Fulcrum – An Open-Implementation Approach to Internet-Scale Context-Aware Publish / Subscribe

Robert T. Boyer William G. Griswold
University of California, San Diego
La Jolla, CA 92039-0114
{ rboyer, wgg }@cs.ucsd.edu

Abstract

Content-based publish / subscribe (CBPS) systems are a natural substrate for context-aware applications because they provide the right separation of concerns, efficient event distribution, extensibility, and scalability. However, the separation of concerns afforded by CBPS middleware precludes publishers from collaborating with each other to efficiently detect context-aware conditions for publication as events. We overcome this problem with an open implementation approach, which enables subscribers to attach a domain-specific implementation strategy to their context-aware subscriptions. Strategies are supported by first-class active subscriptions that can deploy lower-order dynamic publisher / publisher cross-subscriptions where events enter the CBPS system. By exchanging data on an as-needed basis, event traffic can be dramatically reduced. Our algorithm for detecting the proximity of mobile buddies reduces event traffic from O(|events|) to O(|movement|) worst case, O(lg_2|movement|) expected. Experiments reveal an 81% reduction with current positioning technologies.

1. INTRODUCTION

Context-aware computing takes data from a highly dynamic environment and synthesizes it into information and knowledge for decision-making. Of particular interest are the relationships between changing data. Simple examples include: your location with respect to mine, when my objective is to speak with you in an impromptu hallway meeting; or stock price with respect to earnings, i.e., P/E ratio, when the task is to increase retirement investments. Specific context of interest depends on individual users and their current tasks and objectives. Tarasewich provides a useful context-awareness survey [15].

In a sensor network composed of low-powered components deployed in an ecological preserve, we might tag animals of interest with a battery-powered GPS receiver, CPU, and wireless networking. Scientists wish to monitor, throughout the year, the interactions between various animals. To preserve power, communications need to be minimized. The scientists are initially interested to learn when the proximity of certain of animals is less than 10 meters (e.g., $(X_1-X_2)^2 + (Y_1-Y_2)^2 < 10^2$). As these relationships are observed, the system must be able to support measurement for new hypotheses that are generated by the scientists. Also, multiple researchers may observe the same environment with different goals. For cost and feasibility, the sensor network's resources need to be shared.

Generally, we can anticipate that contextual data will ultimately derive from "billions of users connected to millions of services using trillions of devices" [6]. Myriad sensors reporting potentially interesting raw data at high data rates will be as widely distributed as an equally diverse set of consumers, who might themselves be information providers. To effectively support context-aware computing in this environment requires efficient, extensible, and scalable data distribution and processing.

Content-based publish / subscribe (CBPS) systems hold the potential for such a solution. In publish / subscribe systems, publishers publish *events* and subscribers subscribe to *events*, with the CBPS middleware doing event matching and routing. Hence, publishers and subscribers do not need to know about each other, enabling new publishers, subscribers, event types, and subscriptions to be freely added to the milieu.

Not only does the middleware provide separation, but also efficiency: it filters out new events against subscriptions at the publisher's event broker, exploits overlapping subscriptions, and employs multicast-like routing of events to subscribers. Efficient filtering at the publisher's event broker is achieved by content-based pattern matching against a publisher's event in a series of independent filters, e.g., {(e.x < 10) & (e.y > 30)}. Sequences of events can be similarly pattern matched [2,3]. Context-aware relationships like the proximity relationship entail comparing event attributes to each other. This currently requires subscribers to subscribe to the raw location information and compute the distance themselves, hence the efficiencies of evaluating at the publisher's event broker are lost.

Recent work enables the aggregation of attributes from multiple data streams with more complex processing and filtering performed within the network [4,12]. Common aggregations and transformations can also be shared [4]. It is possible, then, to evaluate the proximity relationship at the first common event broker node (***1st CN***) that connects the publishers with the subscriber.

When publishers are distant from the 1st CN, each intervening event broker node must process and forward all events, which is costly. The insidious part of all this is that the extra costs is shared with all other applications using the network. To get the best possible performance

0-7695-2268-8/05/$20.00 (C) 2005 IEEE

requires evaluating context-aware relationships at the publisher's event broker. To do so efficiently requires knowledge of the modeled relationship as well as how it will be used. Only the subscriber has this domain knowledge, and currently has no way of sharing it with the middleware.

The open implementation software design technique [13,14] was developed for just such situations. In the open implementation approach, a module's interface is designed to allow a client to assist in the selection of the module's implementation strategy. The module's auxiliary interface may allow a client to describe its usage patterns (e.g., high insert rate, few deletes), to specify an implementation (e.g., hash table), or even to provide its own implementation – adhering to well-defined interface specifications. This allows the client to tailor the module's implementation strategy to better suit its needs, while retaining the advantages of closed implementation modules (i.e., the traditional black box).

Following the open implementation paradigm, we introduce Fulcrum to create an efficient substrate for context-aware publish / subscribe (CAPS). For the example of tracking deer proximity, Fulcrum enables a reduction in the event traffic from O(|events|) to O($\lg_2$ |movement|) expected, using domain-specific optimizations that filter events at the publishers' event brokers (called *entry nodes*). Fulcrum provides this facility as a three-part extension to CBPS while preserving its anonymous, asynchronous, and loosely coupled nature:

1. A subscriber can subscribe directly to a *derived relationship*, such as the distance between two entities. A derived relationship has no publisher, *per se.*
2. The derived relationship is realized through *active subscription* support. An active subscription, like a Solar *operator* [4], is a first-class pub/sub Java applet; it can be both a publisher and subscriber. In particular, it:
   a. subscribes to event data comprising the relationship;
   b. publishes satisfaction of the derived relationship.
3. A separate *implementation strategy* can be attached to the relationship subscription to exploit the domain's semantics to improve efficiency. In particular, it can push the subscription to the entry nodes and establish a secondary set of subscriptions and publications that permit the relationship to be computed in a distributed, collaborative fashion, thus reducing event traffic.

Note that the same property can be implemented by different strategies as appropriate to the context of use. Similarly, the same implementation strategy can be reused among properties with similar semantics (e.g., the notion of distance is not merely physical, but could be economic).

The remainder of this paper is organized as follows. Section 2 discusses the state of the art capabilities of CBPS and their applicability to context-aware computing. Section 3 describes our design of Fulcrum. Using buddy proximity as an example, Section 4 details how an implementation strategy is deployed to achieve efficiencies. Section 5 evaluates the system as built. Section 6 discusses tradeoffs with our approach. Section 7 concludes.

2. CONTENT-BASED PUB / SUB

In the current state of the practice, as exemplified by Siena [2,3], CBPS systems are composed of three components. One, a *publisher* provides *events (messages)*, each in the form of a tuple sequence $\{(name_1, type_1, value_1), (name_2, type_2, value_2), \ldots\}$. Two, a *subscriber* requests events of interest by using *subscription filters* of the form $\{(name_1, type_1, operator_1, value_1), (name_2, type_2, operator_2, value_2), \ldots\}$, where each *operator* will be a relational operation like <, =, etc. Three, *event brokers* mediate between publishers and subscribers, providing an application-level overlay network for efficiently matching and routing events, providing independence of publishers and subscribers.

Publishers *advertise*[1] their event *types* with brokers, and subscribers register their interest in events through their content subscription filters. Abstractly, the brokers then check every published event against every filter, passing on those events that satisfy a filter. Concretely, the brokers set up a "switching fabric" between publishers and subscribers by pre-matching filters to the advertised event types (i.e., a publisher's event type promises to contain all the required names of a subscriber's filter).

This switching fabric has two performance benefits for the brokers. One, filters are only applied to events that have a chance of satisfying the filter. Two, it is possible to avoid redundant filtering for overlapping subscriptions, as well as send only one copy of each event between brokers when multiple subscribers share parts of the same pathway from the publisher to an intermediate broker. Subscription filters are pushed upstream from the subscriber toward the information producer to quickly suppress information at broker nodes for which no downstream subscriber is currently interested. This eliminates unnecessary network traffic and excess computation at internal event brokers and at the end client. Consequently, the client application is, in effect, being pushed into the network via the middleware of the CBPS system.

CBPS systems seem ideal for supporting extensible, scalable context-aware systems. The brokered publish/subscribe paradigm makes publishers and subscribers largely independent from each other, while providing economies of scale through sharing in the overlay network. Publishers and subscribers can be readily added, accommodating new application functionalities, and new event brokers can be added to improve the efficiency of the network, thus providing Internet-level scalability [2,3].

[1] Many CBPS systems use subscription forwarding instead of advertisements. In leveraging Siena, we are also currently using its advertisement style.

2.1 Context-Awareness Requirements

At its core, a context-aware system needs to detect and react to *situations*, that is, the moment-to-moment circumstances of the entities it supports, tracks, or models. This includes awareness of the changing *relationships* amongst entities. At UCSD we have a context-aware computing research project called ActiveCampus.[8] It was originally designed using a centralized server and database. In exploring the decentralization of ActiveCampus into a federation of cooperating peer servers, our first challenge was efficiently detecting relationships amongst users. In the federated model, each ActiveCampus user would be logged into a "home" server that is responsible for reporting (publishing) that user's context, thus making CBPS a natural substrate. An example relationship would be when a person moves into the vicinity of a buddy, which should result in notifying them of their proximity. This might motivate either person to seek the other out or contact them by instant messaging [9]. A more general example would be the ability of a user to detect when a critical mass of friends are gathering at a location.

The buddy relationship situation is naturally described with a formula[2] like $(X_R-X_W)^2 + (Y_R-Y_W)^2 < D^2$. However, the core capabilities of CBPS do not allow for a subscription to compare event attribute values coming from two different publishers. As a consequence, the subscriber must issue separate subscriptions for Robert's location and William's location. This results in their location events being pushed all the way through the network to the subscriber, which computes their distance itself, likely learning that the friends are no where near each other. Network cycles and bandwidth and subscriber cycles are wasted.

To directly and efficiently support the detection and reporting of relationships, then, a CBPS system must:

1. Enable intra- and inter-event attribute-to-attribute computations (e.g., to compare X and Y attributes from Robert's position event with X and Y attributes from William's position event).

2. Enable a new event type to be advertised and subsequently published from within the CBPS middleware. Such an event either aggregates attributes from other "sub-events" into a new unified event or abstractly represents the satisfaction of a relationship subscription like $(X_R-X_W)^2 + (Y_R-Y_W)^2 < D^2$ in a form "Robert near William."

3. Maintain state information at event brokers to support data aggregation. In particular, two events contributing to an aggregate event (e.g., William's new position and Robert's new position) may not arrive at the same time, so an event broker will need to store events until all the required events are present.

4. Suppress the propagation of sub-events at the publisher's event broker if it could not satisfy an aggregate event. Suppressing at a common node can be wasteful for many broker network configurations. For example, if Robert and William are far apart, and moving slowly, it would be best if their respective position reports were not pushed through the broker network.

5. Provide mechanisms to atomically query on, and subscribe for, an event. Akin to fetch-and-add, such a feature prevents race conditions when the contents of a subscription depend on dynamic data. For example, in adjusting a "suppression limit" based on an event's current attribute value, the value could change between the time of query and time of subscription, causing the desired event to not be propagated, thus missing a key state transition.

Of course, in satisfying these requirements we still wish to retain CBPS transparency and separation of concerns.

2.2 CBPS State of the Art – Related Work

Gryphon [12] and Solar [4,5] are exemplary of systems beginning to meet these requirements, in particular providing support for derived subscriptions.

Gryphon employs a relational database model to events. The proximity of Robert and William would be computed by specifying a join of their location views at the 1^{st}CN, followed by a select on the one-row table with a predicate like $\{(X_R-X_W)^2 + (Y_R-Y_W)^2 < D^2\}$. The authors acknowledge the need for upstream event suppression, and are working on an idea called "selective curiosity", which makes the aggregation node responsible for pushing event-reduction clues back to the information providers.

Solar employs an acyclic dataflow model, extending CBPS with deployed-code operators to filter, aggregate, or transform event data. With Solar, the proximity would be determined with operators that aggregate buddy location events into a single event at the 1^{st}CN, transform it into distance, and then apply a filter for the distance constraint.

Both provide for context-awareness, while off-loading the subscriber. They still load the network with O(|events|) event traffic and processing at every node between the publishers and the 1^{st}CN, as shown in Figure 1. Solar can save some processing through rate-limiting filters and event transformation.

3. FULCRUM

"*Give me a place to stand and a lever long enough and a fulcrum on which to place it, and I shall move the world.*" – Archimedes, 220 B.C.

These costs are unacceptable in context-aware environments, where there can be thousands of publishers (devices) publishing events at high rates. Ideally, the middleware would permit suppressing location events at their entry nodes, only forwarding those that could satisfy a derived relationship event.

---

[2] Subscripts are used here to denote separate events from possibly different publishers.

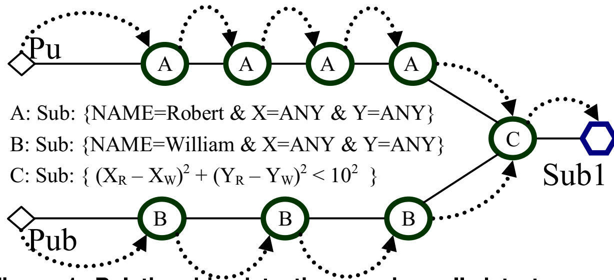

Figure 1. Relationship detection requires all data to pass through all event brokers until a common node is reached. Letters indicate processing occurring at the event brokers. The solid lines represent overlay network connectivity while the dashed arrows represent event movement.

Fulcrum is a context-aware publish / subscribe (CAPS) system that employs open implementation to give subscribers the ability to efficiently and transparently subscribe to relationships. That is, in addition to the ability to subscribe to relationships like Gryphon and Solar, the subscriber can specify an implementation strategy for the subscription. Because the subscriber is specifying the implementation strategy, the strategy can exploit domain-specific properties that the middleware could not know.

An implementation strategy is nothing less than a distributed algorithm for evaluating the subscription. The strategy itself is implemented with publish / subscribe amongst the brokers in the network. A strategy is kept separate from a subscription through a strategy design pattern and a factory for instantiating the subscription-strategy pair [7]. The separation provides several benefits, including the ability to:

1. parametrically substitute different implementation strategies for different environments;
2. permit non-expert programmers to specify an implementation strategy written by an expert programmer;
3. prototype an application's behavior by writing only its subscriptions, followed later by declaratively attaching an appropriate optimizing implementation strategy;
4. reuse an implementation strategy for subscriptions that are similar in structure.

Consider our buddy proximity relationship. The first domain-specific property of note is that the given rate of location reporting is only a sampling of the actual locations. Second, if Robert is reported at location A and next at B, then he must have traveled a continuous line between those points, with time progressing smoothly over the interval. Third, there are a range of expected rates of travel: people walk at about 5 km per hour, travel in cars at 100 km per hour, and fly in planes at 800 km per hour. Fourth, the use of the proximity subscription itself, in our example, is to enable striking up a conversation.

How might these be used to improve efficiency? The distance of two people (200 km), compared against their current rate of movement (5 km per hour) implies that they could not be proximate for 20 hours. Hence, an implementation strategy could suppress location reports at an entry node until that time has passed, either's rate of movement changes, or either moves half the original distance between them.

3.1 Implementation Approach

Fulcrum is built on portions of Siena and Jabber [10]. To support relationship subscriptions and implementation strategies, Fulcrum employs *active subscriptions* – deployed code in the form of Java[tm] class files. They are integrated with Siena's subscription and notification components. To interoperate with the UCSD ActiveCampus project [8], the overlay network is built on Jabber using the XML-based J-XMPP java client [11]. This allows Fulcrum to leverage the presence features of XMPP ("Robert is logged in"), enhanced with location information as detected though ActiveCampus. Support for active subscriptions and the Jabber integration required replacing Siena's event brokers with our own. Fulcrum consists of 81 Java classes, comprising ~7,000 source lines of code.

3.2 Event Broker Architecture

Active subscriptions may take on the role of publisher, subscriber, or both. For example, our proximity subscription subscribes to the location events of two people and publishes a new event regarding proximity. In order for multiple such subscriptions to aggregate and transform events, there is a need for statefulness and multi-threading in the event broker. Our broker architecture is shown in Figure 2. There are two significant pieces as follows:

1. The primary event broker (large circle) and its associated message (event) buffer are like those from traditional CBPS systems. The primary event broker handles the normal advertisement, subscription, and publication processing. It forwards arriving events to the appropriate active subscription or another event broker. It also provides additional hooks to plug-in active subscriptions.

2. A wrapper (smaller box) provides both a sandbox and supporting interfaces for an active subscription (hexagon). It provides a miniature broker environment, including a message buffer and event router (small circle).

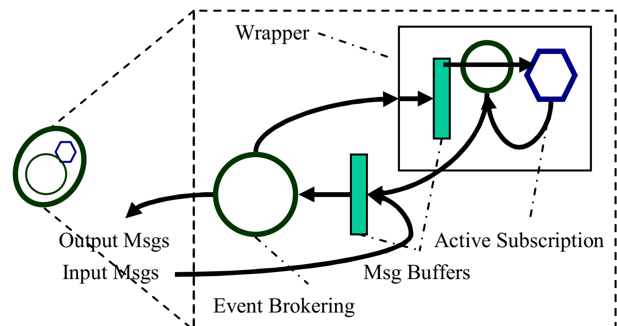

Figure 2. Event broker high-level architecture.

3.3 Subscription Management

As part of our support for suppressing events at entry nodes, we make a distinction between *local* and *global* subscriptions. A global subscription is the normal subscription found in CBPS systems. We added local subscriptions to enable an active subscription to live only immediately adjacent to a data producer of interest and not have its subscription be delivered to remote publishers that might generate similar information. These are used only by implementation strategies, not by normal subscriptions. Both local and global subscriptions receive any data that arrives at a broker node. However, when a new publisher advertisement is received then only global subscriptions are propagated to broker reporting advertisement.

3.4 Subscription API

Following the open implementation approach, the implementation strategy is passed as an optional parameter:

```
subscribe( filter )
subscribe( filter, implStrategy )
```

This both preserves backwards compatibility and allows incremental addition of the strategy.

3.5 Active Subscription Deployment

Two things need to happen in deploying an active subscription. One, similar to Siena subscriptions, the relationship subscription needs to be deployed to every broker in the network between the subscriber and contributing publishers. Two, the implementation strategy needs to be instantiated at the appropriate brokers. Fulcrum manages deployment of an active subscription as follows.

First, the active subscription is packaged to appear as just its relationship subscription, that is, with the implementation strategy hidden inside it. Consequently, the existing subscription routing mechanisms are used. In particular, based on the registered advertisements at each event broker, the subscription follows the path for every advertisement of interest upstream until the edges of the network are reached, with the subscription being left at each intermediary node for forwarding purposes.

Second, the subscription's implementation strategy is evaluated (instantiated) at each node. For non-entry nodes, the strategy is simply discarded, and the subscription merely functions to forward events back to the subscriber. For each entry node, instantiating the strategy results in the following actions:

1. The strategy determines which events are to be subscribed to with local subscriptions. Our proximity subscription $\{(X_R - X_W)^2 + (Y_R - Y_W)^2 < 10^2\}$ implies two constituent subscriptions:

   A. {NAME=Robert & X=ANY & Y=ANY}

   B. {NAME=William & X=ANY & Y=ANY}

At entry nodes advertising Robert's location, the strategy makes a *local* subscription for A. At entry nodes advertising William's location, the strategy makes a local subscription for B. This is shown in Figure 3a.

2. The strategy advertises new event types and makes additional *global* subscriptions that exchange data with the same strategy on the other entry nodes. The data from this sub-protocol is used in two ways: one, to *evaluate* the relationship subscription to possibly forward a proximity relationship satisfied event to the originating subscriber, and two, to collaboratively *schedule* the evaluation of the relationship subscription. That is, the strategy normally evaluates event data locally and only forwards significant events between the entry nodes as shown in Figure 3b.[3] Efficiency is achieved by suppressing useless events. Satisfaction of the relationship results in a single event being passed to the original subscriber as shown in Figure 3c.

Note that the active subscription running on each entry node does not need to know where the other strategies lie or even if the other active subscription is in place. They merely advertise their new event types, and the collaborator subscribes to them. The middleware marries them in the normal publish / subscribe fashion.

Using a buddy proximity explanation, the next section details how Fulcrum can be used to achieve efficient yet transparent context-aware publication / subscribe. Section 5 describes an experiment using Fulcrum to track buddy proximity for the ActiveCampus system, revealing the efficiencies that are possible with this approach. It is impor-

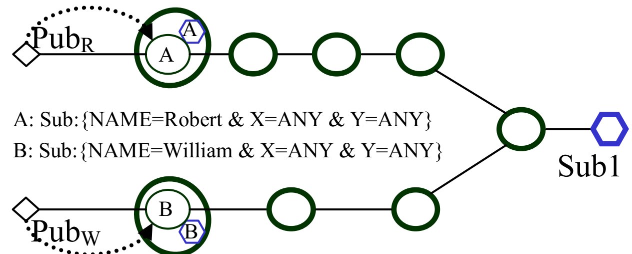

Figure 3a. Active-subscriptions are deployed to entry nodes, such that only entry nodes receive original events.

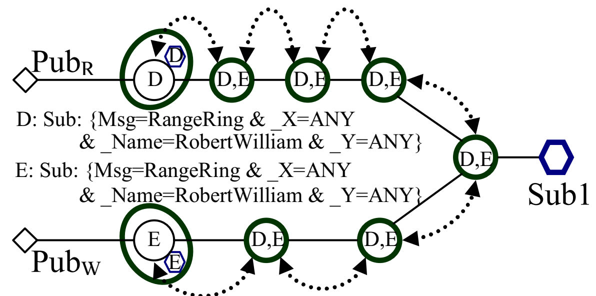

Figure 3b. A logical cross-subscription is established to collaborate between active subscriptions, sharing only significant event information, suppressing all other events.

[3] An underscore is prepended to attribute names that overlap with those of location events, preventing subscribers to location events from also getting the range ring events. A general solution would employ a scoping mechanism like namespaces.

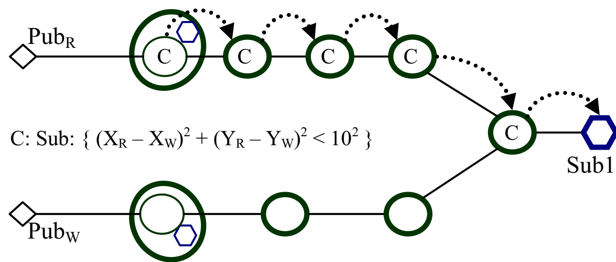

Figure 3c. Only a single event representing the satisfaction of the relationship traverses the network to the subscriber.

tant to recognize that this pattern is useful anytime a distributed state machine can be described that allows a substantial reduction in event traffic at the entry nodes.

4. EXAMPLE – BUDDY PROXIMITY

Evaluating subscriptions at the entry nodes is a necessary but not sufficient condition for the most efficient evaluation of relationship subscriptions. In particular, an efficient distributed algorithm for relationship detection must be implemented at the entry nodes. This section describes such an algorithm and its implementation for the proximity of two buddies. [4]

The essential idea is that the proximity of two distant entities cannot be achieved unless the territory between them is crossed: Robert cannot be co-located with William unless one or both of them first moves so that the distance separating them is cut in half, and so on. In our *Range-Ring Strategy*, these "halfway" points are the only events communicated between collaborating active subscriptions at the entry nodes. All other event traffic can be suppressed. When the initial active subscription is deployed, the distance between the two buddies is computed by the strategies running at William's and Robert's entry nodes. Each strategy then computes a "range ring" around its user's position, where the separation between rings is $distance_{desired}$ and the radius of each ring is ($distance_{current}$ – $distance_{desired}$) / 2 as in Figure 4.

The essential invariant, then, is that each user is to be kept inside his range ring. When William, say, leaves his range ring, a new ring is computed and Robert's strategy is notified in the form of a new range ring and new location for William (solid blue circle). At such steps, the proximity subscription itself is also tested. Otherwise, all location events are suppressed. When a user leaves his ring by approaching its buddy, the range rings shrink; when the user leaves his ring while receding from his buddy, the range rings grow.

To preserve the containment invariant, a few things are necessary. First, the user's (William's) new range ring and location are published as an aggregate event. Thus, the re-

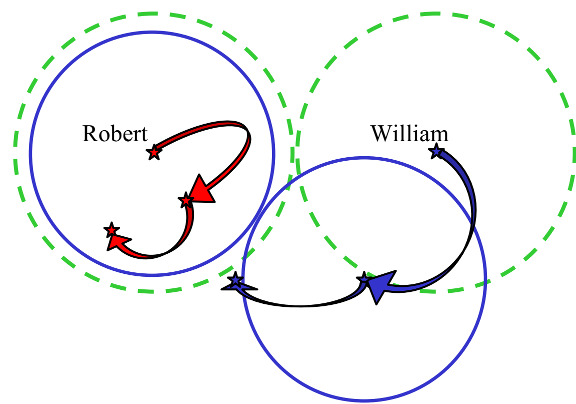

Figure 4. Range-rings act as event suppression filters.

ceiver (Robert's strategy) knows exactly what data was used in computing the new range ring. Second, note that William's strategy must compute the size of its new range ring by using the last reported location of Robert. Consequently, when this range ring is received by Robert's strategy, the range ring must be placed at its own last reported position (since that's the location that William's strategy knows). This in effect is an update to Robert's strategy's suppression filter. Thus, third, it is necessary for Robert's strategy to atomically check whether Robert's current location is outside the new range ring. If so, the strategy follows up with its own new range ring event to ensure preservation of the invariant. The race condition of the users simultaneously leaving their range rings is handled in part by this atomic check, and in part by the strategies keeping the smaller of the two rings that are exchanged.

In the case where one user moves directly towards the other, who is stationary, each successive range ring would be half the size, representing the fact that less additional movement will bring the two into proximity. For this scenario, with initial separation S and proximity bound of D, there are ($2 + \log_2($ S-D $)$) range ring events sent before the proximity event itself is sent. In a scenario where both users are moving at velocity V and remain at a constant separation S, the range-ring event *rate* is 2V/(S-D). The publication rate of location events does not matter in either scenario because only the significant events—those that represent leaving a range ring—are sent. Thus, the algorithm has a desirable "pay for what you use" property, and its suppression of location events has greater value as the reporting rate of the deployed positioning technology increases.

4.1 Strategy Discussion

In summary, active subscriptions replace numerous user location events with a few self-regulating range-ring events, thus substantially reducing the event traffic for computing buddy proximity. If one of the actors is known

[4] The full algorithm, its generalization to *n* buddies, and proof of correctness are beyond the scope of this paper.

to move faster than the other, the range ring's sizes could be skewed in size to account for the additional context.

The use of range rings to indicate when to next publish positional information is only one possible implementation strategy. We could have identified the perpendicular line between the two positions and then created two parallel lines 5 meters on either side of the center, using them as significant event triggers.

If a subscriber is interested in proximity, but in a different conceptual domain, such as temporal or financial, then perhaps an existing strategy could be used. Consequently, a library of strategies could be a useful resource in an open publish / subscribe system.

4.2 Client Mobility

In our example, the entities move. Efficiency considerations suggest that a client (publisher or subscriber) should be close to its associated middleware node. There are two straightforward mechanisms by which to address client mobility.

4.2.1 Default CBPS Semantics. CBPS focusses on content rather than identity. The rehosting of either client is of no consequence to the other – just so long as the appropriate events are propagated through the network. A mobile publisher can abandon its access point, move to a new location, and re-register itself. Upon disconnection, the middleware will remove the active subscription and later a new one will be introduced when the publisher reconnects to the network. The active subscriptions then resynch as if starting from scratch. A similar scenario exists for moving the subscriber. In both cases, the publisher and subscriber must have the contextual awareness to know they are exiting and reentering the network. Also, the active subscription must be designed robustly to support such activities.

4.2.2 Mobility Service. If a mobility service [1] were available, then the pub/sub client need not be aware of any disconnected state. Such services automatically handle the separation with moveOut/moveIn APIs. They use a proxy at the old access point, buffering any messages, until such time as the client reconnects. Active subscriptions do not present any additional mobility problems.

In either case, race conditions can arise. Most notably, if the publisher's state satisfies a subscription while disconnected, the event may not be pushed into the network. If events are buffered via a mobility service, then an event may get delivered more than once, requiring disambiguation at the subscriber.

5. EXPERIMENTAL EVALUATION

The efficiencies achieved by the Fulcrum approach are dependent on properties of the supported applications, the way people use them, the state of the underlying technology, and the structure of the event broker network.

This section describes an experiment of Fulcrum supporting a federated architecture for ActiveCampus and compares those results to the basic and enhanced CBPS approaches. We perform this evaluation in two parts. First, how much suppression of original events does Fulcrum achieve? Second, what overall efficiencies are achieved?

5.1 Experiment

ActiveCampus user positions are automatically generated via triangulation based on 802.11 signal strength and automatically reported to the system. On ActiveCampus's "buddy page", the user display shows all buddies with an indicator of *nearby* or *far*, mirroring the proximity relationship that we have used in our examples. Consequently, we took a week's worth of data from ActiveCampus and "replayed" it through Fulcrum, using a conservative broker configuration.

5.1.1 Setup. The ActiveCampus test data consists of:

643	users (anonymized for privacy).
2,165	buddy relationships
604,800	seconds = 168 hours = 1 week
1-6	events per logged-in user per minute
360,067	location reports

The overlay network was configured with 5 event broker nodes in a crossbar configuration with one internal "routing" node (Figure 5). Such a configuration might be used for geographic coverage (e.g., one node per city plus the crossbar) or to mediate different commercial providers (e.g., AIM, MSN, ICQ, and Yahoo connected through a crossbar). Different configurations would change the number of hops that an event has to travel. This configuration has a maximum hop count of 2, minimizing the penalty for an event not being suppressed.

The 643 users were randomly distributed across the 4 edge brokers, resulting in 149, 158, 165, and 171 users at each respective node. Each user was configured as a publisher of its location as well as a subscriber to the proximity of each buddy.

This configuration resulted in the distribution of proximity subscriptions as shown in Table 1. Users attached to node 0 have proximity relationships with the number buddies on nodes 0 to 3 as 125, 155, 165, and 112 respectively. This resulted in 2,165 active subscriptions being

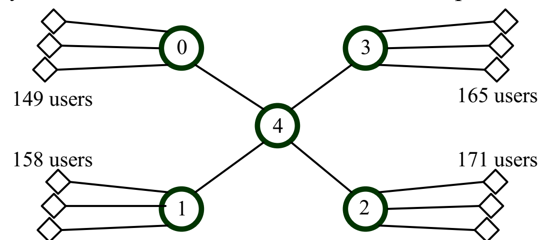

Figure 5. Crossbar event broker overlay network.

deployed. Each was instantiated on two nodes, yielding 4330 instances.

Table 1. Proximity relationship distribution.

Nodes	0	1	2	3
0	125	155	165	112
1	131	140	167	115
2	125	171	224	126
3	85	110	122	92

5.1.2 Results. Of the original 360,067 location reports, 11,955 were associated with users not participating in a buddy relationship, leaving 348,112 location reports that were subject to the proximity subscription.

Many users were not logged in to ActiveCampus at the same time as their buddies during the one-week period. Of the 2,165 active subscriptions, then, only 1,028 received data from both buddies. This means, effectively, that the 348,112 location reports were concentrated on half of the users. The range ring strategy resulted in 57,348 published range-ring events, 4,626 proximity relationship events (i.e., reports of a buddy moving into proximity), and an additional 4330 advertisements for the active subscriptions – 16.5%, 1.3%, and 1.2% of the original event count respectively, for a cumulative reduction of 81%. The breakdown of event reduction for each user's buddy is shown in Figure 6.

Comparing the number of original events to the number resulting from the active subscriptions yields an average event savings of 81% (a factor of 5.25). The average savings factor does not tell the whole story, however. Due to the centralized architecture of ActiveCampus at the time of data capture, user events were rate limited to 1 to 6 events per minute. In the wild, it is not unreasonable to expect 1-second (standard GPS reporting rate on naval ships) or better update rates – a 10-times increase. Because the number of range-ring events is based on distance, not the number of reports, the average event savings would then be 98%, a factor of 52.5. Also, user activity plays a significant role in event reduction as shown by a few sample data points displayed in Table 2.

Table 2. User activity affects event reduction.

User Activity	# orig events	# range ring events
Stationary	22,657	6
Stationary	11,743	2
Mobile	1,008	127
Mobile	4,746	145

In many cases we find that users are relatively stationary. Their usage model is to go somewhere, turn their device on, remain in a confined space for a time, and finally to turn off their device and repeat the cycle. Thus, almost all events are suppressed at the source. Conversely, when people are in motion, there is usually much less suppression because buddies frequently exit their assigned range rings yet seldom walk directly toward each other.

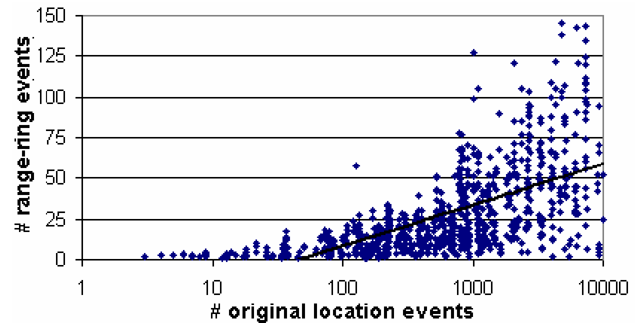

Figure 6. Range-ring events generated per original event. A line is overlaid to show an O(lg_2 movement) expectation.

5.2 Analysis

Although an 81% reduction in events was achieved, it still remains to be seen how much event processing was reduced system-wide.

Due to difficulties in running live Siena, Gryphon, and Solar CBPSs on the ActiveCampus data, we measured system efficiencies in terms of analytically accessible properties. The core costs are the processing of an event at an event broker and any subsequent forwarding of the event through the broker network. We use the number of *hops per original event*, or *event hops*, as our common unit of measure to evaluate the efficacy of Fulcrum. We discuss the cost per hop in the next section.

To simplify the calculation of effectiveness we assume a uniform distribution of publishers, subscribers, and active subscriptions across all edge nodes. To verify that this simplification would not bias our results, we compared Fulcrum's empirical aggregate hop count based on the random distribution to the analytically derived one. The empirical aggregate count based on the configuration in the previous subsection is 94,334 hops. Analytically, it computes as 94,260, less than 0.1% smaller. We have confidence, then, that the analysis here is accurate.

Basic CBPS like Siena requires the subscribers to evaluate all relationships. Consequently, ¼ of the events coming into entry nodes and are immediately returned to subscribers attached to the same node, resulting in 1 event hop apiece (we only count processing and output, not the input). ¾ of the events will come into an entry node and get passed to the center node, on to one of the remaining three edge nodes, and finally to the subscriber, yielding 3 event hops. The subscriber itself evaluates the relationship requiring 0 event hops. On average this is:

$\frac{1}{4} * 1 + \frac{3}{4} * (3 + 0.0133 * 0) = 2.5$ event hops.

Aggregation enhanced CBPS like Gryphon or Solar allows relationships to be computed at the first common node (1^{st}CN). For the given broker configuration, the numbers work out on average to:

$\frac{1}{4} * (0.0133 * 1) + \frac{3}{4} * (1 + 0.0133 * 2) = 0.773$ hops.

Open-implementation CAPS like Fulcrum's reduces original events into potentially significant events at the entry nodes. In this case, ¼ of the events come into an entry node that possesses both parts of an active subscription, yielding 1-hop success events. ¾ of the events come into an entry node hosting an active subscription that collaborates with a distant entry node. Empirical data from the previous sub-section shows that 6.07 original events will become one collaboration event (range ring). These events travel through the center node and out to an associated active subscription. This yields 0.165 event hops for such events. Success events are then passed to the subscriber, which half the time will be local (1 hop) and the other half will be remote (3 hops) for an average of 2 hops. The net event hops per original event are:

$\frac{1}{4}(0.0133*1) + \frac{3}{4}(0.165 * 2 + 0.0133 * 2) = 0.27$

Active-subscription advertisements add (0.0124 * 4) = 0.05 hops. Largely non-recurring, they amortize over time.

If sensor reporting increases to 1 report per second, a 10-times increase, then basic and enhanced CBPS will have the same hop count and see a 10-times increase in traffic. In Fulcrum, because forwarded events are driven by user behavior, not event rate, the overlay network sees no increase in traffic, which yields a corresponding 10-times effective suppression, resulting in 0.027 event hops per original event. Solar could achieve similar reductions by adding a rate-limiting transformation operator, but would sacrifice the added accuracy of the higher rate unless the rate-limiter made inferences over all the events.

6. DISCUSSION

Filtering is the lifeblood of CBPS. Our ability to reduce event traffic by filtering relational events at entry nodes is our key efficiency measure. We now discuss some of the tradeoffs that arise in achieving this reduction.

The core assumption of CBPS is that economies of scale can be achieved by leveraging overlapping interest in the same data. Having tailored active subscriptions as we have done violates that assumption. However, context is a very individualized concept as exemplified by buddy lists. We believe we gain the best of both worlds by leveraging CBPS as a core infrastructure for the commonality while providing efficiency mechanisms for the individuality of context-awareness.

One concern is the large number of subscriptions that might be deployed as Java code. For our experiment, there were 2,165 relationship subscriptions for 643 users with 3.37 average buddies. It should be noted, first of all, that there is intrinsic sharing. The Java class loader only loads unique classes, so only the first such subscription at a node results in transmitting code and configuring the class. Second, open implementation allows the subscriber to achieve higher-levels of sharing by plugging in a new strategy. For example, a user could attach a strategy that tracks multiple buddies for a user, resulting in a sharing of range rings and event subscriptions behind the scenes.

Another question is whether subscribers could, with an existing CBPS like Solar or Gryphon, write collaborative subscriptions like those realized through our open implementation approach. Indeed, for our example, a subscriber could put out subscriptions for range ring events on each publisher. The subscriber itself would have to act as the clearinghouse that receives the events. Upon doing so, it would create new range ring subscriptions and retract the old ones. There are three problems. For one, this added level of indirection would increase the hop count unless the subscriber was attached to the 1^{st}CN. Two, there is not a clear separation between the relational property of interest and its efficient implementation, increasing complexity and reducing the opportunities for reuse. Three, there is a race condition, at least with this straightforward adaptation. Our algorithm avoids the race condition by *updating* the existing subscription on the other entry node, and before doing so checks that the other user has not already moved outside the new range ring. This kind of atomic check is not possible in normal CBPS when placing a new subscription; a subscription filter simply waits for the arrival of the next event.

Although hop counts are reduced by our approach, event processing is slower than in a basic CBPS like Siena. In providing the ability to perform attribute-to-attribute comparisons, efficient structuring of subscription evaluations by event brokers (e.g., exploiting overlaps, like X > 10 is subsumed by X > 5) is hampered. Solar and Gryphon share this problem. To mitigate this, we take a cue from database query optimization and reorganize subscriptions to evaluate the "normal" part of a subscription first, and only perform the attribute-attribute comparisons if it succeeds. More aggressive optimization is a subject of future work. It should be noted however, that reducing the hop count through open implementation is scalable with respect to increasing sensor reporting rate, whereas as basic CBPS is not.

The Java code of an active subscription could be inefficient, buggy, or malicious. The resources available to "greedy" subscriptions can be reduced by sandboxing the computation in a separate thread and using mechanisms akin to Jabber's *karma*.

Active networks share similar considerations in employing deployed code to allow the "switches of the network to perform customized computations on the messages flowing through them." [16] The principal use envisioned for active networks is the enabling of rapid innovation in network security (e.g., firewalls), performance (e.g., caching, compression), and compatibility (e.g., transcoding) [16]. In principle, CAPS and CBPS could be implemented using active networks, removing the need for a separate middleware layer and improving performance.

7. CONCLUSION

The commoditization of networked sensors is fueling internet-based context-aware applications. Content-based publish / subscribe (CBPS) systems are a natural substrate for supporting context-aware application development because they provide for separation of publishers and subscribers, efficient event distribution, extensibility, and scalability. Yet, the CBPS notion of subscribing to a publisher's content does not capture the data relationships across publishers that drive context-awareness.

Recent advances in aggregation-enhanced CBPS permit subscribing to these relationships, but the separation of concerns afforded by the middleware precludes publishers' event brokers from collaborating with subscribers and each other to implement algorithmically efficient, application-specific context-aware inferences.

The number of events processed and forwarded through the middleware can be dramatically reduced by using open implementation to permit subscribers to enhance the middleware by deploying a distributed algorithm to where events enter the system. Open implementation provides for separation of subscriptions and implementation strategies, allowing for separate, modular development and reuse of relationship subscriptions and implementation strategies.

Fulcrum supports open-implementation context-aware publish / subscribe (CAPS). It employs *active subscriptions* in the form of Java applets. They are first-class entities that both subscribe to events and publish new events, enabling collaboration amongst instances of the strategy deployed where raw sensor events enter the system.

We evaluated the Fulcrum approach by implementing a buddy-proximity relationship and accompanying strategy. The resulting strategy's performance is tied to the amount of user movement, not to the number of events reported. We then federated the ActiveCampus context-aware infrastructure and replayed a week's worth of its user data. For a conservative broker network configuration and current sensor technology, we found an 81% net event reduction and a 64% reduction in event hops compared to aggregation-enhanced CBPS, which uses similar underlying technology. These factors increase proportionally with increased reporting rates, naturally insulating against advances in sensor technology. The data curve-fits to $O(\lg_2|\text{movement}|)$ expected event processing.

8. REFERENCES

[1] Caporuscio, M., Carzaniga, A., Wolf, A., Design and Evaluation of a Support Service for Mobile, Wireless Publish / Subscribe Applications, University Colorado Technical Report CU-CS-944-03, January, 2003.

[2] Carzaniga, A., Architectures for an Event Notification Service Scalable to Wide-area Networks, PhD. Thesis, Computer Science, University of Colorado, Boulder, 1998.

[3] Carzaniga. A., Rosenblum, D. S., and Wolf, A. L., Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service, *19th ACM Symposium on Principles of Distributed Computing*, pp 219-227, 2000.

[4] Chen, G. and Kotz, D., Solar: Toward a Flexible and Scalable Data-Fusion Infrastructure for Ubiquitous Computing, *UbiComp 2001 UbiTools Workshop,* 2001.

[5] Chen, G. and Kotz, D., Solar: A Pervasive-Computing Infrastructure for Context-Aware Mobile Applications, Dartmouth Technical Report, TR2002-421, 2002.

[6] Cohen, N., Purakayastha, A., Turek, J., Wong, L., Yeh, D., Challenges in Flexible Aggregation of Pervasive Data, IBM T.J. Watson Research Center, Tech Report RC21942, 2000.

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

[8] Griswold, W., Boyer, R., Brown, S., and Truong, T., A Component Architecture for an Extensible, Highly Integrated Context-Aware Computing Infrastructure, *2003 International Conference on Software Engineering*, May 2003.

[9] Griswold, W., Shanahan, P., Brown, S., Boyer, R., Ratto, M., Shapiro, R., and Truong, T., ActiveCampus – Experiments in Community-Oriented Ubiquitous Computing, Technical Report CS2003-0750, UC San Diego, June 2003.

[10] Jabber Software Foundation. 2004. Jabber: Open Instant Messaging and a Whole Lot More, Powered by XMPP. [Online] Available: http://www.jabber.org/ [2/22/04]

[11] JabberStudio. 2003. J-XMPP Project Information, [Online]: http://www.jabberstudio.org/projects/j-xmpp/project/view.php [2/22/04]

[12] Jin, Y. and Strom, R., Relational Subscription Middleware for Internet-Scale Publish-Subscribe, *Proceedings Second International Workshop on Distributed Event-Based Systems*, 2003.

[13] Kiczales, G., Beyond the Black Box: Open Implementation, *IEEE Software*, January 1996.

[14] Kiczales, G., Lamping, J., Lopes, C.V., Maeda, C., Mendhekar, A., Murphy, G., Open Implementation Design Guidelines, *1997 International Conference on Software Engineering (ICSE)*, May 1997.

[15] Tarasewich, P., Toward a Comprehensive Model of Context for Mobile and Wireless Computing, *Proceedings of America's Conference on Information Systems (AMCIS) 2003*, 2003.

[16] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A Survey of Active Network Research. IEEE Communications, pp 80-86, January 1997.