

We have presented the SIMBA user alert service architecture and demonstrated its four contributions. First, we have proposed the use of Instant Messaging (IM) with acknowledgements for end-to-end, timely and reliable delivery of critical alerts. Second, we have introduced delivery modes as a simple and effective abstraction for users to specify their personalized dependability requirements, taking into account unavoidable communication failures. Third, we have described the use of *MyAlertBuddy* to protect the privacy of user addresses as well as to support flexible and effective alert management. Finally, we have introduced the concept of exception-handling automation; presented an implementation for IM and email client software; and described how it is used to enhance the robustness of *MyAlertBuddy*. At the time of this writing, the SIMBA system has been operational for about 6 months and is being used on a daily basis. The fault-tolerance techniques for maintaining a highly available *MyAlertBuddy* have proven to be most critical and very successful.

References

[1] P. Bahl and V. N. Padmanabhan, "RADAR: An In-Building RF-Based User Location and Tracking System," in *Proc. IEEE INFOCOM*, March 2000.

[2] W. K. Edwards, "Core Jini", Prentice-Hall Inc., 1999.

[3] E. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," CMU Technical Report CMU-CS-99-148, June 1999.

[4] E. Horvitz, A. Jacobs, and D. Hovel, "Attention-Sensitive Alerting," in *Proc. Conf. on Uncertainty and Artificial Intelligence (UAI'99)*, pp. 305-313, 1999.

[5] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software Rejuvenation: Analysis, Module and Applications," in *Proc. FTCS-25*, pp.381-390, June 1995.

[6] M. Roussopoulos, P. Maniatis, E. Swierk et al., "Personal-level Routing in the Mobile People Architecture," in *Proc. USENIX Symp. on Internet Technologies and Systems*, Oct 1999.

[7] B. Raman, R. H. Katz, and A. D. Joseph, "Universal Inbox: Providing Extensible Personal Mobility and Service Mobility in an Integrated Communcation Network," in *Workshop on Mobile Computing Systems and Applications*, Dec. 2000.

[8] D. Sturman, G. Banavar, and R. Strom, "Reflection in the Gryphon Message Brokering System," *Reflection Workshop at Object-Oriented Programming Languages and Applications*, 1998.

[9] Y. M. Wang, W. Russell, and A. Arora, "A Toolkit for Building Dependable and Extensible Home Networking Applications," in *Proc. USENIX Windows Systems Symp.*, pp. 101-112, Aug. 2000.