RZ 3353 (# 93399) 06/25/01
Computer Science 16 pages

Research Report

Intelligent Messaging for the Enterprise

Carl Binding, Stefan G. Hild, Reto Hermann and Andreas Schade

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

Intelligent Messaging for the Enterprise

Carl Binding, Stefan G. Hild, Reto Hermann and Andreas Schade

IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland
E-mail: {cbd,sgh,rhe,san}@zurich.ibm.com

Abstract

This paper illustrates the use of active database management systems combined with emerging "push" technologies to implement intelligent messaging for enterprises. The described solution allows highly personalized notification of individual customers upon occurrences of arbitrary data conditions; thus providing the foundations for personalized customer relationship management.
We describe related notification systems and how they compare with our work. The essential elements of active database technology are reviewed and an industry-standard "push" technology is discussed. These elements are combined into an intelligent messaging system which is illustrated by two examples.

1 Introduction

The phenomenal growth of mobile and personal communication systems has marked the past decade. In various European countries the penetration rate of mobile telephones exceeds the one for fixed, land-based telephone lines, and generally a 50% penetration rate can be assumed throughout Europe. Other geographies, most notably Asia and Oceania, are exhibiting similar growth rates and market penetration for mobile telephones which are—with the notable exception of Japan—based on the ETSI[1] specified Global System for Mobile Communication (GSM) standard as is the case for Europe [21]. North America has also started to catch up on usage of mobile telephony; its success has however been hampered by the use of various competing technology standards. However, GSM is now also finding its way into the USA and Canada. The Central and South American countries have also largely adopted the GSM standard. World-wide roughly 400 million mobile telephone handsets were sold in the year 2000, illustrating the pervasiveness of the mobile telephone as *the* mobile access device to IT infrastructure of choice.

Although GSM-based systems are still used primarily for voice communications, they are also increasingly used for data communications. One possibility is to use GSM's Circuit Switched Data (CSD) supporting the Point-to-Point Protocol (PPP) for IP traffic over a modem channel [13], thus enabling higher level protocols such as UDP [11] or the equivalent Wireless Data Protocol [37] and their applications.

The other, somewhat more rudimentary and currently more user-perceptible, packet-oriented data exchange mechanism in GSM is its Short Message System (SMS) [26, 16]. SMS messages can be sent from one mobile phone to another mobile phone[2] or from some access point into a Short Message Service Center (SMSC) to a mobile phone[3] and carry a payload of 140 octets. Depending on the short message's alphabet, these 140 octets translate into 160 7-bit symbols of the GSM default alphabet or into 70 UCS-2 16 bit symbols needed for non-latin alphabets [15].

Further growth in numbers and in the quality of the mobile telephony market is expected to continue. The following developments are imminent or can be anticipated soon:

- *WAP*: The *Wireless Application Protocol* [33, 36] standard has been—possibly somewhat prematurely—touted as the future "mobile internet". Although not the entire World Wide Web (WWW) [39] content is available or reachable through WAP, WAP indeed re-uses the underlying architectural principles of a browser-based "pull" model which has been tailored to match the restrictive capabilities of a mobile communication device and the wireless communication system. It thus uses a specific mark-up language and optimized communication protocols for the wireless infrastructure. (For a detailed discussion of the WAP technology see [29].)
  In addition, WAP also supports a specific "push" architecture which enables applications to send information asynchronously[4] from an IT environment towards the user's mobile device [35, 34].
  Although the marketing-driven exaggerations of WAP have led to disappointments with the early adopters of the technology, WAP's architectural principles will continue to evolve and the underlying wireless infrastructure will provide better suited services in terms of connection set-up time and bandwidth. Thus, although WAP in its current version still may undergo various technical changes, the implementation of a browser-

[1] European Telecommunication Standardization Institute
[2] This is termed *mobile originated*.
[3] A so-called *mobile terminated* communication.
[4] As seen from the end-user's perspective.

based "pull" application model with scripting extensions for mobile telephones will remain.

- *i-mode*: NTT DoCoMo's *i-mode* [12, 22]—a proprietary technology—is similar to WAP but based on different underlying technologies. The principal differences are the mark-up language and the underlying mobile network technology.
  i-mode uses a modified version of the Web's HTML [28]: *compact HTML* (c-HTML) [32], which is a super- and subset of HTML, version 3. Hence, existing WWW HTML-based applications must still be translated into c-HTML before being used by i-mode terminals. From an application provider's point of view, an application must thus be re-engineered before becoming suitable for i-mode. Similar to a WAP application, it is not only the individual application panels (or screens) that have to be adapted to the mobile devices' limited screen sizes, but also the interactive transactions of the applications have be adapted to their usage over a mobile system.
  The underlying mobile network technology of i-mode is based on the Personal Digital Cellular Packet (PDC-P) packet-switched data network, a variant of time-division multiple access (TDMA) technology, which is mainly deployed in Japan. Unlike GSM CSD, PDC-P avoids lengthy connection establishment for i-mode applications. The available bandwidth compares favorably with the currently most commonly deployed WAP implementation over GSM CSD: PDC-P can reach a transmission rate of up to 28.8 kps compared with an average of 9.6 kps for GSM CSD.
- GPRS: The emerging Generalized Packet Radio System (GPRS) [6] is being deployed by various European telecommunication companies, and end-user handsets are emerging in the market.
  Unlike GSM CSD, GPRS will support Internet Protocol (IP) connectivity over a packet-oriented bearer, thus not only eliminating the circuit set-up time of GSM CSD but also increasing the available bandwidth. The envisioned bandwidths are expected to average 40 kbits/s per user, with a maximum data rate of 171.2 kbits/s for a single user within a GPRS cell.
- UMTS: Somewhat further out on the time horizon is the availability of Universal Mobile Telecommunication System (UMTS), which will probably be deployed around 2003 and promises even more increased bandwidths for packet-oriented IP data traffic.

We thus assume that mobile telephones and other mobile communication devices, such as personal digital assistants, will increase their computing power because of technological hardware progress on the one hand and better communication capabilities due to improved mobile communication systems on the other hand.

As discussed above, application architectures for such devices will continue to use the "pull" paradigm examplified by most internet applications. Furthermore, and this is of large consequences, applications can also "push" content asynchronously onto the end-user's mobile device, thereby enabling application providers to be continuously in touch with their end-users. Such content must not be limited to text and graphics, other interaction media such as voice and audio will become integrated into the overall, multi-modal interfaces for such devices in the future.

Examples for "push" applications are manifold:

1. *Finances*: Customers can be contacted by their financial service provider on the occurrence of a particular situation with their personal finances. Be it alerts on interesting arbitrage conditions for the sophisticated investor to more mundane notifications about bills due for the main-street banking customer: the financial service provider can reach a large number of its customers using widely available and thus also affordable technology.

2. *Travel*: An airline may alert its customers about delayed or cancelled flights. Using the combined "push" and "pull" technology, the airline can offer travel alternatives for customers whose flights have been delayed or cancelled to book another flight.
   Less individualized travel services, such as railways, will continue to be servicing their mass transit customers through "pull" services, but may use "push"-based applications to alert their operating personnel.
3. *Entertainment*: Registered user groups can be notified on last-minute deals on remaining seats at concerts or shows. Customers can be notified about upcoming events with the immediate possibility of making reservation using a "pull"-based application.
   Last-minute advice on, for example, traffic information around the venue of a concert or a show can be individually sent to registered users, thereby creating a highly individualized customer relation between the entertainment enterprise and its customers.

One can expand such examples to other application areas. Fundamentally we notice that the possibility of sending individualized alerts to a potentially large customer group based on some specific occurrence of events can open up new applications and extend the reach of enterprises towards their customers.

It is thus of interest to investigate how such systems can possibly be implemented given available technologies—be it as research systems or as available IT products. In the remainder of this paper, we shall take a closer look at some system research in the area of notification or *content-based* addressing as well as *event-modelling* languages (section 2). We then introduce commercially available active database technology (section 3) and "push"-oriented communication standards and systems (section 4). Section 5 illustrates how these technologies can be combined to provide semantically very powerful, intelligent messaging systems, and discusses early experiences with our prototype solution. Further examples are given in section 6. Finally, section 7 concludes the paper with a comparison between our work and the research discussed in section 2.

2 Related Work

Various bodies of work related to messaging are described in the literature. This section reviews several of these systems and attempts to capture their aims and salient features.

One of the earliest messaging systems is described in [25] and exhibits similarities to even earlier work in subject-based publish/subscribe systems [7]. It is a *subscription-based* messaging environment in which clients express interest in information published under some subject name. The message contents published under the subject name is relatively open and self-describing. This *information bus* architecture has been turned into a commercial product used in financial trading/brokerage firms and manufacturing plants to disseminate updates for named information items [31]. For example, in a financial setting, modifications of stock prices or currency exchange rates are published under some commonly agreed upon name of a commodity and are thus made available to individual traders' workstations.

Such subject-based messaging systems are easy to use but require a commonly agreed upon naming scheme for messages. Their underlying model of operation targets a one-to-many relationship for individual messages: a large audience subscribes to identical subjects, although quasi-private subjects can also be published and subscribed in these systems. One challenge in these systems is to provide efficient routing between publisher and subscribers. In practice, one feasible approach is to establish fixed topology networks along which the identified messages are disseminated based on established subscriptions. Thus, the routing is essentially fixed and requires a fixed network infrastructure with message delivery over a given route based on the effectively active subscriptions.

The concept of subscription-based messaging has been taken one step further towards *content-based* messaging. Here the subscriber formulates a logical condition on the data content which is disseminated over the communication substrate, and it is the responsibility of the communication infrastructure to filter and forward only the actively subscribed information items based on their content.

Two systems have been described in the literature. The Gryphon system [1–3] addresses the issue of optimizing the routing of content-based messages. It uses a combination of the network topology and the content-based subscriptions to determine which messages are routed over which links in the network and thus to deliver a message efficiently from a publisher to all subscribers interested in a given message.

The formalism to specify the desired content of a message is based on a logical conjunction of predicates on data values present in the information. Predicates take the form of equality to specific values or a wildcard. Each subscription is representable as a path through a *parallel search tree* (PST) where data is injected at the root and trickles down data-matching paths. Taking the network topology—given as a minimum spanning tree—into account, the PST is annotated with information to indicate whether a given message needs to be forwarded over some outbound network link. This decision can be reached while traversing the PST and not only when reaching the leaf nodes of the PST: outbound links for which no active PST nodes remain can be ignored.

The key advantage of Gryphon is the evaluation of common subscription prefixes in the PST as well as the grouping of subscriptions onto links from the routing tables which are based on a minimum spanning tree of the network. Among its shortcomings, we note the content-matching formalism which is limited to equalities and "don't cares". Second, Gryphon assumes both a relatively stable network topology as well as a stable set of subscriptions which are used to build the PST. Thirdly, the routing optimization assumes a hierarchical network topology which does not degenerate into a flattened tree. In practice, however, personal messaging systems do not generally exhibit their internal routing topologies and thus map onto a very flat hierarchy with one, logically centralized, messaging center forwarding messages to a large number of destinations. A last argument can be made for the specificity of messages: the degree of customization determines if some commonality in the PST exists or if the subscribers' branches are totally disjoint. It depends on the application area to assess the variance in subscriptions.

The Siena system [8, 9] provides a somewhat richer formalism to specify a notification's data matching condition, where a notification is defined by a condition over a set of typed attribute–value pairs. The filtering condition is expressed as a conjunction over operators on attributes which include the common equality and ordering relations ($=, \neq, <, >$ etc.) and some more type-specific relations such as substring, string prefix and suffix as well as a wildcard *any* value.

The Siena work concentrates on the discovery of *covering* filters: data events that match one filtering condition may also satisfy a more specific filtering condition. The routing is then based on the covering relationship in as far as for each filter a set of subscribing nodes is maintained. Data is then processed in a top-down fashion from most generic to most specific filters—or in Siena terms across the partially ordered set of filters—and forwarded to network nodes from which a matching filtering condition originated.

The algorithm is distributed as the filters are propagated to neighboring nodes in the network. Each routing node then maintains the partially ordered set of filtering conditions, inserting new filters into the covering hierarchy as necessary. In that respect, Siena differs from Gryphon, which takes a more centralized approach to the maintenance of the PST. Another difference to Gryphon is that the covering relationship is less stringent than exact matching of common subexpressions: a condition $(x > 3 \wedge x < 7)$ covers $(x = 5)$ without

being syntactically equal. The details on how such covering relations are decided are not given; intuitively however it appears as a computationally complex, if not undecidable, problem.

Whereas the above systems concentrate on the optimal routing of notifications, the actual monitoring of some dataspace is considered in the work on *continual queries* [19, 20]. The OpenCQ system provides push-enabled, event-driven, content-sensitive information delivery capabilities and combines "pull" and "push" services in a unified framework. An event is defined as a modification to some data value in the dataspace under observation, and content sensitivity consists in the ability to express some specific condition on the data universe that needs to be satisfied for the event to occur. OpenCQ borrows the concepts of earlier work on active databases [38], but extends it with the ability to send a notification once some event has occurred. In addition to data-modifying events, time-based events can also be defined[5].

The CORBA Event Service is part of CORBA's Common Object Service Specification [23] and provides reliable communication of asynchronous events. The service is defined in a scalable manner; there is no central server avoiding single points of failure. It allows using abstract as well as specifically typed event models.

The CORBA Event Service decouples *supplier* objects sending notification messages when an event occurs from *consumer* objects interested in receiving these notifications. Based on these roles, the service defines interfaces which have to be implemented by suppliers and consumers. The pairwise existence of these interfaces allows the creation of *event channels* through which multiple suppliers can be connected to multiple consumers (thus enabling *many-to-many communication*). In essence, the applications build up a network of suppliers and consumers which are interconnected through the event channels. Thus, there is no implicit routing based on content or addresses: messages flow along the logical network of CORBA objects.

The CORBA Event Service supports two communication models and two data-transfer models which can be arbitrarily combined. Events are communicated either

- using the *push* model, in which the supplier initiates the event communication, or
- using the *pull* model, in which the consumer actively requests the event communication.

The data-transfer models are

- the *generic model*, in which events are commonly represented using an unspecific data type (`any`), or
- the *typed model*, in which events respresented as specific data types are communicated using dedicated methods specified using CORRBA's Interface Definition Language (IDL).

The CORBA Notification Service Specification [24] is based on the Event Service and enhances it by introducing the concepts of event filtering and configurablity of quality-of-service requirements. In addition to the data-transfer schemes of using untyped or typed events, the notification service introduces *structured* events which provide well-defined data structures into which a variety of events can be mapped. Structured events can be transmitted directly between suppliers and consumers without the need of any data marshalling. Structured events mainly consist of a set of name–value pairs (with some standard names and string values) and thus are a compromise between untyped and strongly typed CORBA events.

[5] Evidently however one can also consider a clock as some continuously updated data value (see also section 5).

The Notification Service defines *notification channels* at which clients can specify the events they are interested in receiving. This is done by associating filter objects encapsulating the consumer's constraints with proxies through which clients communicate with event channels. A default filter-constraint language for expressing filter conditions is provided. It uses a naming scheme to identify structured events to which the filtering constraint is to be applied. The constraint language is based on CORBA's *Trader Constraint Language*. Subscriptions for specific events can be shared between channel and clients. The channel can also map between untyped, typed, and structured events to provide backward compatibility with ordinary CORBA event service clients.

Additionally, the notification service allows discovery of event types required by all consumers of a channel from some supplier. Conversely, consumers can detect which events are offered by some supplier. Thus, suppliers can avoid producing notifications no consumer is interested in, and consumers can subscribe to new notification types as they become available. An optional event type repository contains information about all known event types, which allows consumers to construct meaningful constraints and enables type and validity checking of these constraints.

3 Active Databases

Research on active databases is described in [38]. Such systems allow the definition of *triggers*, which are associated with database relations and are executed upon the modification of some data contained in a particular relation, the insertion of new data into a relation, or the deletion of data from a relation. After or before the data modification, some action is executed when an (optional) data-matching criterion is met. For example, one may define a trigger to execute when an update to table T occurs and when the new or old data has a certain value. The action associated with the trigger can be an arbitrary SQL expression.

Commercial database systems, such as IBM's DB2 Universal Database (UDB) [17], have incorporated the concepts of active databases and support the creation of triggers. As an example of a trigger, consider the following SQL code:

```
CREATE TRIGGER RAISE_LIMIT
AFTER UPDATE OF SALARY ON EMPLOYEE
REFERENCING NEW AS N OLD AS O
FOR EACH ROW
WHEN ( N.SALARY > 1.1 * O.SALARY)
BEGIN ATOMIC
  INSERT INTO WINNERS
  VALUES ( N.FIRST_NAME, N.LAST_NAME, N.SALARY)
END
```

This code creates a trigger which is executed after updates to the column *SALARY* of table *EMPLOYEE*. For rows in which the old and new value of *SALARY* differ by more than 10%, an insertion of the employee's data is performed into the table *WINNERS* which eventually collects the employees with salary increases which are greater than 10%.

The above SQL fragment achieves this by first creating aliases for the old and new values of table *EMPLOYEE*, which are referred to as *O* and *N*, respectively.

The triggering condition is then given in the *WHEN* clause, which relates the old and new salary values. When the condition is met, some atomic action is performed. This action can take the form of an SQL *full-select* statement such as an *INSERT*, a *SELECT*, or an *UPDATE* statement. Here, an *INSERT* statement is executed: the data of table N (i.e. which contains the new values) is inserted into yet another table, *WINNERS*. Note that the execution of the atomic action can cause other, cascaded, triggers to be executed[6].

[6] There may be some system-specific limitation on the number of cascaded triggers.

During the execution of the trigger's body it can invoke a so-called *user-defined function*[7], which are arbitrary functions written in a conventional, third-generation programming language such as C or Java. A parameter-passing convention maps SQL data types into programming language data types. Another possibility for user-defined functions is to use an extended SQL dialect which is similar to a third-generation programming language, i.e. which provides scalar data types, if-then-else statements, looping constructs, etc.

The combination of triggers with user-defined functions allows the implementation of a notification service in which the trigger's body invokes a user-defined function to send a notification. The parameters of this function, such as a destination address, the message body, etc., can be extracted from the database itself and thus can be adapted to the stored data values and the business processes supported by the various data models. The details of this approach are given in section 5.

4 Push Technology

A data *push* is distinguished from a data *pull* in as far as it is the data source that actively forwards data to some destination. In most traditional applications, including internet-based systems, data is pulled from some IT infrastructure based on a user's or an application's request. There are a few application areas in which data is pushed from a data source to some destination, and most often this is done to distribute data to a set of data subscribers. Some such publish-subscribe systems have been described in section 2. Here we give a more detailed description of one, newly emerging, push technology and its architecture, namely the WAP Push Access Protocol (PAP) [34, 35], which can be used to not only target WAP-compliant mobile stations, but also older-generation GSM phones implementing the SMS standard or traditional e-mail services.

PAP is a classical request-response protocol; the *push-initiator* (PI) initiates a push request structured as a multi-part MIME document [18] composed of

1. a *control header*: it specifies a unique push-id, one or multiple target addresses of the message's body, possibly constrains the delivery time of the message, and optionally indicates a service delivery quality of service. The latter supports selection of a particular network as well as the delivery method, which can essentially be *confirmed* or *unconfirmed*. In the confirmed case, the target device is responsible for acknowledging reception of the pushed content.
   The control header also allows the push initiator to request notification regarding the final outcome of the message delivery. The notification is sent to the indicated URL as an HTTP POST message [4, 5, 14].
2. a *message body*: this is the actual content pushed to the target device. In the WAP context it takes the form of some WAP-application-specific content, i.e. a Wireless Markup Language (WML) document, a WML Script, a WAP service indication, or a WAP service loading document. For SMS and e-mail delivery, plain text messages are forwarded to the destination addresses.
3. a *client device capability* description: this optional part of the push request message allows indicating the minimum requirements of the target devices that are required for the push to take place. For example, the push initiator can request that the target device have a certain screen size.

The push initiator emits its requests to the Push Proxy Gateway (PPG). The PPG server validates the request, replies with a response, and controls further handling of the

[7] In other database systems, these features are sometimes called *external procedures*.

push request. This includes possible delaying of the delivery if the *deliver-after* option has been used. Once the message is ready to be delivered, the appropriate bearer is selected, various possible message transformations are performed, and message delivery state is maintained based on status reports from the delivering bearer.

Figure 1 illustrates the push system architecture. The push initiator emits push requests using the HTTP POST method and sends the MIME multi-part body, described above, to PPG.

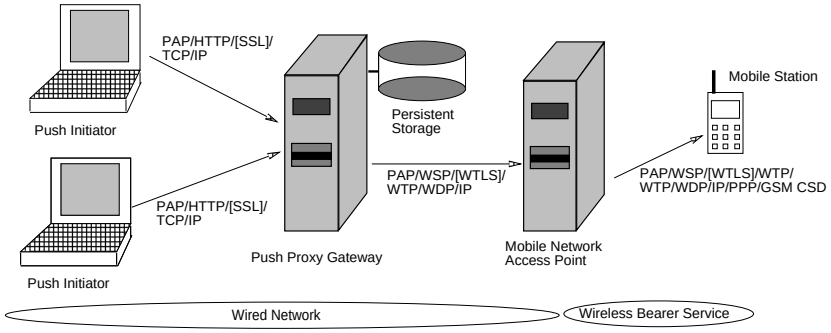

Fig. 1. Push Architecture

PPG forwards messages over a set of bearers. For pure WAP content, any of the WDP bearers can be selected, such as UDP/IP, GSM CSD, or GSM GPRS [37]. For straight SMS or SMTP traffic, special access protocols are used to interface to a SMSC or a mail-transferring agent (MTA), respectively. For the former, dedicated protocols such as EMI/UCP or SMPP [10, 30] are used; e-mail is forwarded using the SMTP protocol [27].

The push request is the principal class of requests originating from a push initiator. Other requests include

- *status query*: the push initiator or any other client can inquire the delivery status of a given push request, identified through its (unique) push-id.
- *cancel request*: the push initiator or any other client can cancel a push request that awaits delivery because of to a *deliver-after* value in the future.
- *result notification*: when submitting the push-request, the push initiator can indicate a target to be notified about the final outcome of the message delivery[8]. The result notification is eventually sent from the PPG to the notification target once the final state of the push-request is reached.
- *client device capability query*: push requests allow the indication of the client device's capabilities. In order to query capabilities of a particular device, a PPG client can issue this request to receive the client device capability description as a response.

One of the roles of PPG is to shield the push-initiating application from the idiosyncrasies of the various bearer services and to provide a more abstract interface to a push infrastructure. Hence, the push-initiating application does not have access to the network topology. Furthermore, PPG instances also are shielded from the underlying network topologies: the underlying networking infrastructure is opaque to PPG. Thus content-based routing schemes similar to the schemes described in section 2 do not work in a push architecture as described here; nor would they work in a GSM SMS only infrastructure which has a similar architecture. Any data-related logic triggering notifications and the routing of

[8] Note that this must not necessarily be the original push initiator.

such notifications are completely distinct and, except for special purpose applications that justify large dedicated infrastructures welding together data evaluation with the network topology, such investments appear not viable in practice.

5 Intelligent Messaging for the Enterprise

In the preceding sections we have described several research projects aimed at optimizing content-based routing (section 2). Existing active database technology, in particular the triggering and external function mechanisms, were discussed in section 3. Section 4 has introduced features of an emerging, commercially available messaging system. This section describes how these technologies can be brought together to implement intelligent messaging for an enterprise.

We conjecture that many enterprises are already using relational database technologies. Although not all of the enterprise's data may be held in a single data warehouse, subsets thereof can be made available through a relational database. This enables use of the triggering mechanism to implement distinct business application logic which causes sending of notifications to specific target customer groups. In essence, a database trigger is executed upon modification of the underlying data universe and filters out relevant data conditions which are then signalled to some intended customer group. The signalling is achieved by invoking a notification service through a user-defined function called from the trigger. Figure 2 illustrates the underlying architecture.

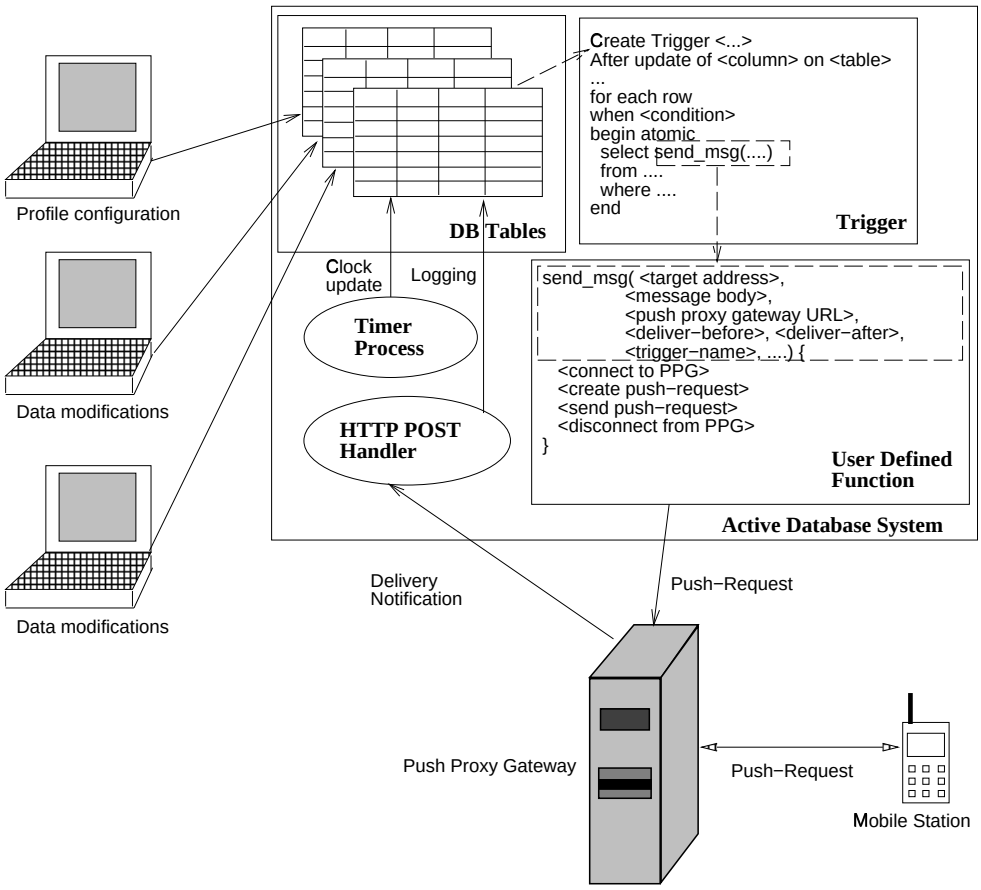

Fig. 2. Intelligent Messaging System Architecture

The central subsystem is the active database system. It holds the business and customer data (*DB Tables*), the triggers which detect relevant conditions on that data (*Trigger*), and a user-defined, external, function (*User Defined Function*) to invoke the notification service. The notification service is implemented via the WAP Push Access Protocol architecture discussed in section 4.

Two further software entities are associated with the database environment:

1. *Timer process*: this process updates a *CLOCK* relation to trigger periodic notifications; the details to handle such periodic notifications are given below.
2. *Status tracking*: the WAP Push Access Protocol (PAP) supports notifications to an arbitrary software entity—indicated through some URL—about the outcome of a given message delivery. Some HTTP POST method handling software entity, labelled *HTTP Post Handler* in figure 2, can thus update the database to log the outcome of specific push requests. Further reporting tools may then access information stored in the database.

Distinct systems are used to configure the user profile and manage triggers and user-defined function (labelled *Profile configuration*) as well as to insert, update, or delete business data of particular applications (labelled *Data modifications*).

The invocation of the messaging system occurs from a database trigger. The following code fragment illustrates the approach:

```
CREATE TRIGGER RAISE_LIMIT
AFTER UPDATE OF SALARY ON EMPLOYEE
REFERENCING NEW AS N_ROW OLD AS O_ROW
FOR EACH ROW
WHEN ( N_ROW.SALARY > 1.1 * O_ROW.SALARY)
BEGIN ATOMIC
  SELECT SEND_MSG( <address>, <message body>)
  FROM EMPLOYEE
  WHERE EMPLOYEE.ID = N_ROW.ID;
END
```

The above trigger notifies the lucky employees when their salary increases by 10%. Note that the trigger's atomic action is performed *after* the update of the *salary* relation occurs. The selection of the particular employee is done through an *equi-join* between the updated row's *N_ROW.ID* field with the relation *EMPLOYEE*[9]. The invocation of the *SEND_MSG* user-defined function occurs as a side effect of the *SELECT* statement executed once the trigger condition has matched: for each selected data tuple, this function is executed. (Note that this represents normal SQL *SELECT* statement behavior.)

Not shown in the above call to the user-defined function *SEND_MSG* is the value of the address and the value of the message body. In order to tailor the notification to the particular employee, we can introduce further user-defined functions. For example, function *PREF_EMP_ADDR* would return the notification address of a given employee. This function would depend on the employee's various addresses, such as e-mail or mobile telephone number, and may include further situation-dependent data to select the address to use. The same approach can be used for a function *SAL_INC_MSG* which creates the body of the notification. It can be customized using the employees first and last name, his or her gender, his or her preferred language etc. Important to point out is that *all* the data regarding the employee is already stored in the database and thus any conceivable personalization of the notification can be provided using this data.

In section 6 we provide two concrete examples of triggers for a travel-related notification system as well as a financial environment.

[9] We assume that *EMPLOYEE* is indexed on column *ID*.

Despite the elegance of the combination between relational database logic and a messaging infrastructure, a few further points warrant discussion.

The first such issue is the handling of time. Triggers are only executed upon modification of data, which does not immediately provide a means to implement periodic triggers. One simple approach however is to create an auxiliary relation *CLOCK* which is updated by an external process. Updates to the *CLOCK* relation then trigger actions with variant periods. Assume that relation *CLOCK* has attributes *TIME* to denote the currently valid time, *PERIOD* indicating the periodicity of the timer, and an arbitrary *NAME* which could be used to refer to such a timer[10]. The following trigger can then be defined:

```
CREATE TRIGGER PERIODIC
AFTER UPDATE OF TIME ON CLOCK
REFERENCING NEW AS N_ROW OLD AS O_ROW
FOR EACH ROW
WHEN ( MOD( N_ROW.TIME, N_ROW.PERIOD) = 0)
BEGIN ATOMIC
  UPDATE MESSAGE SET DELIVER = 1 WHERE DELIVER_PERIOD = N_ROW.PERIOD;
END
```

The above assumes that the data-dependent trigger creating the notification data does not immediately send out said notification, but instead inserts the notification into a *MESSAGE* relation which acts as an intermediate buffer. An *UPDATE* trigger on relation *MESSAGE*'s column *DELIVER* then invokes the *SEND_MSG* user-defined function as in the previous example.

One beneficial side effect of the intermediary relation *MESSAGE* is the possible elimination of overly repetitive notifications. When, for example, some rapidly changing data would cause annoyingly repetitive messages to a customer, the *MESSAGE* relation can be used to update an existing message to a given customer rather than creating repetitive messages: The data-dependent trigger invokes a user-defined function which tests whether a message for the target address already exist and, if none exists, inserts the message. Alternatively, the existing message body is updated.

Further time-dependent data conditions can be expressed using SQL's *TIMESTAMP* arithmetic.

A second issue in implementing production-grade messaging infrastructures clearly is the growth of data volumes in situations where events are stored in a relation. The approach to automatically manage such growth can be based on triggers, possibly combined with timestamps. Consider the above example: various data-dependent conditions generate messages which are sent periodically. These messages can be kept for a certain time period, and a time-dependent trigger can be used to erase all messages that were sent but are older than some given duration.

6 Examples

This section illustrates our approach further through the use of examples. The first example is a rudimentary flight information broadcasting system. It is based on the following relations:

1. *FLIGHT*: contains flight-relevant information including the flight number, the estimated departure time (EDT), the flight's leaving gate, its status, origin, destination etc.

[10] The *name* is not used in this example.

2. *PASSENGER*: holds information on passengers including personal data such as name, first name, preferred language, mobile phone number, e-mail address, preferred telecommunications provider, etc. and a unique passenger identity.
3. *BOOKINGS*: is a relation to capture which passengers are booked on which flights. It relates the passenger identity with a flight number on a given date.

We assume that modifications to a flight cause updates to the *FLIGHT* relation. Thus, we create a trigger which detects such changes, selects all the passengers booked on the particular flight and sends out notifications to this set of passengers. The message body is individualized based on passenger-specific information such as the preferred language, last name, title, etc.

The following SQL code represents the needed trigger[11]:

```
CREATE TRIGGER db2inst1.notify_passenger
AFTER UPDATE OF EDT, GATE, status ON FLIGHT
REFERENCING NEW AS N_ROW OLD AS O_ROW
FOR EACH ROW MODE DB2SQL
WHEN ((N_ROW.EDT <> O_ROW.EDT) OR (N_ROW.GATE <> O_ROW.GATE) OR
      (N_ROW.STATUS = 'cancelled'))
BEGIN ATOMIC
  SELECT
    SEND_MSG(
      ( 'wappush=' || PASSENGER.MOBILE_PHONE_NBR ||
        COALESCE( CONCAT( '/smsc=', PASSENGER.SMSC_NAME), '') ||
        '/type=sms@rigi.zurich.ibm.com' ),
      GET_MSG_TXT( PASSENGER.TITLE, PASSENGER.LAST_NAME,
                   PASSENGER.LANG, N_ROW.NUMBER,
                   FLIGHT.DESTINATION,
                   CAST( TIME(N_ROW.EDT) AS VARCHAR( 8)),
                   N_ROW.GATE),
      'http://rigi.zurich.ibm.com:13131',
      '', '', '', 'db2inst1.notify_passenger trigger', '', 0)
  FROM PASSENGER, BOOKING, FLIGHT
  WHERE ((BOOKING.FLIGHT_NBR = N_ROW.NUMBER) AND
        (BOOKING.DATE = DATE( N_ROW.EDT)) AND
        (BOOKING.PAX_ID = PASSENGER.PAX_ID) AND
        (FLIGHT.NUMBER = BOOKING.FLIGHT_NBR));
END
```

The above code snippet invokes the user-defined function *SEND_MSG* which interfaces to the WAP PPG. Thus the address is formatted accordingly with the PAP client address format. As customer-specific information, it includes not only the customer's mobile phone number, but also the preferred telco's SMSC, if any.

The message body is composed by the *GET_MSG_TXT* user-defined function which combines customer-specific information with the details of the flight information. The routine would return a language-specific greeting, specifically tailored for a customer, such as *"Dear Mr. ..., your flight ... is now parting from gate ... at ..."* (assuming the preferred language is English!).

Another example of triggered notifications is based on financial data. Assume a relation *PRICE* which contains ticker information such as stock name, price, and time. We now desire to be notified when the price of the IBM share is 5% higher than the last price. As we assume a time series of prices, we need to select the last one and compare it with the newly inserted value. The trigger thus becomes:

```
CREATE TRIGGER five_pct_change
NO CASCADE BEFORE INSERT ON PRICE
```

[11] We have only deleted some *RTRIM* calls for clarity purposes.

```
REFERENCING NEW AS N_ROW
FOR EACH ROW MODE DB2SQL
WHEN ( N_ROW.SYMBOL = 'IBM')
BEGIN ATOMIC
  SELECT
    SEND_MSG( <address>, <msg body>,
              'http://rigi.zurich.ibm.com:13131',
              '', '', '', 'five_pct_change', '', 0)
  FROM PRICE,
       TABLE (SELECT MAX( TIME) AS LAST_TICK
              FROM PRICE WHERE PRICE.SYMBOL = 'IBM') AS LATEST
  WHERE
    ( N_ROW.VALUE >= 1.05 * PRICE.VALUE)
    AND ( PRICE.SYMBOL = 'IBM')
    AND ( PRICE.TIME = LATEST.LAST_TICK) ;
END
```

where the auxiliary *SELECT* statement creates an intermediary table named *LATEST* which contains the timestamp with the highest value, i.e. the last recorded ticker value's timestamp. (This table has exactly one column, named *LAST_TICK* and one row.)

Once we know the timestamp of the last ticker value we can relate the share's price valid at that point in time with the new price (which is about to be inserted) referred to a *N_ROW*. The data selection then filters out the new ticker entry and the next-to-last ticker entry if the prices differ by more than 5%.

Note that the trigger is evaluated *before* the new price is inserted. Not shown in the above code fragment is the code necessary to select target addresses and to generate the notification body. Using SQL expressions or a user-defined function would be the adequate solution for address selection and message body generation.

Another example is a notification when the latest price of the IBM share is 5% above some average price over some past period. The sample code is:

```
CREATE TRIGGER avg_change
AFTER INSERT ON PRICE
REFERENCING NEW AS N_ROW
FOR EACH ROW MODE DB2SQL
WHEN ( N_ROW.SYMBOL = 'IBM')
BEGIN ATOMIC
  SELECT
    SEND_MSG( <address>, <msg body>,
              'http://rigi.zurich.ibm.com:13131',
              '', '', '', 'avg_change', '', 0)
  FROM PRICE,
    TABLE ( SELECT AVG( VALUE) AS AVG_VAL
            FROM PRICE
            WHERE (PRICE.SYMBOL = 'IBM') AND
                  (PRICE.TIME BETWEEN
                     ( CURRENT TIMESTAMP - <n> DAYS - <m> DAYS) AND
                     ( CURRENT TIMESTAMP - <n> DAYS))) AS AVERAGE
  WHERE
    ( N_ROW.VALUE >= 1.05 * AVERAGE.AVG_VAL) AND
    ( N_ROW.VALUE = PRICE.VALUE) AND
    ( PRICE.SYMBOL = 'IBM') ;
END
```

Here we first compute the average price over a period of m days which ends n days in the past. This price is computed into the auxiliary table *AVERAGE* and used in the *WHERE* clause of the *SELECT* statement. Again we have omitted details on the generation of the target address and the notification body. Similarly, we have not shown how to fetch the values for m and n, but these evidently can depend on a given end-user's identity.

7 Conclusion

We have reviewed several state-of-the-art content-based addressing systems. Our conclusion on these systems is that their practicality is limited in currently deployed IT and telecommunication infrastructures: a strict separation of data filtering functionality from available routing infrastructure makes it virtually impossible to implement content-based routing effectively. In addition, some of these systems suffer from an overly simplistic data-filtering formalism.

The work on continuous queries is similar to our work, which is, however, based on commercially available IT middleware and does not require its own data querying infrastructure. The most notable difference between the work described in [19, 20] and ours is the (syntactic) inclusion of periodic triggers in the query language. Our approach requires an additional timer process to generate database updates which are then used in conjunction with data-dependent triggers.

CORBA event services and notification services specify filters on a per-event occurrence level; there is no data matching over a larger data set which can be done using a relational database environment. Furthermore, CORBA requires notification consumers and suppliers to explicitly build up a network of CORBA supplier, consumer and channel objects for a data flow to occur. This uniform model does not easily span multiple, non-CORBA-based communication infrastructures such as the commonly deployed mobile messaging environments.

We have given a few examples to illustrate our approach. First performance measurements indicate an approximative performance of 15 unconfirmed push requests per second on a UNIX workstation[12]. The limiting factor is the interprocess communication and I/O activity of the processes involved, including the persisting of the push requests passing through the push proxy gateway.

We strongly believe that the approach chosen is viable in practice and that it enables novel applications using push technology. The advantage of using a commercial active database environment to cause notifications is a tradeoff against raw performance of custom-designed notification environments, which can only be deployed for specific setups. The availability of *all* relevant enterprise data within a relational data model combined with commercially available push technology enables cost-efficient and timely deployment of new applications, enabling highly individualized customer relationship management applications.

8 Acknowledgment

The example discussed in this paper has been inspired by the needs of Swissair, the major Swiss airline. Careful reviewing of this contribution was done by our colleagues at IBM Research, François Dolivo and Douglas Dykeman.

References

1. Marcos K. Auguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching Events in a Content Based Subscription System. In *Principles of Distributed Computing Systems*. ACM, 1999.
2. Guruduth Banavar, Thushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. In *International Conference On Distributed Computing Systems*, 1999.

[12] We are using an IBM RS/6000 43P model 150.

3. Guruduth Banavar, Marc Kaplan, Kelly Shaw, Rober E. Strom, Daniel C. Sturman, and Wei Tao. Information Flow Based Event Distribution Middleware. In *Middleware Workshop at the International Conference on Distributed Computing Systems*, 1999.
4. T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax and Semantics*. IETF Networking Group, August 1998. RFC 2396.
5. T. Berners-Lee, L. Masinter, and M. McCahill. *Uniform Resource Locators (URL)*. IETF Networking Group, December 1994. RFC 1738.
6. Christian Bettstetter, Hans-Jörg Vögel, and Jörg Eberspächer. GMS Phase 2+ General Packet Radio Service (GPRS): Architecture, Protocols, and Air Interface. *IEEE Communications Surveys*, 2(3), 1999.
7. Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, April 1989.
8. Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design of a Scalable Event Notification Service: Interface and Architecture. Technical Report Technical Report CU-CS-863-98, Dept. of Computer Science, University of Colorado, August 1998.
9. Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Content-Based Adressing and Routing: A General Model and its Application. Technical Report Technical Report CU-CS-902-00, Dept. of Computer Science, University of Colorado, January 2000.
10. CMG Telecommunications & Utilities BV. *Short Message Service Centre: External Machine Interface*, January 1999.
11. Douglas E. Comer. *Internetworking with TCP/IP: Principles, Protocols, and Aarchitectures*. Prentice-Hall, 2000.
12. Kei-Ichi Enoki. i-mode: The Mobile Internet Service of the 21st Century. In *Proceedings 2001 IEEE International Solid-State Circuits Conference*, pages 12–15. IEEE, 2001.
13. Robert Fielding et al. *Point-to-Point Protocol (PPP)*. IETF Networking Group, July 1994. RFC 1548.
14. Robert Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. IETF Networking Group, January 1997. RFC 2068.
15. European Telecommunications Standard Institute. *Digital cellular telecommunications system (Phase 2+): Alphabets and Language-specific Information*, Version 7.2.0 edition, December 1998.
16. European Telecommunications Standard Institute. *Digital cellular telecommunications system (Phase 2+): Technical Realization of the Short Message Service (SMS), Point-to-Point (PP)*, Version 5.8.1 edition, December 1998.
17. International Business Machines Corp. *IBM DB2 Universal Database: Application Development Guide*, Version 7, 2000.
18. E. Levinson. *The MIME Multipart/Related Content-Type*. IETF Networking Group, August 1998. RFC 2387.
19. Ling Liu, Calton Pu, and W. Tang. Continual Queries for Internet-Scale Event-Driven Information Delivery. *IEEE Knowledge and Data Engineering*, 11(4):610–628, July 1999.
20. Ling Liu, Calton Pu, and W. Tang. WebCQ - Detecting and Delivering Information Changes on the Web. In *Proceedings of the Ninth International Conference on Information Knowledge Management (CIKM 2000)*, pages 512–519. ACM, 2000.
21. Michel Mouly and Marie-Bernadette Pautet. *The GSM System for Mobile Communications*. Mouly & Pautet, 1992.
22. NTT Mobile Communications Network, Inc. Docomo's i-mode. http://www.ietf.org/proceedings/00mar/slides/plenary-imode-00mar, March 2000.
23. Object Management Group. *Event Service Specification*, June 2000. OMG Document 00-06-15.
24. Object Management Group. *Notification Service Specification*, June 2000. OMG Document 00-06-20.
25. Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 58–68. ACM, 1993.
26. Guillaume Peersman, Srba Cvetkovic, Paul Griffiths, and Hugh Spear. The Global System for Mobile Communications Short Message Service. *IEEE Personal Communications*, pages 15–23, June 2000.
27. Jonathan B. Postel. *Simple Mail Transfer Protocol (SMTP)*. IETF Networking Group, August 1982. RFC 821.
28. Dave Raggett, Arnaud Le Hors, and Ian Jacobs. *HTML 4.0 Specification*. Worldwide Web Consortium (W3C), April 1998.
29. Sandeep Singhal, Thomas Bridgman, Stefan Hild, Jari Alvinen, Lalitha Suyranarayana, Jim Chan, and David Bevis. *The Wireless Application Protocol: Writing Applications for the Mobile Internet*. Addison Wesley Professional, 2000.
30. SMPP Developer Forum. *Short Message Peer to Peer Protocol Specification v3.4*, January 1999.
31. Tibco: Infrastructure for real-time e-business. WWW Site, 2001. http://www.tibco.com.

32. W3C. *Compact HTML for Small Information Appliances*, February 1998. http://www.w3.org/TR/1998/NOTE-compactHTML-19980209.
33. WAP Forum. *Wireless Application Protocol: Architecture Specification*, April 1998.
34. WAP Forum. *Wireless Application Protocol: Push Access Protocol Specification*, November 1999.
35. WAP Forum. *Wireless Application Protocol: Push Architectural Overview*, November 1999.
36. WAP Forum. *Wireless Application Protocol: Wireless Application Environment Overview*, Version 1.3 edition, March 2000.
37. WAP Forum. *Wireless Application Protocol: Wireless Datagram Protocol Specification*, February 2000.
38. Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, Inc., 1996.
39. Erik Wilde. *Wilde's WWW: Technical Foundations of the World Wide Web*. Springer Verlag, 1999.