UbiBroker: Event-based Communication Architecture for Pervasive Display Networks

Tommi Heikkinen, Petri Luojus, Timo Ojala
MediaTeam Oulu, Department of Computer Science and Engineering, University of Oulu
Oulu, Finland
{firstname.lastname}@ee.oulu.fi

***Abstract*—This paper presents an event-based communication middleware developed for a fairly large pervasive display network installed in a city center. We demonstrate the feasibility of the middleware with a set of dynamic and distributed prototype applications implemented for the display network. We also conduct an empirical performance evaluation of the middleware in lab and real world settings.**

Keywords—communication middleware; messaging system; publish-subscribe; public displays

I. Introduction

In this paper we report the current status of our work-in-progress of developing a new communication middleware for a fairly large-scale real world network of multipurpose pervasive displays, so-called UBI-hotspots (Fig. 1), deployed at pivotal outdoor and indoor locations around Oulu, Finland [1]. The first complete release of our display middleware [2] used Fuego [3], an open source publish-subscribe messaging middleware supporting content-based routing of events. However, due to limited client support and stability issues Fuego failed to attract sustained trust and use among application developers and eventually became obsolete. This is typical to many experimental software projects that eventually fail to live up to the standards required by a long lasting real world deployment.

Upon giving up on Fuego, we have re-designed the communication middleware with today's software standards, yielding the proposed UbiBroker event-based communication architecture. The key requirements for UbiBroker included support for distributed and ad hoc application structures, stability, interoperability with a wide range of client platforms, and simplicity to reduce the burden of application developers. These requirements stem from our long-term experience in developing and maintaining a middleware software layer for display applications that are developed by many third party application developers, both academic and industrial, and that have to provide the high availability required by the 24/7 deployment in a city center.

This paper is structured so that we first briefly recap previously proposed communication middleware for pervasive computing systems in Section II. Section III presents the design and requirements underlying the proposed UbiBroker architecture. Section IV describes the implementation of the UbiBroker architecture. Section V briefly introduces several prototypes implemented atop UbiBroker and subsequently deployed on our display network. Section VI reports the empirical performance evaluations of the broker in a lab and in our display network, and briefly summarizes the experiences of the six month long real world deployment so far. Section VIII concludes the paper with a discussion.

Fig. 1. A double-sided outdoor UBI-hotspot at the Oulu market place.

II. Related Work

Various communication middleware have been presented for pervasive computing systems in the literature. Gaia [4] was one of the early attempts to design a complete top down middleware for ubicomp applications. The communication model in Gaia is based on CORBA's event service which provides interoperability between clients implemented in different programming languages. The suppliers and consumers are decoupled by using specific channels similar to the publish-subscribe pattern. Equip [5] application framework is also based on CORBA. Equip is a distributed interactive middleware system to bridge physical world and mixed reality. The communication system dubbed Data Service is based on the publish-subscribe messaging model implemented with a tuple space and loosely coupled template matching.

The Interactive Workspace project [6] developed a communication middleware in the context of an interactive room. The communication is based on a centralized tuple space dubbed EventHeap which in turn is based on IBM TSpaces. The middleware supports multiple programming environments including Java, C++ and web. Routing of events is based on template matching. As the project was limited to an interactive room, the scalability of the proposed communication middleware for larger deployments remained unclear.

The One.world project [7] presented software architecture for pervasive computing systems that embraced contextual change, encouraged ad hoc composition and recognized sharing as default. The communication middleware includes a Java based tuple space and asynchronous event service. Data is represented as tuples in generic self-describing named and optionally typed fields and all communication is implemented using asynchronous events. One.world supports only Java clients.

GREEN [8] is a highly configurable and re-configurable publish-subscribe middleware for pervasive computing applications. GREEN middleware supports pluggable interaction types that are topic-based, content-based and context (proximity) and also pluggable event brokers to embrace different network types and devices. Similar approach was adopted by MundoCore [9] which was designed for an environment with high degree of heterogeneous networks and platforms. The kernel implementation of MundoCore is available in C++, Java and Python and can be installed onto most common operating systems including mobile. The low level communication pattern is based on topic-based publish-subscribe, but similarly to GREEN, MundoCore also offers other interaction patterns.

MAGIC Broker [10] is a communication middleware specifically designed for interactive public displays including mobile device interactions. The MAGIC Broker middleware uses a set of common abstractions: channels, events, state, services, and content. The channels describe entities existing in the environment and support hierarchical structure. The events and state information are routed based on channels, i.e., the channel is similar to topic in a publish-subscribe pattern. The publish-subscribe protocol in MAGIC Broker is implemented using the REST architectural style, i.e., the services communicate with MAGIC Broker using the HTTP protocol.

The above middleware share the common objective of supporting ad hoc and distributed application structures by decoupling communicating processes with event based messaging enabled by a message broker. Further, they tend to combine the communication functionality with the component model into a coherent application development framework. Finally, they are based on an outdated and/or a proprietary protocol, which has resulted in the lack of widespread adoption.

While UbiBroker also uses events, it is different from the above middleware in the sense that it treats communication as an independent functionality. In other words, UbiBroker does not specify how applications should be constructed to be supported by it. This separation of concerns places fewer constraints on the selection of programming languages and platforms, and reduces the complexity of application development by making the use of a particular framework optional. UbiBroker is based on a proven and widely used general purpose open source publish-subscribe messaging software that has active developer support. This way, UbiBroker inherits its stability and scalability, which are required by a large-scale long-term real world deployment.

III. Design

A. *Background*

The motivation for this work stems from our many years of experience of operating and managing the Open UBI Oulu urban computing testbed in Oulu, Finland [12]. The objective of the testbed is to facilitate longitudinal large-scale studies of future ubiquitous computing systems in authentic urban setting with real users. The testbed comprises of diverse ‘fixed’ computing resources such as large WiFi and Bluetooth networks and different types of public displays. These heterogeneous computing resources constitute a large distributed system which is organized with a middleware layer. It provides various resources for supporting technology experiments, open APIs for application developers, and tools for managing the testbed and the end-user applications deployed atop the testbed. So far the testbed has been exploited by over 30 academic research groups in nine different countries and a number of businesses.

The UBI-hotspots are the most visible part of the testbed. We deployed first hotspots in May 2009 and have now in total 18 hotspots in 12 indoor and six outdoor locations. A hotspot is effectively a large LCD panel equipped with other computing resources such as a capacitive touch-screen foil, a control PC with a large hard disk, two overhead cameras, a NFC/RFID reader and a Bluetooth access point. The software architecture is designed to allow each hotspot to function individually based on its proximity context. At the same time the hotspots are also networked in a loosely-coupled fashion via an event-based communication overlay, which allows the hotspots to publish and subscribe to events related to their context. This design allows application distribution on multiple levels, from reliance on one hotspot to the utilization of multiple hotspots simultaneously and to coupling with user devices.

The user interface of the hotspots is implemented using the Web paradigm. It comprises a set of webpages rendered by corresponding webserver processes and managed by our in-house screen real estate management system [13, 14]. A hotspot is in either passive broadcast or interactive mode. In passive broadcast mode, the whole screen is dedicated to the UBI-channel, a customizable playlist of video, animation, and still photographs. When a face is detected from the video feeds of the overhead cameras or someone touches the touch-screen foil, the hotspot changes to interactive mode, where the screen is split between the UBI-channel and a customizable UBI-portal. The UBI-portal contains a varying number of services, typically 25-30, in distinct service categories. Over half of the services typically depend upon third party content that is beyond our administrative control. This kind of distributed service provisioning is a must for a cost-efficient and sustainable realization of our multipurpose displays, and the web paradigm has proven very efficient in implementing it. We can quickly include new services residing on any webserver in the Internet, as long as they conform to certain minimal design guidelines.

B. Requirements

1) Decoupling of communicating processes

A pervasive display network and related computing nodes such as user devices constitute a highly dynamic computing environment that has to support ad hoc composition of distributed applications triggered by user input and context events. This calls for referential decoupling of communicating processes to avoid hard-coupled interfaces which make it difficult to replace and move software components. If processes are executing simultaneously, a temporally coupled meeting-oriented communication model can be used. If also temporal decoupling is desired, then a generative communication model can be considered [14]. Therefore, similar to [4, 5, 6, 7, 8, 9, 10], we base our communication model on events. To facilitate one-to-many event-based communication based on patterns, we choose a topic based publish-subscribe as the preferred messaging paradigm similar to [4, 8, 9, 10]. However, in some cases processes may also need to be referentially coupled with point-to-point or RPC messaging paradigms. Therefore, the communication middleware should also support multiple messaging patterns [8, 9].

2) Stability

The middleware has to provide high availability, i.e., the down times must be rare and the system needs to quickly recover from crashes. It is challenging to achieve stability if communication middleware is built from scratch, as reliability tends to come over time after careful testing and sustained development support. Therefore, we prefer proven existing communication middleware solutions, which are preferably open source and supported by active developer community.

3) Interoperability

Developers require interoperability with various operating systems, communication protocols and programming languages. The middleware need to be installable on multiple platforms in order to be compliant with various computing environments and deployments. The communication middleware has to enable communication between different clients, which can be achieved by using standardized protocols and preferably also support integration into most common communication protocols used in a particular application domain. Often the protocol diversity is solved by using adapters. However, it is important that readymade implementations of the adapters exist as implementing a new adapter from scratch can be a tedious task. Developers also have different programming backgrounds and applications may require different technologies. Therefore, it is important to have a wide range of supported programming languages for implementing clients. Web provides best interoperability over different operating systems and is the only common application deployment platform. Therefore, web compliance is an essential feature for the communication middleware.

4) Simplicity

While the developers of middleware components can be expected to tolerate complexity to some extent, the developers of third party applications atop the middleware want simplicity. In our experience they have varying backgrounds and levels of expertise. Simplicity can be achieved by a well-defined lightweight application programming interface, readymade templates, code examples and detailed documentation. The use of the communication middleware should also be optional, i.e., the application developers should not be forced to use it if they do not need it. This is consistent with the web philosophy, where developers have great freedom over technologies they use.

C. Conceptual communication architecture

The conceptual communication architecture of our display network is shown in Fig. 2. The main component is a centralized message broker that takes care of passing messages between processes that we refer to as clients with respect to the broker. The clients loosely belong to three categories. Context sources publish context events originating from an external input device such as a camera or an NFC/RFID reader or from user actions. Applications interact with the users and they can consist of multiple components executed by different processes. Middleware services provide various functionalities and resources to applications.

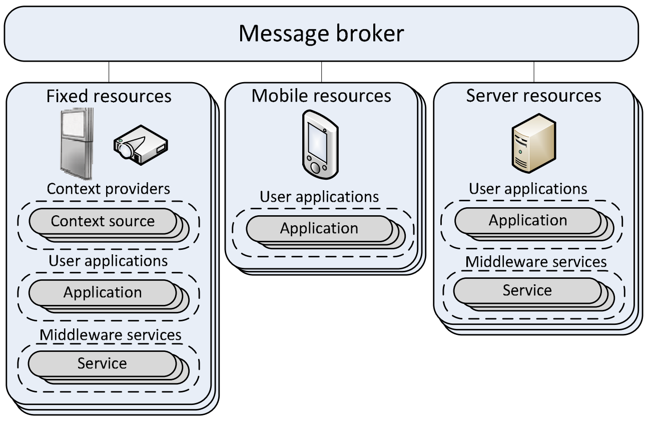

Fig. 2. The conceptual UbiBroker communication architecture.

The clients are executed on different computing platforms, which we divide into three categories. Situated public displays are fixed and typically have high bandwidth network connection and capable hardware. These fixed resources serve people with applications and are managed by the middleware. Depending on their capabilities, they may also be able to capture and produce context events. The second resource category is mobile devices that rely on wireless connectivity and have heterogeneous software platforms. Due to their intermittent connectivity and ad hoc usage patterns, they often establish distributed ad hoc interfaces with other resource types. The third category is server resources that host application processes and general purpose middleware services.

IV. IMPLEMENTATION

We describe the implementation of the UbiBroker architecture using the distributed systems taxonomy of Tanenbaum and Steen [14]. After comparing a number of different message brokers, we chose RabbitMQ [15] for the UbiBroker architecture. RabbitMQ is an open source project with an active developer community and available for a wide range of operating systems and programming languages.

A. Architecture and Processes

Fig. 3 illustrates the client-server architecture of RabbitMQ. There are two types of processes, the RabbitMQ server process and client processes. A client process can act both as a publisher (producer) and a subscriber (consumer) of events simultaneously. RabbitMQ divides resources into multiple administration domains called virtual hosts. The server implementation is available for Windows, Linux/Unix, Mac OS X and EC2 platforms. Official client implementations are available in Java, .NET/C# and Erlang that are supported by a wide range of platforms. Community contributed client implementations are available for many other programming languages, e.g. Ruby, Python, PHP, Perl, and C/C++.

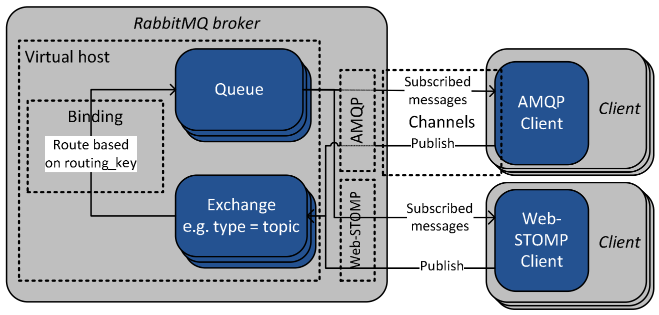

Fig. 3. The C/S architecture of RabbitMQ.

B. Communication

The clients communicate with the broker using TCP as the transport protocol. The core messaging protocol is the Advanced Message Queuing Protocol (AMQP). Clients can have multiple connections to the broker, i.e., channels, which are multiplexed into a single TCP connection to avoid having multiple TCP connections. Clients publish event messages to an exchange, where routing decisions are made depending on the binding information (i.e. the routing_key) stored in the exchange. Binding is a rule that defines the relationship between an exchange and a queue informing the exchange in which messages the queue is interested in. Exchanges support multiple routing patterns using one of the four alternative exchange types:

- *direct* exchange for unicast (and multicast) routing,
- *fanout* exchange for broadcast routing,
- *topic* exchange for routing based on matching of dot separated words with wild cards, and
- *headers* exchange for routing based on matching key value pairs.

A message queue is a FIFO buffer maintained at the broker. A client can subscribe to multiple queues and a queue can be associated with multiple clients. Finally, subscribing clients can consume messages from queues either by using push or pull delivery modes.

Non-AMQP clients connect to the broker using protocol-specific adapters that transparently translate protocol-specific methods into their AMQP equivalents and back. Fig. 3 shows the Web-STOMP adapter that is a simple bridge exposing the STOMP protocol over emulated HTML5 WebSockets. The main purpose of the Web-STOMP adapter is to allow web clients to use the RabbitMQ broker.

AMPQ treats message payload as a byte array and does not constrain its formatting. However, the utilization of a middleware benefits from a common structured data format. Therefore, in the UbiBroker architecture we use text-based JSON message format due to its lightweight nature, widespread adoption in web messaging, and support for parsers in many programming languages.

C. Naming

Clients need to know two categories of names. The first name is the URI pointing to the adapter on the RabbitMQ server, which includes the protocol, username, password, hostname, port and virtual host. The protocol and port vary between adapters. The second category is the routing_keys, i.e., the topics used to route event messages. These names are independent from the client protocol and are defined by the developers. Topics in AMQP are not strictly hierarchical. Each word in a topic is independent, i.e., each word could define an independent aspect that describes the topic. However, as the hierarchical structure is easier to comprehend, we prefer a format where broader categories are listed first and then come the more limiting sub-words defining a specific location or a session, for example.

D. Synchronization

AMQP provides optional transactions for channels to guarantee message delivery between publisher clients and the RabbitMQ broker. The queues handle and deliver the event messages in FIFO order to subscribing clients. Further synchronization needs to be done by the clients if for example correct ordering of causally related events on different queues needs to be guaranteed.

E. Consistency and Replication

The RabbitMQ server can be replicated to improve fault tolerance and/or scalability. Multiple RabbitMQ brokers can be assigned into a cluster. By default most of the internal components are replicated with full ACID (Atomicity, Consistency, Isolation, Durability) capabilities with the exception of the queues. The queues are located in the node where they are created which improves scalability and reliability. If high availability is required, then also the queues can be replicated, in which case the nodes could be structured as having one master and the others are mirrored slaves.

F. Fault Tolerance

Clients can decide upon the persistency of event messages. In the persistent mode the broker stores messages to permanent storage until they are delivered which protects against broker crashes and shutdowns. Message delivery from a client to the broker and from the broker to a client can be guaranteed by two mechanisms. First, a subscribing client can use AMQP acknowledgements to inform the broker that it has received or processed the message. The broker safely holds the messages in a client's queue until it receives an acknowledgment from the client, which guards against client crashes. Second, the publishing of messages can be guaranteed by channel confirms. Both mechanisms can be performed per message or per multiple messages simultaneously.

G. Security

RabbitMQ provides optional confidentiality of communication via the SSL encryption of channels. In terms of controlling access to the broker, clients need to authenticate themselves using username and password. RabbitMQ allows also access control per resource (virtual host, exchange and queue). In order to perform operations on a resource, users must have an appropriate permission granted for it. Only authorized clients can access resources thus restricting misuse between clients.

V. PROTOTYPE IMPLEMENTATIONS

We have developed atop RabbitMQ multiple prototypes of context producers, middleware services and applications that have been deployed on our display network.

A. Context Producers

The Context Producers collect and distribute context information for context aware applications and logging purposes. A particular type of context information is the numbers and demographic attributes (age, gender) of faces detected from the video feed of the overhead camera in a hotspot.

B. Middleware Services

1) Logger Service

Logger Service listens to registered topics and stores the message data into a database for later analysis. The general purpose Logger Service is motivated by the fact that many of the event message payloads are already in the format suitable for system level logs. Logger Service utilizes JSON schemas and stores only valid messages into a database.

2) Multi-Display Session Manager

Multi-display Session Manager uses RabbitMQ to synchronize multiple hotspots into a shared and distributed multi-user application session. Each hotspot has its own state and context, which Session Manager monitors. If a multi-user application is to be launched, Session Manager interrupts local sessions and establishes a shared and distributed session across multiple hotspots.

C. Applications

1) Ubidoku

Ubidoku is a multiplayer Sudoku, which is played on multiple hotspots simultaneously. Multi-Display Session Manager establishes a shared and distributed Ubidoku session on multiple hotspots. RabbitMQ is used to synchronize gaming events between the separate components of the distributed user interface.

2) Map

Map has also a distributed user interface that comprises of a hotspot and a mobile phone. The hotspot shows a large interactive map while the mobile phone has a simple interface for entering text strings to search for points of interest on the map. The search strings are sent via RabbitMQ.

3) UbiLibrary

UbiLibrary is a service tailored for and deployed at the hotspot placed at the Oulu City Library. UbiLibrary dynamically configures its interface according to the current interaction mode and the demographic information of detected faces: hidden in the passive mode (no-one is interacting with the hotspot nor within visual proximity), show context (demographic) filtered book recommendations in the subtle mode (face has been detected from the video feed of the overhead cameras) to entice the user to interact; and full user interface in the interactive mode (user is interacting with the hotspot).

VI. PERFORMANCE EVALUATION

A. Performance Evaluation

We have evaluated the performance of RabbitMQ both in a lab and in a real world setting. The RabbitMQ server process was installed on a virtual machine running CentOS 6.4 with two 2.7 GHz CPU's, 4 GB RAM, 40 GB disk and 1 Gbps LAN. The AMQP was used as the messaging protocol with topic-based routing. The performance was measured as the average message delivery rate per subscriber per second during a test lasting 10 minutes.

1) Lab evaluation

In the lab evaluation the publishers and subscribers resided in the same 1 Gbps Ethernet LAN with the broker. Subscribers were configured to acknowledge messages after every 50 received messages. The first lab test assessed message throughput for different numbers of subscribers bound to a single topic and for different message payload sizes. Fig. 4 summarizes the results. As expected, the fastest per-subscriber throughput was achieved with a single subscriber and adding subscribers decreases throughput. Different message payloads under 2^9 bytes did not have a significant effect on throughput. First noticeable drop in the message delivery rate can be seen between 2^10 and 2^11 byte payloads, where the total message size exceeds Ethernet's 1500-byte MTU and packets begin fragmenting. Second drop can be seen between payloads 2^12 and 2^13 bytes, when the virtual machine's host computer's network interface controller started to limit packet flow to around 120 000 packets per second. RabbitMQ's flow control restricts publishers from publishing events faster than they can be processed.

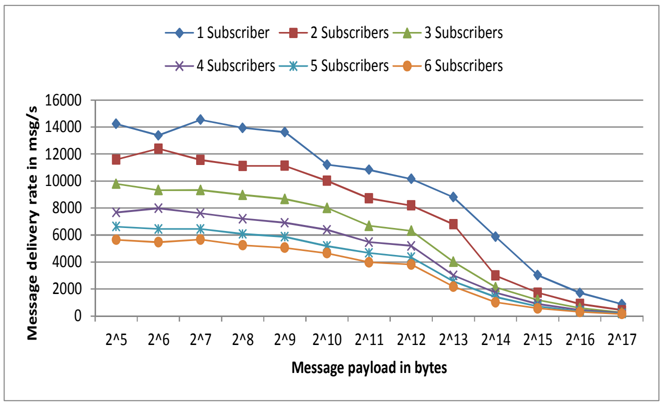

Fig 4. Throughput for different numbers of subscribers bound to a shared topic and different message payload sizes.

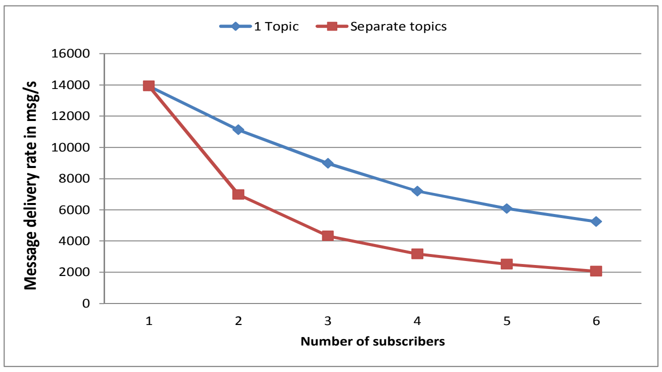

Fig. 5. Throughput for different numbers of subscribers with one shared topic and with separate individual topics.

The second lab test assessed the effect of having multiple topics and subscribers involved in the routing process so that each client subscribed to a separate topic, i.e. each topic had only one subscriber listening to it at any given point of time. The message payload was set to 256 bytes. Fig. 5 shows the throughput as a function of the number of subscribers/topics, the first test providing the baseline for multiple subscribers subscribing to one shared topic.

2) *Real world evaluation*

In the real world evaluation the RabbitMQ clients were deployed on the actual hotspots in the city of Oulu. The hotspots have fiber connections with a nominal bandwidth of 10 Mbps. The clients were installed on virtual machines running CentOS 5.6 with 2.53 GHz CPU, 512 MB RAM and 30 GB disk. The resources of the virtual machines were also used by the middleware software of the hotspots and the hotspots served the users normally during the evaluation.

In the first test each client published messages size of 175 bytes into the same topic at a rate of 1000 msg/s whereas only one client subscribed to the topic to simulate the Logger Service situation. The highest recorded delivery rates without and with message persistence were 12976 msg/s and 6805 msg/s, respectively.

The second test simulated a future scenario of a significantly higher load where in total 333 producers and 340 subscribers were scattered on six different hotspots. The producers of a particular hotspot published messages to the same exchange but on different topics. Each subscriber had its own queue and subscribed to a separate topic, thus, overall, the system had 60 different topics. The producers published messages in a steady rate of 5 msg/s and the subscribers auto-acknowledged each message separately. The publishers were able to publish up to 1043 msg/s in total, which resulted in 5513 msg/s delivered to subscribers in total. The clients were not able to produce the messages at the given rate due to the constrained resources of hotspot virtual machines. Also, the test needed to be cancelled prematurely due to overloading one of the network routers, which affected the whole network performance.

B. *Real World Deployment*

A single RabbitMQ server instance has been running continuously for six months serving the aforementioned context producers, middleware services and applications deployed across our network of 18 hotspots over a metropolitan area network. So far, there has not been a single availability issue due to the message broker being unstable. During the six month deployment we have had one incompatibility issue with the clients. The WebSocket implementation in the Web-STOMP protocol adapter was not compatible with a new Chrome browser release. Chrome browser releases rigorously implement the latest version of the evolving WebSockets specification and thus invalidated the not so rigorous Web-STOMP clients. We reported the bug to RabbitMQ developers, who have fixed it since then. While waiting for the Web-STOMP plugin to be updated, we simply modified the processing of WebSocket headers in the STOMP library as a work around.

VII. Discussion

A. *Decoupling*

In related middleware the choice of the messaging pattern is divided between content based routing [5, 6, 7, 8] and topic based routing [4, 8, 9, 10]. AMQP also supports multiple messaging patterns, but not content based as such. The headers exchange comes close as it allows key value pairs to be added into message headers and using them for dynamic filtering of messages. However, message body is not processed upon. This is motivated by the fact that message bodies can contain large amounts of data and processing them to make routing decisions would decrease the overall performance of the messaging system. Therefore, the topic and headers exchanges of AMQP protocol provide a good compromise between the decoupling of clients and performance.

B. *Simplicity and interoperability*

Setting up messaging between clients using RabbitMQ is surprisingly simple. A client only has to include the adapter library and a couple lines of program code to start publishing and subscribing events. The development overhead is thus minimal and considerably smaller than in achieving similar functionality from scratch. Since the deployment of the UbiBroker architecture, we have utilized the communication middleware with web clients and Java components. However, we foresee much more heterogeneous clients in the future, which will take up the full power of multiple client implementations available for RabbitMQ. The same thing applies to multi-protocol support, since a common toolkit for building pervasive display applications is not yet a reality.

Among the application developers involved with developing applications for our hotspots, there have been also third parties with no previous experience of RabbitMQ and our hotspot middleware. We provided them with a short online documentation with simple code snippets as examples of using the UbiBroker. The documentation provided by RabbitMQ works as an additional source for developers wanting an in depth description of features and/or willing to use different features of RabbitMQ not defined by us. None of the third party developers needed our assistance in using UbiBroker, mainly due to the simplicity of the RabbitMQ client model and the vast documentation available from the RabbitMQ support community.

C. *Stability*

The RabbitMQ broker has proven to be very stable. This is crucial as it reduces maintenance and increases trust among developers. Without trust they easily resort to ad hoc solutions, which exists plenty in the web technologies. Ad hoc solutions can easily become isolated or incompatible with the rest of the system and difficult to maintain afterwards.

D. *Performance*

Performance evaluations revealed hard upper limits for the message delivery capacity of the proposed architecture. Increasing the number of subscriber queues affects the overall performance more than increasing the number of subscribers in the system. Therefore, for optimum performance it is better to share queues, i.e., have similar topics for multiple clients when applicable. Also, using persistent messages decreases the performance which can be expected. In any case, based on these evaluations, even a single RabbitMQ message broker will be able to satisfy the performance needs of our hotspot network in the future. This conclusion is supported by the second part of the real world evaluation which revealed that the performance of the hotspots and the network were the limiting factor instead of the RabbitMQ broker.

VIII. Conclusion

We presented the UbiBroker, event-based communication architecture for a pervasive display network. The design requirements were elicited from our experiences of operating a real world display network for many years and from the third party developers developing applications for the display network. The UbiBroker architecture is based on the off-the-self RabbitMQ open source message broker. We evaluated the stability and performance of the middleware in a lab setting and in a real world deployment. We have learned that any performance bottlenecks are more likely going to originate from the operating environment than from the broker itself, even with a single broker instance.

In our ongoing work we are looking at implementing a seamless JSON based state storage service for automatic storage and easy retrieval of system state. This would allow any late joining clients to construct the system state without having to wait for events to arrive first [6]. Another interesting challenge is integrating media intensive data such as web cam streams to the middleware, whose raw data streams cannot be transmitted through the message broker for practical reasons.

Acknowledgments

The financial support of the Academy of Finland, the Finnish Funding Agency for Technology and Innovation, the Council of Oulu Region, the European Regional Development Fund, the City of Oulu, and the UBI (UrBan Interactions) consortium is gratefully acknowledged.

References

[1] T. Ojala, V. Kostakos, H. Kukka, T. Heikkinen, T. Lindén, M. Jurmu, S. Hosio, F. Kruger and D. Zanni, “Multipurpose interactive public displays in the wild: Three years later,” Computer, vol. 45, no. 5, pp. 42-49, 2012.

[2] T. Ojala, H. Kukka, T. Lindén, T. Heikkinen, M. Jurmu, S. Hosio and F. Kruger, “UBI-hotspot 1.0: Large-scale long-term deployment of interactive public displays in a city center”, Proc. ICIW’10, Barcelona, Spain, pp. 285-294, 2010.

[3] S. Tarkoma, J. Kangasharju, T. Lindholm, and K. Raatikainen, “Fuego: Experiences with mobile data communication and synchronization,” Proc. PIMRC’06, Helsinki, Finland, pp. 1–5, 2006.

[4] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R.H. Campbell and K. Nahrstedt, “A Middleware Infrastructure for Active Spaces,” IEEE Pervasive Computing, vol. 1, no. 4, pp. 74-83, 2002.

[5] C. Greenbagh, “Equip: a software platform for distributed interactive systems,“ Technical Report, Equator-02-002, Equator, 2002

[6] B. Johanson, A. Fox and T. Winograd, “The Interactive Workspaces project: experiences with ubiquitous computing rooms,” IEEE Pervasive Computing, vol. 1, no. 2, pp. 67- 74, Apr-Jun 2002.

[7] R. Grimm, J. Davis, E. Lemar, A. Macbeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble and D. Wetherall, “System support for pervasive applications,” ACM Trans. Comput. Syst., vol. 22, no. 4, pp. 421-486, Nov 2004.

[8] T. Sivaharan, G. Blair and G. Coulson, “Green: A configurable and reconfigurable publish-subscribe middleware for pervasive computing,” In: On the Move to Meaningful Internet Systems: CoopIS, DOA, and ODBASE, Springer, pp. 732–749, 2005.

[9] E. Aitenbichler, J. Kangasharju and M. Muhlhauser, “MundoCore: A light-weight infrastructure for pervasive computing,” Pervasive and Mobile Computing, Elsevier Science Publishers B. V., vol. 3, no. 4, pp. 332-361, August 2007.

[10] A. Erbad, M. Blackstock, A. Friday, R. Lea and J. Al-Muhtadi, “MAGIC Broker: A Middleware Toolkit for Interactive Public Displays,” Proc. PerCom’08, Washington, DC, USA, pp. 509-514, 2008.

[11] T. Ojala, H. Kukka, T. Heikkinen, T. Lindén, M. Jurmu, F. Kruger, S. Sasin, S. Hosio and P. Närhi, “Open urban computing testbed”, Proc. TridentCom’10, Berlin, Germany, pp. 457-468, 2010.

[12] T. Lindén, T. Heikkinen, T. Ojala, H. Kukka and M. Jurmu, “Web-based framework for spatiotemporal screen real estate management of interactive public displays”, Proc. WWW’10, Raleigh, NC, USA, pp. 1277-1280, 2010.

[13] T. Heikkinen, T. Lindén, M. Jurmu, H. Kukka and T. Ojala, “Declarative XML-based layout state encoding for managing screen real estate of interactive public displays”, Proc. MUCS’11, Seattle, WA, USA, pp. 312-317, 2011.

[14] A. S. Tanenbaum and M. Steen, “Distributed Systems: Principles and Paradigms (2nd ed.)”, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[15] RabbitMQ project http://www.rabbitmq.com/.