Proceedings of the 5^{th} MiNEMA Workshop

Middleware for Network Eccentric and Mobile Applications

11-12 September 2007, Magdeburg, Germany

We thank the European Science Foundation (ESF) for funding and supporting the 5^{th} MiNEMA Workshop in Magdeburg

Contents

Table of contents **2**

Foreword **4**

Organisation **6**

Session 1: Streaming and Multicast **8**
Building multicast trees in ad-hoc networks,
Raphaël Kummer, Peter Kropf, Pascal Felber 8
A Gambling Approach to Scalable Resource-Aware Streaming,
Mouna Allani, Benoît Garbinato, Fernando Pedone, Marija Stamenković 14
Removing Probabilities to Improve Efficiency in Broadcast Algorithms,
Hugo Miranda, Simone Leggio, Luís Rodrigues, Kimmo Raatikainen 20

Session 2: P2P Systems and Overlay Networks **26**
GossipKit: A Framework of Gossip Protocol Family,
Shen Lin, François Taïani, Gordon Blair 26
Enabling Cyber Foraging for Mobile Devices,
Mads Kristensen 32

Session 4: Publish/Subscribe **38**
Strategies for implementing Peer-to-Peer Publish/Subscribe with Persistent Events in Wireless Settings, *Eugster Patrick, Benoît Garbinato, Adrian Holzer, Jun Luo* 38
Probabilistic Publish/Subscribe in Mobile Ad Hoc Networks,
José Mocito, José Côrte-Real, Luís Rodrigues 44
Predictive Publish/Subscribe for Delay Tolerant Mobile Ad Hoc Networks,
Paolo Costa, Cecilia Mascolo, Mirco Musolesi, Gian Pietro Picco 48

Session 5: Architectures and Frameworks **54**
Towards a Peer-to-peer Middleware for Context Provisioning in Spontaneous Networks,
Tuan Dung Nguyen, Siegfried Rouvrais 54
Semantic Middleware for Designing Collaborative Applications in Mobile Environment,
Lamia Benmouffok, Jean-Michel Busca, Marc Shapiro 58
Handling membership dynamicity in service composition for ubiquitous computing,
Jeppe Brønsted 62

Session 6: Wireless Sensor Applications **68**
End-to-end middleware for distributed sensor applications,
Nelson Matthys, Sam Michiels, Wouter Joosen, Pierre Verbaeten 68
Using COSMIC – A real world case study combining virtual and real sensors,
Michael Schulze, Sebastian Zug 74

Index of authors **78**

Foreword

On behalf of the organisation team, and the MiNEMA project, I would like to welcome you to the 5^{th} MiNEMA Workshop. MiNEMA is an European Science Foundation (ESF) Scientific Programme aiming to bring together European groups from different communities working on middleware for mobile environments. The programme intends to foster the definition and implementation of widely recognised middleware abstractions for new and emerging mobile applications.

This year's MiNEMA workshop again promises to be extremely interesting, with 13 accepted contributions from 12 different countries: 10 within Europe (Switzerland, Portugal, UK, Denmark, Germany, Finland, Belgium, France, the Netherlands, and Italy), and two in North America (USA and Canada). The accepted papers cover a wide range of exciting topics, from Multicast Protocols and Overlay Networks through to Publish/Subscribe Systems and Wireless Sensor Applications.

I would like to use this place to thank warmly the Programme Committee Members for their dedicated work, which allowed for a very smooth reviewing process. I would also like to thank the Organisation Team in Magdeburg for their perfect logistics, with a special mention to Thomas Kiebel for his tremendous support in all things technical, and in particular for setting up and maintaining the workshop web-site. Also thanks to Prof. Jörg Kaiser and Prof. Luís Rodrigues for their advise and help in arranging the workshop programme. Last but not least, thanks to the MiNEMA project and the European Science Foundation for providing the funding for this event.

François Taïani
5^{th} MiNEMA Workshop Programme coordinator

Organisation

Organisation / Registration

Thomas Kiebel, OvGU-MD, Madgeburg, Germany

Local Arrangements

Michael Schulze, OvGU-MD, Magdeburg, Germany
Sebastian Zug, OvGU-MD, Magdeburg, Germany

Programme Committee

Michael Schulze, OvGU-MD
François Taïani, Lancaster University
Jeppe Brønsted, Aarhus Universitet
Danny Hughes, Lancaster University
Nico Janssens, Katholieke Universiteit Leuven
Boris Koldehofe, Universität Stuttgart
Sam Michiels, Katholieke Universiteit Leuven
Hugo Miranda, Universidade de Lisboa
Aline Senart, Trinity College Dublin
Alexander Tyrrell, DoCoMo Euro-Labs
Kari Heikkinen, Lappeenronta Univerity

Building Multicast Trees in Ad-hoc Networks

Raphaël Kummer
raphael.kummer@unine.ch

Peter Kropf
peter.kropf@unine.ch

Pascal Felber
pascal.felber@unine.ch

Computer Science Department
University of Neuchâtel
Emile-Argand 11, CP 158,CH-2009 Neuchâtel, Switzerland

***Abstract*—Multicast trees are used in a variety of applications, such as publish/subscribe systems or content distribution networks. Existing algorithms for ad-hoc networks typically produce inefficient multicast trees as they require many nodes to act as relay even though they are not part of the multicast group. In this paper, we propose an algorithm for building efficient multicast trees that strives to minimize the number of non-member relay nodes and balance their degree. Our algorithm relies upon a lightweight distributed hash table (DHT), proposed as part of previous work, to construct and optimize the multicast trees. We evaluate the efficiency and scalability of our algorithm by simulations with various network configurations and sizes.**

I. Introduction

Multicast communication is widely used in distributed applications for efficiently delivering data from one source to a potentially large group of destinations. Multicast algorithms typically create and maintain distribution trees spanning all destinations in a way that messages are transmitted over each link of the network only once.

In ad-hoc networks, communication between remote nodes usually requires multiple hops via relay nodes, which complicates the task of building efficient multicast trees. Most importantly, one must minimize the number of nodes that act as relays but are not part of the multicast group (i.e., the set of destinations), as message relaying consumes scarce CPU cycles and energy.

As centralized tree construction algorithms do not scale well and offer but limited reliability, we investigate, in this paper, decentralized strategies where nodes are organized in a peer-to-peer (P2P) configuration. Broadly speaking, P2P overlay networks can be divided in two main categories: structured and unstructured ones. An ad-hoc network spontaneously forms an unstructured network, where physical proximity defines the neighborhood of nodes. To disseminate information to a subset of nodes, a source can simply flood the network. One can also use flooding to construct shortest-path spanning trees. Yet, such trees make quite inefficient usage of the (limited) resources as we will discuss shortly. In general, flooding is not a good approach for designing scalable algorithms.

Superimposing a structured overlay on top of the physical network can help to construct efficient multicast trees. This approach has been used, for instance, by Scribe [10]: multicast trees are rooted at "rendez-vous nodes" managed by the underlying Pastry [9] distributed hash table (DHT). Nodes interested in joining the multicast group route a request via Pastry and connect to the first member reached on the way to the rendez-vous. While this strategy is effective in wired networks, it cannot be easily transposed to ad-hoc or mobile networks.

In this paper we propose a novel algorithm for building multicast trees in ad-hoc networks using a lightweight DHT overlay [7]. The ad-hoc DHT provides efficient lookup in a logical space by exploiting the physical proximity of peers and the properties of the wireless broadcast communication medium. As in Scribe, the DHT is used for localizing the source of the multicast and one tree is created per source. We use several techniques to connect joining nodes to an existing member that is physically close and in the direction of the source. We propose various extensions to reduce the number of non-members implicated in the relaying of messages.

We have performed extensive simulations of our algorithm using different scenarios, but without taking into account neither mobility nor churn. Results indicate that our algorithm does produce efficient multicast trees, with a limited number of non-member relays, and that it scales well to large networks.

The remainder of this paper is organized as follows. Section II discusses related work. We detail our algorithm in Section III and present results from experimental evaluation in Section IV. Finally, Section V concludes.

II. Related Work

Several P2P approaches have been suggested for multicast in ad-hoc networks. Some use a logical overlay substrate for source localization while others rely on flooding. We cannot discuss all approaches due to space limitation (see [3] for a good survey).

Different logical structures can help tree construction in ad-hoc networks. MZR [5] relies upon zone routing protocol [6] to build a multicast tree. The nodes in ZRP define a zone around them and maintain proactively routes to all nodes within that zone. A reactive route discovery protocol is used when the destination is outside the sender's zone. When a source has data to multicast, it advertises it to all the nodes in its zone, then extends the tree to nodes at the border of other zones. An interested node has to answer to the source and, when the message reaches a group member, a branch is created. While the zone structure contains the flooding

necessary to build the tree, it still floods the whole network zone by zone. Its bandwidth and energy requirements are thus significant. In addition, this protocol doesn't provide generic lookup facilities as it is done in our algorithm by the DHT.

XScribe [8] and Georendezvous [2] use a DHT to support the multicast tree creation. XScribe is based on CrossROAD [4], a cross layer DHT providing the same features as Pastry, but based on a proactive routing protocol and with lower bandwidth requirements. XScribe exploits the DHT routing capacities to distribute multicast messages. Unlike in our system, each source has to know all the members of the group and sends the multicast messages directly to each member by unicasting messages. Therefore, the approach does not scale well, nor does it try to optimize resource consumption (by minimizing the number of relay nodes).

Georendezvous relies on CHR [1], a specialized ad-hoc DHT that groups nodes in clusters accordingly to their physical location. The DHT is used to efficiently localize the cell responsible for a group. The nodes in this cell manage the membership for the group and forward the multicast to all the members. Membership management is centralized in a cell containing multiple nodes, which are also responsible for distributing multicast messages. Our approach is expected to produce more efficient multicast trees, with lower energy and bandwidth requirements, and to offer better scalability.

III. The Tree Construction Algorithm

In ad-hoc networks, multicasting messages to a group of nodes can be achieved essentially in two different ways. The first approach relies on flooding to discover a source or to build a tree, as it is done for example in MAODV [11]. As already mentioned, this method does not scale well and may lead to an overload of the network due to the flooding mechanism applied. Moreover it forces all the nodes in the network to participate whether they are interested in the content provided or not. The second approach uses directed search to locate the data sources and construct the trees, as done for instance by Scribe [10]. This method is more bandwidth and energy efficient—two important considerations for mobile devices—but cannot easily be applied to ad-hoc networks where communication is multi-hop and physical awareness is an essential consideration.

To construct a multicast tree, we first lookup the data source, which acts as rendez-vous point for the multicast group. Hence it is important that data consumers can locate efficiently the source. As in Scribe, we use a DHT for that purpose because it provides suitable facilities for efficient lookup (see Figure 1) without flooding the network and because it scales well to large number of nodes. While the DHT allows us to easily localize the source, lookup messages in the overlay do not follow the shortest path in the underlying ad-hoc network. As we shall see, this extra level of indirection permits building better multicast trees because they offer more connection alternatives to joining nodes. As a result, the tree has less non-member relays and the average degree remains reasonably low.

Our algorithm strives to construct multicast trees that use member nodes as relays. Membership is handled in a decentralized way as a joining node might connect to the tree without the source knowing it. Consequently, the load on the source is reduced and we can avoid bottlenecks. One tree is constructed per active data source. We assume best effort delivery for multicast messages; additional mechanisms could be easily incorporated to implement reliable delivery.

A. The Distributed Hash Table (DHT)

The ad-hoc DHT used by our multicast tree construction algorithm consists of a minimalist logical overlay where nodes are organized into a ring and ordered according to their unique identifiers. Unlike most others DHTs, we do not maintain long-range neighbors in the logical space. Instead, long-range links are spontaneously found during the lookup in the physical neighborhoods of the traversed nodes.

The DHT maps keys to nodes in the P2P infrastructure and the logically nearest node on the ring (i.e., in the identifier space) is responsible for the key. We use this mapping facilities to identify multicast groups and locate sources. For more details, we refer the interested reader to [7].

B. The Connection Algorithm

The main principle of the algorithm consists in building a multicast tree using DHT lookups, and then apply various optimizations to improve the tree. To join the multicast tree, a node routes a request to the identifier associated with the source of the multicast (group identifier). When receiving a join request on its way to the source, a node checks if it is member of the group. If that is not the case, it simply forwards the request to the next node towards the source (according to the DHT lookup protocol). Otherwise, the current node also replies to the joining node and proposes itself as parent in the multicast tree. Thus, the joining node receives, most of the time, several potential parents (but at least one because the request is always routed towards the source).

To join the distribution tree as soon as possible, the requester connects to the first parent it receives. Thereafter, if it receives further responses from potential parents, it changes only if (1) the distance to the new parent is shorter than to the old parent and the new distance to the root is no more than twice the old distance; or (2) the new parent is at the same distance as the old one but the distance to the root has shortened. At the end of the process, the node is connected to the multicast tree with the node that it considers as being the best parent.

The parent selection method has been designed empirically. The design of an optimal decision heuristic still remains an open question.

With this straightforward algorithm, many nodes are connected to the source with direct paths and many non-member peers are located on the paths from the source to the destinations. We shall now present the methods that we have developed to improve the tree structure and to reduce the number of non-member nodes involved in multicasting. In the three methods presented below, the first one is used during the

lookup, while the second and third ones rely on information added by nodes to the multicast message for reorganizing the tree. These optimizations should be combined for increased efficiency.

C. Opt. #1: Finding More Potential Parents

The nodes that are not involved in a communication listen to the requests (radio communications are broadcasted) and cache some of the gathered information. In particular, a node listening to a join message will send a response to the requester if it is a member of the joined multicast group. Listening to communications is typically a cheap operation as it does not generate extra messages, yet it often allows to improve the structure of the tree, as we shall see in simulations. In particular, listening to messages avoids pathological situations where two multi-hop requests cross but do not traverse a common node.

D. Opt. #2: Finding Better Parents

When a message is multicast, it usually traverses several non-member nodes between a parent and a child in the logical multicast tree. Two messages from one node to two different nodes may traverse a certain number of common relays. In fact, if two messages follow the same physical path, their destinations are likely to be in the same area of the network. Consequently, one can inform one of the nodes that the other one is possibly a better child or parent in the multicast tree. This optimization is interesting because, by reducing the distance between parents and children in the tree, we also reduce the network load.

We propose a solution where no additional messages are needed. When a node relays a multicast message (uniquely identified by a group and a message identifier) it memorizes the group, the message identifier, and the destination address. If a node receives the same message (same group and message identifier) intended for another destination, the node adds the previously memorized address to the message before forwarding it. If more than one node have an address to add, only the last one is kept in the message (space overhead is negligible).

The receiver of a multicast checks if the address of a relay has been added to the message. If so, the receiver sends a message to the relay and proposes to become its parent or its child. A reconfiguration takes place only if (1) the new parent is better than the previous one according to the criteria discussed above, and (2) the new parent is not a descendant of the new child in the multicast tree. This validity check avoids partitioning the tree and losing the connection with the source.

E. Opt. #3: Removing Redundant Parents

As connections between a node and its children in the multicast tree are typically multi-hop, it may happen that a node is both a member of the logical tree as well as a relay along a multi-hop path between a node and one of its child. Obviously, the resulting structure is sub-optimal and this situation should be avoided, as the affected node receives the same message from more than one parent.

To deal with this situation, a node which detects that it receives the same message from multiple paths (both as a relay or as a member of the tree) keeps only one connection with the closer parent and discards the others. If the node is on a multi-hop path from a parent to its child, it may need to disconnect the child from its former parent and add it to its own children, or it may need to promote itself as new inner node of the tree along that path.

In either case, one physical path is discarded and the number of non-member relays is reduced. Moreover, the tree better maps to the underlying topology and multicast efficiency is improved.

IV. Experimentations

To evaluate our multicast tree construction algorithm, we have extended the experimental system previously used for testing our ad-hoc DHT [7]. The simulator is divided into three layers: the routing layer, the DHT, and the multicast layer. These layers communicate together through dedicated methods. We assume that the routing protocol (a simplified version of AODV) is able to process messages faster than the upper layers and does not represent a bottleneck.

In this paper, we only simulate simple scenarios. A set of nodes are placed randomly (according to a uniform distribution) in a rectangular area. As we study only static scenarios (no mobility nor churn), the position of the nodes does not change along a simulation. Two nodes are connected with each other if they are within communication range, defined by a given radius. We ensure, when placing nodes, that the resulting graph is connected.

DHT identifiers are randomly assigned to nodes, and a group identifier (mapped to a single source) is randomly selected in the same space. We let randomly selected nodes join that group. We first "warm up" the underlying DHT by performing several lookups (100 in our tests); this allows the nodes to populate their routing tables (see [7] for details). Then, between one and three nodes join the tree at each simulation step until the desired number of members is reached. At the same time, multicast messages are sent continuously every one to five simulation step throughout the tree. We stop the simulation once the tree structure stabilizes.

We experimented different configurations and network sizes:

- *Network size:* $1,000$; $2,500$; $5,000$; $10,000$;
- *Connectivity:* the average number of physical connections of a node (network density) varies between 13 and 16;
- *Multicast members:* unless specified, 10% of all nodes join the tree;
- *Warm-up:* 100 random lookups are performed in the DHT before constructing the multicast trees.

Using these configurations, we have evaluated different versions of the multicast tree construction algorithm:

- *Basic:* the construction algorithm with no optimizations;
- *Opt. #1:* during join, member nodes listen and propose themselves as parent when applicable;

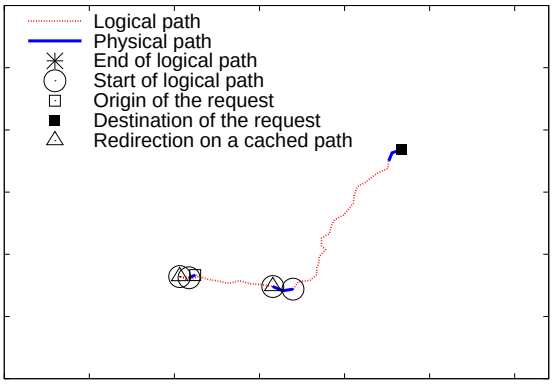


Fig. 1. Path of a request routed by the DHT.

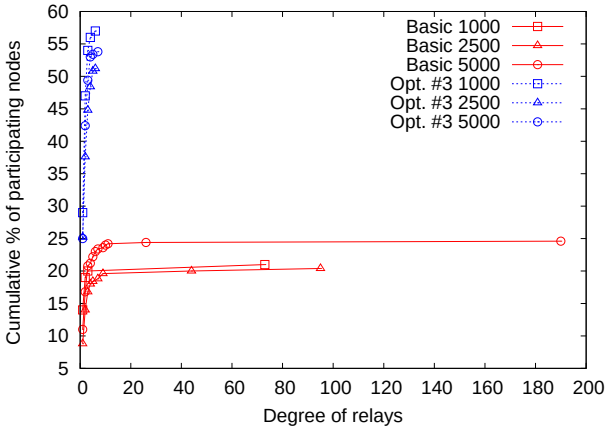


Fig. 2. Cumulative percentage of members acting as relays as a function of their degree.

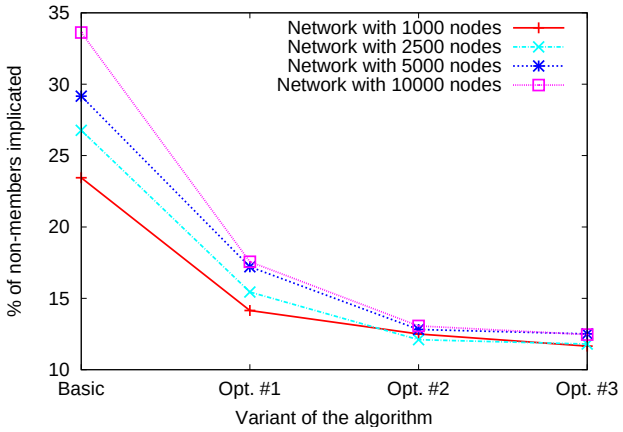


Fig. 3. Percentage of non-members implicated as relay.

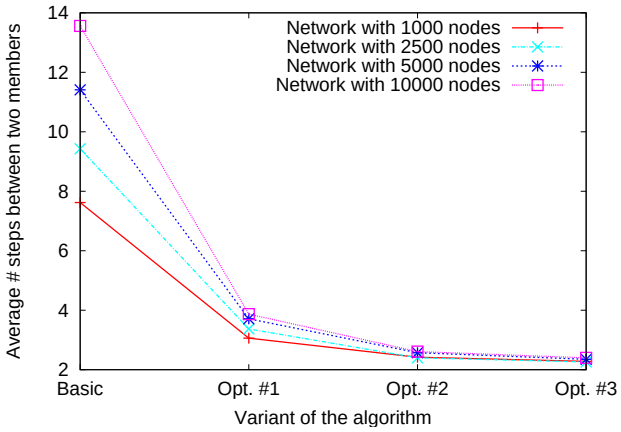


Fig. 4. Average number of physical steps separating two group members.

- *Opt. #2:* when sending messages, we try to identify common sub-paths and reconnect nodes to better parents;
- *Opt. #3:* we prevent nodes to receive duplicate messages when acting both as member and relay in a multicast tree.

We have also experimented with building the tree without using the DHT. Optimizations are always cumulated in the presented results (i.e., optimization n also incorporates optimizations $m < n$.

We are essentially interested in evaluating the tree structures, the degree of implicated nodes, the number of non-member nodes required to route packets, as well as the relative distance between two members. Therefore, we did not provide a comparative study against other algorithms that mostly focus on packet loss and delay [8] or path length overhead and latency [2].

A. Results

We do not study the lookup performances here as it has been extensively done in [7]. However there is an interesting fact to notice about the use of the DHT with our multicast algorithm: although DHT lookups have a sense of direction and can usually locate a node efficiently, they follow a path that is longer than direct AODV-like routing and allow joining nodes to locate better parents for connecting to the tree. Indeed, as shown in Table I, the tree obtained when using the DHT is systematically better than without, because it involves less non-member nodes. Note that the path length increase introduced by the DHT is small (Figure 1 illustrates a sample path; refer to [7] for details).

Network size (#peers)	500	1,000	5,000
No-DHT	13%	16%	17.07%
DHT	10.6%	11.25%	12.65%

TABLE I
PERCENTAGE OF NON-MEMBER NODES IMPLICATED IN MULTICAST TREE.

Figure 2 shows the cumulative percentage of group members implicated in message relaying as a function of their degree. Clearly, in the tree produced by our basic algorithm, less than 30% of the group members participate to the message distribution. In other words, more than 60% are only leaves. Moreover, some nodes have quite a high degree (most notably the source). In contrast, when the tree structure has been

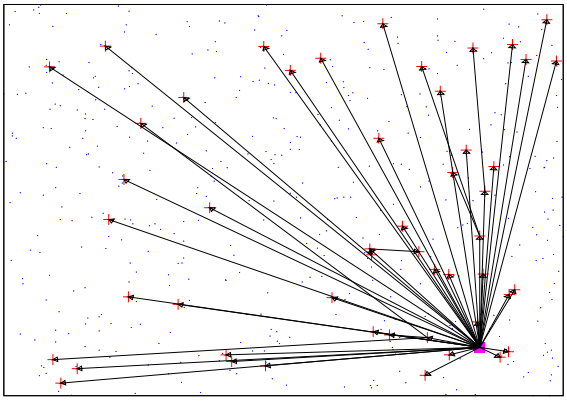


Fig. 5. Tree built by the basic algorithm.

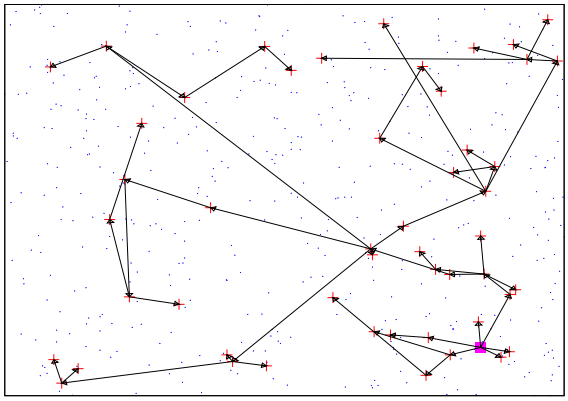


Fig. 6. Tree built by the improved algorithm.

improved by the different methods presented in Section III, more than 50% of the member nodes help distributing content. The degree of these relay nodes does not exceed 7 for all the considered network sizes. Hence, our algorithm contributes to evenly distribute the load between the nodes.

As a result of the increased number of members relaying messages, we expect that the number of non-member nodes acting as relay will decrease. This is confirmed by Figure 3: we observe a significant improvement, from $23 - 33\%$ non-member nodes to less than 13% for all network sizes.

Our algorithm builds a tree in which nodes that are physically close can discover each other and connect by just passively gathering information as messages are multicast. Thus, the physical path length between two consecutive members is small, as demonstrated by Figure 4. Consequently, the tree is closely mapped to the underlying physical topology.

The effect of our optimizations is visible in Figures 3 and 4, where we observe clear improvements on the physical distance between group members and on the percentage of non-members implicated. A more intuitive visualization is given in Figures 5 and 6, which show the multicast tree constructed from the same network without and with optimizations. Clearly, the second tree structure is much more effective and better maps to the underlying topology.

Finally, one should note that when applying optimizations, the physical distance between two nodes and the number of implicated non-members relays are nearly the same for all the considered network sizes. This supports the claim for the scalability of our approach.

V. Conclusion

Although much work has been done on the problem of multicast in ad-hoc networks, most of the solutions use some form of flooding or centralized solutions that are not scalable. In this paper, we presented an algorithm for the construction of efficient multicast trees using an underlying ad-hoc DHT overlay. Our algorithm strives to create trees that involve as few non-member nodes as possible, with short inter-node paths and good scalability. Simulation results indicate that our algorithm meets these objectives in the considered network settings.

References

[1] F. Araujo, L. Rodrigues, J. Kaiser, C. Liu, and C. Mitidieri. Chr: A distributed hash table for wireless ad hoc networks. In *ICDCSW '05: Proceedings of the Fourth International Workshop on Distributed Event-Based Systems (DEBS) (ICDCSW'05)*, pages 407–413, Washington, DC, USA, 2005. IEEE Computer Society.

[2] N. Carvalho, F. Araujo, and L. Rodrigues. Reducing latency in rendezvous-based publish-subscribe systems for wireless ad hoc networks. In *ICDCSW '06: Proceedings of the 26th IEEE International ConferenceWorkshops on Distributed Computing Systems*, page 28, Washington, DC, USA, 2006. IEEE Computer Society.

[3] X. Chen and J. Wu. *Multicasting techniques in mobile ad hoc networks*. CRC Press, Inc., Boca Raton, FL, USA, 2003.

[4] F. Delmastro. From pastry to CrossROAD: CROSS-layer ring overlay for AD hoc networks. In *PERCOMW '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 60–64, Washington, DC, USA, 2005. IEEE Computer Society.

[5] V. Devarapalli and D. Sidhu. MZR: a multicast protocol for mobile ad hoc networks. volume 3, pages 886 – 891, 2001.

[6] Z. Haas. A new routing protocol for the reconfigurable wireless networks. In *IEEE 6th International Conference on Universal Personal Communications Record*, volume 2, pages 562–566, San Diego, CA, USA, 1997.

[7] R. Kummer, P. Kropf, and P. Felber. Distributed lookup in structured peer-to-peer ad-hoc networks. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4276 of *Lecture Notes in Computer Science*, pages 1541–1554. Springer Berlin / Heidelberg, 2006.

[8] A. Passarella, F. Delmastro, and M. Conti. Xscribe: a stateless, cross-layer approach to p2p multicast in multi-hop ad hoc networks. In *MobiShare '06: Proceedings of the 1st international workshop on Decentralized resource sharing in mobile computing and networking*, pages 6–11, New York, NY, USA, 2006. ACM Press.

[9] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001 : IFIP/ACM International Conference on Distributed Systems Platforms*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350, Heidelberg, Germany, 2001. Springer Berlin / Heidelberg.

[10] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[11] E. M. Royer and C. E. Perkins. Multicast operation of the ad-hoc on-demand distance vector routing protocol. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 207–218, New York, NY, USA, 1999. ACM Press.

A Gambling Approach to Scalable Resource-Aware Streaming*

Mouna Allani† Benoît Garbinato† Fernando Pedone‡ Marija Stamenković‡

†University of Lausanne ‡University of Lugano

Abstract

In this paper, we propose a resource-aware solution to achieve reliable and scalable stream diffusion in a unreliable environment. Our solution is resource-aware in the sense that it limits the memory consumption (by strictly scoping the knowledge each process has about the system) and the bandwidth available to each process (by assigning a fixed quota of messages to each process). The underlying stream diffusion algorithm is based on a tree-construction technique that dynamically distributes the load of forwarding stream packets among processes, based on their respective available bandwidths.

Keywords: large-scale systems, reliable streaming, resource awareness.

1 Introduction

Reliable stream diffusion under constrained environment conditions is a fundamental problem in large-scale distributed computing. Many internet systems rely on streaming multicast; consequently, their performace depends on the performance of the underlying streaming mechanism. Environment conditions are constrained by the reliability and the capacity (usually limited) of its components (nodes and links). Nodes and communication links can fail, unexpectedly ceasing their operation and dropping messages, respectively. Moreover, real-world deployment does not offer nodes and links infinite memory nor infinite bandwidth. Therefore, realistic solutions should use local storage and inter-node communication sparingly, and account for node crashes and message losses.

In this paper, we investigate the problem of reliable stream diffusion in unreliable and constrained environment from a novel angle. Our approach is probabilistic: with high probability, all consumers will be reached. The key idea of our solution is to diffuse streams according to a global propagation graph. This graph approximates a global tree aiming at the maximum probability of reaching all processes and efficient use of the available bandwidth. The approach is completely decentralized: nodes build propagation trees, which we call *Maximum Probability Trees* (MPTs), autonomously. Several MPTs are dynamically composed to achieve a global graph reaching most (hopefully all) consumer nodes.

2 Model definition and Problem statement

We consider an asynchronous distributed system composed of processes (nodes) that communicate by message passing. Our model is probabilistic in the sense that processes can crash and links can lose messages with a certain probability. More formally, we model the system's topology as a graph $G = (\Pi, \Lambda)$, where $\Pi = \{p_1, p_2, ..., p_n\}$ is a set of n processes and $\Lambda = \{l_1, l_2, ...\} \subseteq \Pi \times \Pi$ is a set of bidirectional communication links.[1] Process crash probabilities and message loss probabilities are modeled as *failure configuration* $C = (P_1, P_2, ..., P_n, L_1, L_2, ..., L_{|\Lambda|})$, where P_i is the probability that process p_i crashes during one computation step and L_j as the probability that link l_j loses a message during one communication step.

Given this probabilistic model, the main question addressed in this paper is the following: *how can we make stream messages reach all consumers with a high probability, in spite of unreliable processes and links, and the limited bandwidth and memory available to each process?*

Formally, the *limited bandwidth constraint* is modeled as $Q = (q_1, q_2, ..., q_n)$, the set of *individual quotas of messages* q_i at disposal of each process p_i to forward one single stream packet. By defining the different aspects of our model formally,

*This research is partly funded by the Swiss National Science Foundation, in the context of Project number 200021-108191.

[1] That is, we have $V(G) = \Pi$ and $E(G) = \Lambda$; we only consider systems with a connected graph topology.

we then can say that the tuple $S = (\Pi, \Lambda, C, Q)$ completely defines the system considered in this paper. In order to take into account the *limited memory constraint*, we further assume that each process has only a partial view of the system representing its neighborhood [2]. Formally, the limited knowledge of process p_i is modeled with *distance* d_i, which is the maximum number of links in the shortest path separating p_i from any other node in its known subgraph. That is, distance d_i implicitly defines the partial knowledge of p_i as scope $s_i = (\Pi_i, \Lambda_i, C_i, Q_i)$. We can now restate more formally the problem we address in this paper: *given its limited scope s_i, how should process p_i use its quota q_i in order to contribute to reach all consumers with a high probability?*

3 A Gambling Approach

Our solution is based on the four-layer architecture pictured in Figure 1. The top layer executes the *Scalable Streaming Algorithm (SSA)*, which is responsible for breaking the outgoing stream into a sequence of messages on the producer side, and for assembling these messages back into an incoming stream on the consumer side. For multicasting and delivering a stream packet respectively on the producer and consumer sides, the SSA layer relies on the *Packet Routing Algorithm (PRA)*, which is responsible for routing stream messages through a *propagation graph* covering the whole system. This propagation graph results from the spontaneous aggregation of various *propagation trees* concurrently computed by intermediate routing processes. Both producers and consumers execute the SSA and PRA layers, while pure routing processes only execute the PRA layer.

The responsibility for building propagation trees is delegated to the *Propagation Tree Algorithm (PTA)*, which in turn relies on the partial view delivered by the *Environment Modeling Layer (EML)*. The latter approximates the environment within distance d_i of each process p_i. Explaining how environment modeling actually works goes beyond the scope of this paper and can be found in [4]. Finally, the *Unreliable Link Layer (ULL)* allows each process p_i to send messages to its direct neighbors in a probabilistically unreliable manner.

[2]This limited view could be quickly approximated with the environment modeling method presented in [4].

3.1 Packet Routing Algorithm

The packet routing solution, presented in Algorithm 1, consists in disseminating stream messages through a *propagation graph* generated in a fully decentralized manner. This propagation graph results from the spontaneous aggregation of several *propagation trees*. Each propagation tree is in turn the result of an incremental building process carried out along the paths from the producer to the consumers. It is important to note however that the propagation graph itself might well not be a tree.

```
 1: uses: PTA, ULL and EML
 2: initialization:
 3:    r ← ...

 4: procedure multicast(m)
 5:    pt ← PTA.incrementPT( ({p_i}, ∅, {P_i}, {q_i}) )
 6:    m⃗ ← optimize(pt)
 7:    propagate(m, pt, p_i, m⃗)

 8: upon ULL.receive(m, p_k, pt, m⃗) do
 9:    if EML.distance(p_k, p_i) ≥ r then
10:       pt ← PTA. incrementPT(pt)
11:       m⃗ ← optimize(pt)
12:       propagate(m, pt, p_i, m⃗)
13:    else
14:       propagate(m, pt, p_k, m⃗)
15:    if p_i is interested in m then
16:       SSA.deliver(m)

17: procedure propagate(m, pt, p_k, m⃗)
18:    for all p_j such that link (p_i, p_j) ∈ E(pt) do
19:       repeat m⃗[j] times :
20:          ULL.send(m, p_k, pt, m⃗) to p_j
```

Algorithm 1: PRA at p_i

On the producer. The routing process starts with producer p_i calling the *multicast*() primitive (line 4). As a first step, p_i asks the PTA layer to build a first propagation tree pt, using the *incrementPT*() primitive (line 5). This primitive is responsible for incrementing the propagation tree passed as argument, using the scope of the process executing it (here p_i). Since p_i is the producer, the initial propagation tree passed as argument is simply composed of p_i and its associated information (failure probability P_i and quota q_i). As discussed in Section 3.2, the returned propagation tree pt maximizes the probability to reach everybody in scope s_i, based on available quotas. Process p_i then calls the *optimize*() primitive, passing it pt (line 6). This primitive is discussed in details in Section 3.3. At this point, all we need to know is that it returns a propagation vector $\vec{m}$ indicating, for each link in pt, the number of messages that should be sent through that link in order

to maximize the probability to reach everybody in scope s_i. Finally, p_i calls the *propagate*() primitive (line 7), which simply follows the forwarding instructions computed by *optimize*(). That is, it sends stream message m, together with some additional information, to the direct neighbors of p_i. As we shall see below, this additional information is used throughout the routing process to build up the propagation graph.

On the consumer. When a consumer p_i receives message m, together with the aforementioned information (line 8), it has first to decide whether to increment pt before further propagating m (lines 10 to 12), or to simply follow the propagation tree pt it just received (line 14). The propagation tree pt should be incremented if and only if the distance that separates p_i from p_k, the process that last incremented pt, is greater or equal to r, the *incrementation rate*. Intuitively, r defines how often a propagation tree should be incremented as it travels through the propagation graph. The latter then spontaneously results from the concurrent and uncoordinated incrementations of propagation trees finding their ways to the consumers. Finally, process p_i delivers message m to the SSA layer only if it is interested in it (lines 15 and 16). If this is not the case, process p_i is merely a router node.

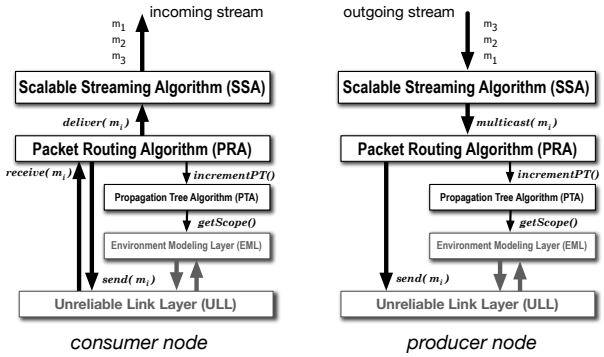


Figure 1: Scalable Streaming Architecture

3.2 Propagation Tree Algorithm

The solution to increment propagation trees is encapsulated in the *incrementPT*() primitive, presented in Algorithm 2. This primitive takes a propagation tree pt as argument and increments it if needed, i.e., if something changed in the scope of p_i or if pt is different from the propagation tree that was last incremented (line 6).[3] To get an up-to-date view of its scope, p_i calls the *getScope*() primitive provided by EML (line 5). In order to increment pt, process p_i then builds a *local* propagation tree lpt_i, based on its most up-to-date scope s_i (lines 7 to 10), and finally merges lpt_i with pt (line 11).

To build local tree lpt_i, process p_i first builds a *Maximum Probability Tree (MPT)*, using the *mpt*() primitive (line 9) detailed in Section 3.3. Note that primitive *mpt*() increments pt as a whole tree, with the best branches in scope s_i, even if some of these branches are not descendant from p_i. Whereas Algorithm 2 is only interested in the subtree rooted at p_i (line 10). This subtree is precisely the local tree lpt_i. Process p_i has indeed no way to inform processes that are not its descendants about its incrementing decisions, and has no guarantee that concurrent trees will be incremented coherently with respect to each others. This materializes the gambling risk taken during the construction of the propagation graph.

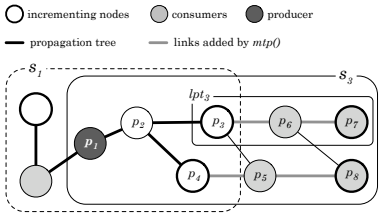
```
1:  uses: EML
2:  initialization:
3:    lpt_i, pt_i, s_i ← ∅, ∅, ∅

4:  function incrementPT(pt)
5:    s ← EML.getScope( )
6:    if pt_i ≠ pt ∨ s_i ≠ s then
7:      pt_i ← pt
8:      s_i ← s
9:      mpt ← mpt(s_i, pt_i)
10:     lpt_i ← subtree of mpt with p_i as root
11:   return pt ∪ lpt_i
```

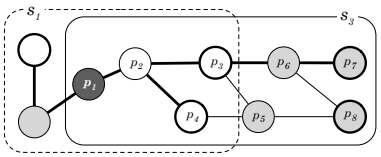
Algorithm 2: PTA executed by p_i

Execution example. Figure 2 illustrates the propagation tree incrementation process. In this scenario, the distance defining the scope and the incrementation rate r are the same for all processes and equal to 2. Process p_1, the producer, builds a first *propagation tree* pt_1 covering its scope s_1; this tree is pictured in Figure 2 (a) using black links. All nodes in pt_1 that are at a distance $r = 2$ from p_1 have to increment pt_1 when they receive it. Process p_3 being such a node, it calls the *mpt*() function, passing it pt_1 and its scope s_3. This function adds the gray links pictured in Figure 2 (a) to pt_1 and returns the resulting Maximum Probability Tree (MPT); this MPT contains the local propagation tree rooted at p_3, i.e., lpt_3. The latter is then extracted from the MPT, merged with the initial propagation tree pt_1 and returned. Figure 2 (b) pictures the new propagation tree resulting from the above incrementation process.

[3]The conditional nature of this incrementation is motivated by performance concerns: during stable periods of the system, propagation trees remain unchanged, cutting down the processing load of incrementing nodes.



(a) before the return statement



(b) returned propagation tree

Figure 2: Propagation tree incrementation

3.3 Maximum Probability Tree

The concept of *Maximum Probability Tree (MPT)* is at the heart of our approach. Intuitively, an MPT maximizes the probability to reach all processes within a scope by using the quotas of these processes. Before describing how the *mpt*() function given in Algorithm 3 builds up an MPT, we first need to introduce the notions of *reachability probability* and *reachability function*borrowed from [4].

Reachability probability. The *reachability function*, noted $R()$, computes the probability to reach all processes in some propagation tree T, given a vector $\vec{m}$ defining the number of messages that should transit through each link of T. We then define the probability returned by $R()$ as T's *reachability probability*. Equation 1 below proposes a simplified version of the reachability function borrowed from [4]. This version assumes that only links can fail by losing messages, whereas processes are assumed to be reliable.[4]

$$R(T,\vec{m}) \;=\; \prod_{j=1}^{|\vec{m}|} 1 - L_j^{m[j]} \quad with \quad L_j \in C(T) \qquad (1)$$

Using $R()$, we then define the $maxR()$ function presented in Algorithm 3 (lines 8 to 10), which returns the maximum reachability probability for T. To achieve this, $maxR()$ first calls the *optimize*() function in order to obtain a vector $\vec{m}$ that distributes the quotas at disposal of processes in T in order to maximize the latter *reachability probability*. It then passes this vector, together with T, to $R()$ and returns the corresponding reachability probability.

[4]Note that this simplification causes no loss of generality; see [4] for details.

```
1:  function mpt(S, T)
2:    while V(S) ⊈ V(T) do
3:      O ← {l_{j,k} | l_{j,k} ∈ E(S) ∧ p_j ∈ V(T) ∧ p_k ∈
        V(S) − V(T)}
4:      let l_{u,v} ∈ O such that ∀l_{r,s} ∈ O :
5:      maxR(T ∪ l_{u,v}) ≥ maxR(T ∪ l_{r,s})
6:        T ← T ∪ l_{u,v}
7:    return T

8:  function maxR(T)
9:    m⃗ ← optimize(T)
10:   return R(T, m⃗)

11: function optimize(T)
12:   let m⃗ : ∀l_j ∈ E(T), m⃗[j] is the number of
      messages to be sent through link l_j
13:   m⃗ ← (1, 1, ··· , 1)
14:   for all p_s ∈ V(T) do
15:     let Λ_s ⊂ E(T) : l_k ∈ Λ_s ⇒ (p_s, p_k) ∈ E(T)
16:     if | Λ_s | > q_s then
17:       return (0, 0, ··· , 0)
18:     while Σ_{l_k∈Λ_s} m⃗[k] < q_s do
19:       let m⃗_u : (l_u ∈ Λ_s) ∧
20:           (∀_{t≠u} m⃗_u[t] = m⃗[t]) ∧
21:           (m⃗_u[u] = m⃗[u] + 1) ∧
22:           (R(T, m⃗_u) − R(T, m⃗) is max)
23:       m⃗ ← m⃗_u
24:   return m⃗
```

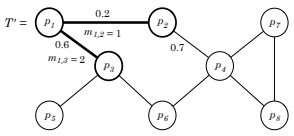
Algorithm 3: MPT | Building Process

The *optimize*() function iterates through each process p_s in T and divides individual quota q_s[5] in a way that maximizes the probability to reach direct children of p_s (line 14 to 23). For this, function *optimize*() allots messages of q_s one by one, until all messages have been allocated (line 18 to 23). That is, in each iteration step it chooses the outgoing link l_u from p_s that maximizes the gain in probability to reach all p_s's children in T, when sending one more message through l_u (line 22). When all individual quotas have been allocated, *optimize*() returns a vector $\vec{m}$ that provides the maximum reachability probability when associated with T.

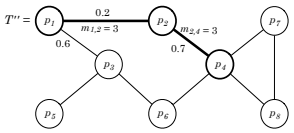
MPT building process. The MPT building process carried out by *mpt*(), given a scope S and an initial propagation tree T. This function simply iterates until all processes in S but not in T have been linked to T (line 2 to 6). In each iteration step, the *mpt*() function adds the link that produces a new tree exhibiting the maximum reachability probability (line 5).

[5]When q_s is sufficient to reach all p_s's children in T.

Execution example. Figure 3 illustrates the MPT building process on a simple example. In this example, the initial tree T is composed of only process p_1. The pictured graph represents s_1. During the first iteration, the algorithm simply chooses the most reliable outgoing link, i.e., link $l_{1,2}$ with failure probability $L_{1,2} = 0.2$. At this point, it means that the entirety of p_1's quota has been allocated to reach p_2. In this example, the quota is identical for all processes and equal to 3, i.e., $\forall p_i : q_i = 3$. At the beginning of the second step, the algorithm faces two alternatives: either adding link $l_{1,3}$ and splitting the quota of p_1 between links $l_{1,2}$ and $l_{1,3}$, or adding link $l_{2,4}$ and using the entirety of q_2, the quota of p_2, to reach p_4. These two alternatives are pictured in Figure 3 as trees T' and T'' respectively. Based on the result of function $maxR()$,



$$maxR(T') = (1 - L_{1,2}^{m_{1,2}}) \times (1 - L_{1,3}^{m_{1,3}}) = 0.512$$



$$maxR(T'') = (1 - L_{1,2}^{m_{1,2}}) \times (1 - L_{2,4}^{m_{2,4}}) = 0.651$$

Figure 3: Alternative trees of the second iteration

the algorithm chooses to keep T'', since it is the tree that offers the maximum probability to reach everybody. Note however that this decision implies adding link $l_{2,4}$ rather than link $l_{1,2}$, although the latter is more reliable.

3.4 Fine tuning via scope d_i and incrementation rate r

Then, since by definition we have that $r \leq d_i$, the scope of p_i, we can say that r and d_i determine the tradeoff between memory and computation load. We already know that as d_i increases, so does the memory required by p_i to store its scope s_i. On the other hand, as r decreases, the computation load increases because more processes turn out to be incrementing nodes. However, a big d_i also allows for a big r, meaning that we are trading off computation load for memory.

4 Final remarks

Several application-level multicast systems based on a tree have been proposed in the literature. Some of them define a multicast tree that aims at optimizing the bandwidth use [3, 5]. Others, also deal with scalability by limiting the knowledge each process has about the system [2, 6]. Yet, other systems aim at increasing robustness with respect to packet loss [1, 7]. Our approach differs from these systems in that it targets the three goals simultaneously. Our propagation structure is build collaboratively by distributed processes using their respective partial views of system. Reliability is accounted for by each process when building its local tree. Finally, bandwidth constraints are considered when defining how to forward packets along the propagation graph. Our strategy shares some design goals with broadcast protocols such as [4]. Both rely on the definition of a criteria for selecting the multicasting graph. In our strategy, however, we strive to both decrease packet loss and balance the forwarding load.

Differently from more traditional approaches, we resort to a "gambling approach," which deliberately penalizes a few consumers in rare cases, in order to benefit most consumers in common cases. We believe that this main open up new directions for future work on large-scale data dissemination protocols. Our current work is investigating alternative gambling algorithms.

References

[1] J. G. Apostolopoulos and S. J. Wee. Unbalanced multiple description video communication using path diversity. In *IEEE International Conference on Image Processing*, 2001.

[2] M. Castro, P. Druschel, A.M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *Proceedings of ACM SOP*, 2003.

[3] Y. Chu, S. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of ACM Sigmetrics*, 2000.

[4] B. Garbinato, F. Pedone, and R. Schmidt. An adaptive algorithm for efficient message diffusion in unreliable environments. In *Proceedings of IEEE DSN*, 2004.

[5] J. Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and Jr James W. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proceedings of OSDI*, 2000.

[6] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using random subsets to build scalable network services. In *Proceedings of USITS*, 2003.

[7] T. Nguyen and A. Zakhor. Distributed video streaming with forward error correction. In *Packet Video Workshop*, 2002.

Removing Probabilities to Improve Efficiency in Broadcast Algorithms*†

Hugo Miranda
University of Lisbon

Simone Leggio
University of Helsinki

Luís Rodrigues
University of Lisbon

Kimmo Raatikainen
University of Helsinki

Abstract

The broadcast of a message in Mobile Ad Hoc Networks requires its retransmission by multiple devices, consuming both bandwidth and power. In general, broadcast algorithms limit the number of retransmissions but randomly select the nodes that retransmit. This may adversely affect their performance. This paper presents an alternative mechanism for node selection in broadcast algorithms. Evaluation results show that our mechanism can improve both the cost and the coverage of broadcast operations.

1. Introduction

Due to the decentralised nature of Mobile Ad Hoc Networks (MANETs), many services, like route discovery [6, 9], reputation systems [8] or code propagation for sensors [7], require the delivery of some messages to every node. This operation is commonly referred as a broadcast. In some cases, the underlying infrastructure may provide tools to efficiently broadcast messages. This is the case, for example, of spanning trees provided by multicast routing protocols for ad hoc networks. This paper will focus on networks where these tools are not available.

The most common implementation of broadcast is by flooding the network. In flooding, all nodes retransmit a broadcast message after receiving it for the first time. Flooding creates a large number of redundant transmissions. Many nodes receive multiple copies of the message, each transmitted by a different node. Therefore, it wastes a non-negligible amount of bandwidth and power. Independently of the contribution of each retransmission, it consumes resources at the sender-side. Furthermore, receivers also spend a non-negligible amount of energy at the reception [3] and CPU to decide if the message should be retransmitted.

Although the redundant reception of messages cannot be completely avoided, not all participants should be required to retransmit. The minimal number of nodes required to retransmit a broadcast message depends on factors outside the control of any broadcast algorithm, like the transmission range of the devices, the location of the source, the size of the region covered by the nodes or their geographical distribution. The role of broadcast algorithms is to devise a subset of nodes to retransmit that simultaneously: $i)$ is minimal and $ii)$ provides the largest coverage, measured by the proportion of nodes that receive the message. This paper describes a broadcast algorithm that uses a novel scheme for node selection based on the received signal strength indication (RSSI) of the first retransmissions heard by each node. Evaluation shows that in comparison with algorithms that perform a random selection of the nodes, our algorithm either requires a smaller number of nodes to retransmit or achieves a bigger coverage.

The paper is organised as follows. Section 2 describes previous work and shows the motivation for the development of an alternative algorithm. Our algorithm is described in Sec. 3. The results of simulations are presented in Sec. 4. Finally, Sec. 5 summarises the results described in the paper and highlights some future work.

2. Related Work

Reducing the number of nodes required to retransmit a broadcast message in a Mobile Ad Hoc Network (MANET) is not a new subject. In the majority of the algorithms, after receiving a message for the first time, nodes wait for a small period of time, hereafter named the "hold period". The hold period is used by nodes to collect information about the propagation of the message. When the timer expires, some function decides if the node should retransmit, based on the information collected.

The most simple implementation of this generic algorithm is GOSSIP1(p) [4]. In this algorithm, nodes do not collect any additional information. Instead, the decision to retransmit is solely dictated by a probability $p, p < 1$. A

*The work described in this paper was partially supported by the ESF under the MINEMA project and by Fundação para a Ciência e Tecnologia (FCT) and FEDER under project Probabilistically-Structured Overlay Networks (P-SON), POSC/EIA/60941/2004.

†An extented version of this paper appeared in the Procs. of the 17th Int'l Symp. on Personal, Indoor and Mobile Radio Communications (PIMRC'06)

fundamental limitation of GOSSIP1(p) is that the selection of an adequate value for p depends of the node's density: in regions with a small number of neighbours, message propagation can only be achieved if p is high. On the other hand, a high value of p will result in an excessive number of retransmissions when the number of nodes in some neighbourhood is also high.

To address this problem, different variations of GOSSIP1(p) have been proposed. In some, all nodes retransmit at the early stages of the message dissemination while in others, nodes increase their probability of retransmission when the number of neighbours is small [4, 2]. We name all these variations as "probabilistic algorithms".

The coverage of a transmission is dictated by the transmission power and is independent of the number of receivers. In "counter-based algorithms", the primary decision criterion is the number of retransmissions heard. That is, nodes use the hold period for counting the retransmissions. The function then decides to retransmit if the number of retransmissions listened was not sufficient to ensure the successful propagation of the message. In a popular combination of the features of "probabilistic" and "counter-based" algorithms [4, 2], the first decision to retransmit is taken using a probability p. Nodes that decided not to retransmit enter a second hold period. When this second timer expires, nodes retransmit if a sufficient number of retransmissions has not been heard.

Section 4 compares our proposal with two counter-based algorithms, which, for self-containment, are briefly described here. In the "counter-based scheme" [10], after receiving the first copy of a message, nodes randomly select the duration of their hold period, during which they count the number of retransmissions they listen. Nodes decide to retransmit if the number of retransmissions is below some predefined threshold n. A timer for a random delay is also set in the "Hop Count-Aided Broadcasting" (HCAB) algorithm [5] when the first copy of the message is received. In HCAB, messages carry an hop count field (HC), incremented at every retransmission. Nodes decide to retransmit if no retransmission with an HC field higher than the initial was received.

If all nodes transmit with equal power, a retransmission can increase between 0% and 61% the space covered by the previous transmission [10]. The gain increases with the distance between the two transmitting nodes. Although there are some variations, all the algorithms surveyed use randomisation to select the nodes that will perform the retransmission. The random selection may be explicit, like in the "probabilistic algorithms", or implicit, as in "counter-based algorithms". In these algorithms, after receiving a message, each node chooses a random delay, and decides to retransmit at the end of the delay if some criteria has not been satisfied by previous retransmissions. Therefore, a node that randomly selected a smaller delay is more likely to be required to retransmit.

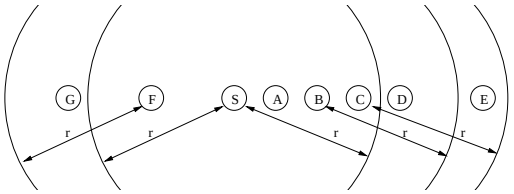


Figure 1. Deployment and transmission range of some nodes

An important factor to improve the performance of broadcast algorithms for MANETs and which has been previously neglected in the literature is the location of the nodes performing a retransmission with respect to the source of the previous transmissions. We provide a simple case study supporting this claim.

Figure 1 represents a region of a MANET with a source S of a broadcast message and its neighbours. It is assumed that all nodes transmit with the same power and are capable of receiving a message if the signal strength at the receiver is above some minimum threshold. In the figure it is assumed that the transmission range of all nodes is r. The transmission range of nodes S, B, C and F is represented by circles.

In this example, retransmissions should be ideally performed by nodes C and F. This subset provides full coverage and presents a minimal number of retransmissions: the message is delivered to nodes A, B, C and F by the first transmission and nodes C and F could retransmit to deliver it respectively to D and E, and G. We emphasise that, in runtime, nodes do not have access to the information required for following the same rationale presented above.

Applying the algorithms surveyed in the related work to the scenario depicted in Fig. 1 shows that for all algorithms the random selection of the nodes may result in either additional retransmissions or incomplete coverage. For example, note that a retransmission performed by node B will not deliver the message to node E. This can only be achieved with an additional retransmission, to be performed by node D. Also as an example, we note that node E would not receive the message in the counter-based scheme if nodes are configured with a threshold of two and nodes A and B are the first to retransmit.

3. A Power-Aware Broadcasting Algorithm

In this section, we propose a novel algorithm to reduce the resources consumed by the nodes and the bandwidth required by broadcasts in MANETs. This is a challenging

problem because we want to minimise the signalling overhead and we do not want to enforce the use of special hardware (e.g. nodes are not required to use a GPS receiver to become aware of their location). Our algorithm only assumes that nodes are able to retrieve the power with which each message is received. The algorithm, named Power-Aware Message Propagation Algorithm (Pampa) is distinguished from the previous proposals by removing the randomness associated with the decision on the nodes that will retransmit a message.

3.1. Pampa

The key idea of Pampa is to run a fully distributed algorithm that makes nodes more distant to the source to retransmit first, instead of relying on a random selection. In an ideal environment, and independently of the node's distribution, this would ensure that each retransmission would be providing the highest additional coverage possible, what would be achieved by the other algorithms only in the fraction of the cases where the more distant node is randomly selected for retransmission.

In Pampa, when receiving a message for the first time, nodes store the message and set a timer for a delay d, given by a function *delay* to be addressed later. During this period, the node counts the number of retransmissions listened. The message is transmitted if, when the timer expires, the node did not listen to a sufficient number of retransmissions.

Central to Pampa is a function *delay* which gets the Received Signal Strength Indication (RSSI) of a transmission and outputs a delay. This function is expected to map an increasing distance to the source (corresponding to a smaller RSSI) in a smaller return value. Because the RSSI will be different for each node, the function *delay* will return a different value for each node receiving the same transmission. Implicitly, the function orders the nodes according to the distance to the source, with nodes more distant to the source expiring their timers first. It should be noted that the function is fully distributed: the algorithm is triggered exclusively by the transmission of the broadcast message and it does not require any coordination between the nodes. Like in the "counter-based scheme" [10], the algorithm prevents excessive redundancy by having nodes to count the number of retransmissions listened. However, Pampa bias the delay such that the nodes refraining from transmitting are usually those that are closer to the source.

***Delay* Assignment.** The selection of a good *delay* function is key to the performance of Pampa. We estimate that a *delay* function that varies linearly with the distance to the source would provide the best results. However, such a function would require complex computations unsuitable to be performed by mobile devices for each received message.

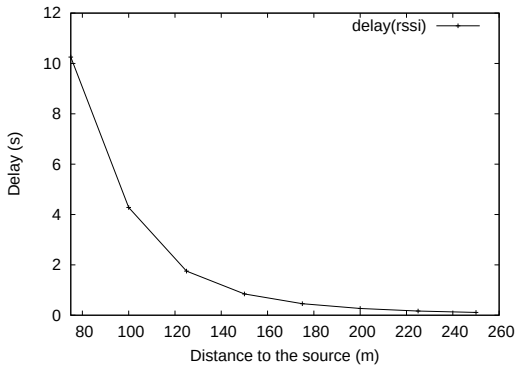


Figure 2. Function *delay*

In our tests, we defined a simpler *delay* function that multiply the RSSI by a constant k to return the number of seconds that the node should wait before retransmitting. The most adequate value of k is likely to depend of the execution scenario. For our simulation environment (Two Ray Ground propagation model as defined in the *ns–2* network simulator version 2.28) we have found 300×10^6 to be an adequate value for obtaining distinct wait times for nodes close to each other. The behaviour of the function *delay*$(\text{rssi}) = 300 \times 10^6 \times \text{rssi}$ is presented in Fig. 2.

As expected, the function follows the logarithmic decay of the reception power of a message. For short distances, the function returns excessively large delay values. However, nodes at these distances from the source have a large probability of not being required to retransmit. A careful implementation of the algorithm can free the resources consumed by the messages on hold as soon as the threshold number of retransmissions is heard.

3.2. Comparison with Related Work

In Pampa, the instant at which each node forwards a message is locally determined from its distance to the sender. In the absence of abnormal effects on the signal propagation, Pampa assures that the first nodes to perform a retransmission are those that provide the higher possible additional coverage. In the example presented in Fig. 1, node C would be the first to retransmit, delivering the message to both nodes D and E. Although slightly later, node F would also be required to retransmit and therefore, guarantee the coverage of node G. In this example, Pampa requires the minimal three transmissions for delivering the message to every node. In addition, node E is more likely to receive the message given that the first retransmission is performed by the node more distant from the source.

4. Evaluation

We have implemented the "counter-based scheme" [10], HCAB [5] and Pampa algorithms in the *ns–2* network simulator v. 2.28. For the "counter-based" and Pampa, we tested different thresholds for the number of times that the same message is received after which a retransmission is discarded. This threshold is shown as the number following the name of each algorithm in the captions of the figures.

Each algorithm had some parameters immutable for all simulations. The maximum random delay used by the "counter-based scheme" and HCAB was set to 0.75s. Pampa multiplies the RSSI by 300×10^6.

All simulations are run with 100 nodes. Different node densities have been experimented by changing the size of the simulated space. Eight simulated regions were tested from $250\text{m} \times 250\text{m}$ to $2000\text{m} \times 2000\text{m}$, providing ratios between $625\text{m}^2/\text{node}$ and $40000\text{m}^2/\text{node}$. Nodes were configured to emulate a 914MHz Lucent WaveLAN DSSS radio interface running an IEEE802.11 protocol at 2Mb/s. Network cards present a transmission range of 250m using the Two Ray Ground propagation model.

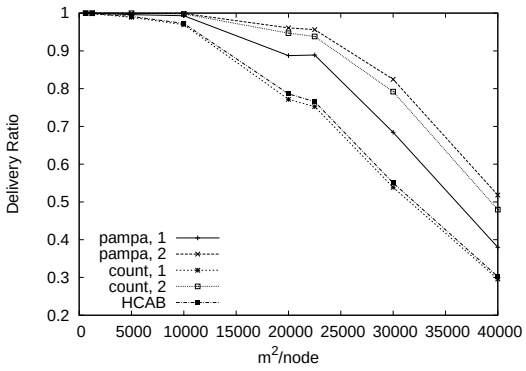
At the beginning of each test, nodes are uniformly deployed over the simulated region. In one set of tests, nodes do not move for the entire duration of the simulation. These tests have been named "Speed 0". In the remaining set, nodes move using the Random Waypoint Movement Model. The minimum and maximum speeds are 9m/s and 11m/s. Nodes never stop. This set was named "Speed 10".

For each simulated region and speed, 100 different tests were defined and experimented with each of the algorithms. Each test combines different traffic sources and movement of the nodes. Traffic in each test is composed of 1000 messages, generated at a pace of one message per second. The source of each message is selected at random. The size of each message is 1000 bytes. Each point in the figures presented below averages the result of the 100 runs.

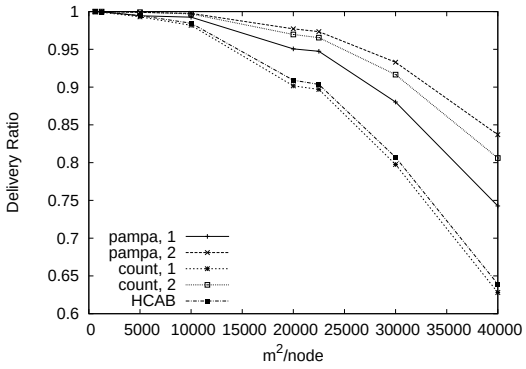
4.1. Coverage

To compare the efficiency of the algorithms we use the average of the proportion of the nodes that receive each message. Figure 3 compares the performance of the "counter-based scheme" and HCAB algorithms with Pampa. A comparison between the two plots of the figure shows that the performance of all algorithms improves with the movement of the nodes. This behaviour is attributed to a reduced number of partitions, which results from the concentration of nodes at the centre of the simulated space, a well-known effect of the random way-point movement model [1].

The figure shows that for high densities, all the algorithms are capable of delivering every message to all nodes.



(a) Speed 0



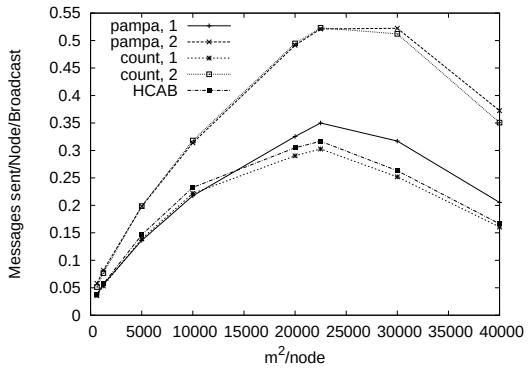
(b) Speed 10

Figure 3. Delivery Ratio

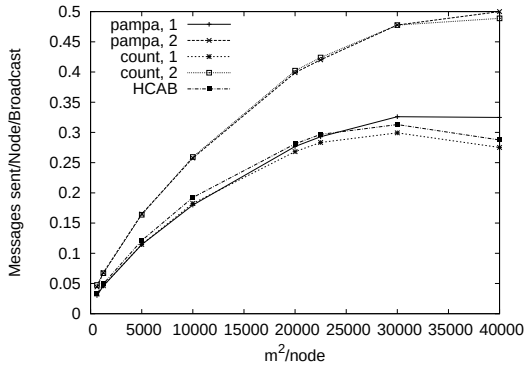
As the area of the simulation increases, so does the average distance between the nodes and the gains provided by Pampa become more clear. This becomes more evident in the cases where the message threshold is lower. For a threshold of one message, the delivery ratio of the "counter-based scheme" begins to decay at a much faster pace than Pampa. We attribute this behaviour to the randomness associated with the node selection in the "counter-based scheme". In Pampa, the nodes forwarding the message have a higher probability of reaching more distant locations. When the simulated space is of $2000\text{m} \times 2000\text{m}$ $(40000\text{m}^2/\text{node})$ network partitions begin to affect message dissemination. The benefits of using Pampa can be more clearly observed in these extreme conditions: for the same thresholds, Pampa always presents a higher delivery ratio. The unique node selection criteria of Pampa helps to have the messages delivered to distant nodes improving its delivery ratio. HCAB presents a delivery ratio comparable to the "counter-based scheme" with threshold one.

4.2. Retransmissions

The proportion of nodes that retransmit each broadcast message is depicted in Fig. 4. For high densities,



(a) Speed 0



(b) Speed 10

Figure 4. Number of Retransmissions

Pampa does not require more retransmissions than the count-based scheme with the same threshold. This confirms that when nodes are closer, the location of the retransmitting nodes loses relevance. For lower densities, Pampa in general requires more retransmissions than the remaining. This is justified by the additional coverage it achieves. It should be noted that in the most advantageous cases, (e.g. $10000\text{m}^2/\text{node}$, Speed 0, threshold 1 and $20000\text{m}^2/\text{node}$, Speed 0, threshold 2), the additional coverage is achieved with a similar number of retransmissions.

5. Conclusions

In Mobile Ad Hoc Networks, broadcasting a message to every node is an operation that consumes a non-negligible amount of resources at all participants. However, broadcast is a basic mechanism often required by protocols at different levels of the network stack. The most simple implementation of broadcast consists in having each node to retransmit each message after receiving it for the first time. This implementation, usually referred as flooding, creates a large redundancy of messages in the network and unnecessarily wastes resources at the participating nodes.

This paper presented a new algorithm that uses information locally available at each node to reduce the redundancy of the broadcast operation. The novelty of the algorithm, named Pampa, is the ranking of the nodes according to their distance to the source. Pampa does not require the exchange of control messages or specialised hardware.

The algorithm was compared with previous proposals and it was shown to improve their performance, particularly in more adverse conditions like sparse networks. In the future, we plan to deploy and evaluate Pampa in real wireless networks to confirm Pampa's usability in more adverse conditions, for example with the influence of the environment on the signal strength perceived by each node.

References

[1] C. Bettstetter, G. Resta, and P. Santi. The node distribution of the random waypoint mobility model for wireless ad hoc networks. *IEEE Trans. on Mobile Computing*, 2(3):257–269, 2003.

[2] V. Drabkin, R. Friedman, G. Kliot, and M. Segal. RAPID: Reliable probabilistic dissemination in wireless ad-hoc networks. Technical Report CS-2006-19, Computer Science Department, Technion - Israel Institute of Technology, 2006.

[3] L. M. Feeney and M. Nilsson. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *Procs. of the 20th Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM 2001)*, pages 1548–1557, 2001.

[4] Z. J. Haas, J. Y. Halpern, and L. Li. Gossip-based ad hoc routing. In *Procs. of the 21st Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM 2002)*, pages 1707–1716, 2002.

[5] Q. Huang, Y. Bai, and L. Chen. Efficient lightweight broadcasting protocols for multi-hop ad hoc networks. In *Procs. of the 17th Annual IEEE Int'l Symp. on Personal, Indoor and Mobile Radio Communications (PIMRC'06)*, 2006.

[6] D. B. Johnson, D. A. Maltz, and J. Broch. *Ad Hoc Networking*, chapter DSR: The Dynamic Source Routing Protocol for Multi-Hop Wireless Ad Hoc Networks, pages 139–172. Addison-Wesley, 2001.

[7] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Procs. of the 1st USENIX/ACM Symp. on Networked Systems Design and Implementation (NSDI 2004)*, 2004.

[8] F. Perich, J. Undercoffer, L. Kagal, A. Joshi, T. Finin, and Y. Yesha. In reputation we believe: query processing in mobile ad-hoc networks. In *Procs. of the 1st Annual Int'l Conf. on Mobile and Ubiquitous Systems: Networking and Services (MOBIQUITOUS 2004)*, pages 326–334, 2004.

[9] C. E. Perkins and E. M. Royer. Ad-hoc on-demand distance vector routing. In *Procs. of the 2nd IEEE Work. on Mobile Computing Systems and Applications*, pages 90–100, 1999.

[10] Y.-C. Tseng, S.-Y. Ni, Y.-S. Chen, and J.-P. Sheu. The broadcast storm problem in a mobile ad hoc network. *Wireless Networks*, 8(2/3):153–167, 2002.

GossipKit: A Framework of Gossip Protocol Family

Shen Lin, François Taïani, Gordon S. Blair
Computing Department
Lancaster University
Lancaster LA1 4YR, UK
(s.lin6, f.taiani, gordon)@comp.lancs.ac.uk

***Abstract*—A large number of gossip protocols have been developed in the last few years to address a wide range of functionalities. So far, however, very few software frameworks have been proposed to ease the development and deployment of these gossip protocols. To address this issue, this paper presents GossipKit, an event-driven framework that provides a generic and extensible architecture for the development of (re)configurable gossip-oriented middleware. GossipKit is based on a generic interaction model for gossip protocols and relies on a fine-grained event mechanism to facilitate configuration and reconfiguration and promote code reuse.**

1. Introduction and Problem Statement

Gossip-based algorithms have recently become extremely popular. The underlying concept of these algorithms is that individual nodes repeatedly exchange data with some randomly selected neighbours, causing information to eventually spread through the system in a "rumour-like" fashion. Gossip-based protocols offer several key advantages over more traditional systems: 1) they provide a scalable approach to communication in very large systems; 2) thanks to the randomised and periodic exchange of information, they offer self-healing capacities and robustness to failures; and 3) they are simple to implement. Because of these benefits, gossip-based protocols have been applied to a wide range of contexts such as peer sampling [9], [17], ad-hoc routing [14], reliable multicast [1], [2], database replication [10], failure detection [11], and data aggregation [12].

Unfortunately, past research has mainly focused on the development and evaluation of new gossip protocols. In particular very few attempts have been made at developing (re)configurable middleware architectures to support gossip-based systems. T-Man [5] and the recent work at Bologna [6] are two of the early gossip-dedicated frameworks that have been proposed in this area. They both rely on a common periodic gossip pattern to support a variety of gossip protocols. Although these frameworks can help develop gossip-based systems to a significant extent, we contend that they only partially address the issues faced by the developers of gossip-based applications. First, the common periodic gossip pattern they rely on only captures the features of proactive gossip protocols. As such, it does not support reactive gossip algorithms. Second, these frameworks tend to be monolithic and as such do not provide a flexible architecture that is easily extensible. Third, these frameworks do not support runtime reconfiguration.

This paper introduces GossipKit, a fine-grained event-driven framework we have developed to ease the development of (re)configurable gossip-based systems that operate in heterogeneous networks such as IP-based networks and mobile ad-hoc networks. The goal of GossipKit is to provide a middleware toolkit that helps programmers and system designers develop, deploy, and maintain distributed gossip-oriented applications. GossipKit has a component-based architecture that promotes code reuse and facilitates the development of new protocols. By enforcing the same structure across multiple and possibly co-existing protocols, GossipKit simplifies the deployment and configuration of multiple protocol instances. Finally, at runtime, GossipKit allows multiple protocol instances to be dynamically loaded, operate concurrently, and collaborate with each other in order to achieve more sophisticated operations.

The contributions of this paper are threefold. First, we identify a generic and modular interaction pattern that most gossip protocols follow. Second, we propose an event-driven architecture based on this pattern that can be easily extended to cover a wider range of gossip protocols. Third, we briefly evaluate how our event-driven architecture provides a fine-grained mechanism to compose gossip protocols within the GossipKit framework.

The remainder of the paper is organised as follows. Section 2 discusses related work. Section 3 presents a study of existing gossip protocols and explains how this study informed the key design choices of GossipKit. Section 4 gives an overview of GossipKit's architecture. Section 5 describes the current implementation of the GossipKit framework, while an early evaluation is provided in Section 6. Finally, Section 7 concludes the paper and points out future work.

2. Related Work

Two categories of communication frameworks have been proposed to support gossip protocols: Gossip Frameworks, which explicitly and directly support gossip-based systems, and Event-driven communication systems, which tend to be more generic and more flexible. In this section we analyse the strengths and weaknesses of both of them from the viewpoint of gossip protocol development.

Gossip frameworks are specifically designed to support gossip protocols. Typical examples of such framework are T-Man [5] and the recent work on this topic at Bologna [6]. These two frameworks assume that most gossip protocols adopt a common proactive gossip pattern. In this gossip pattern, a peer P maintains two threads. One is an active thread, which periodically pushes the local state S_P to a randomly selected peer Q or pulls for Q's local state S_Q. The other is passive, which listens to push or pull messages from other peers. If the received message is pull, P replies with S_P; if the received message is push, P updates S_P with the state in the message.

To develop a new gossip protocol within this common proactive gossip pattern, one only needs to define a state S, a method of peer selection, an interaction style (i.e. pull, push or pull-push), and a state update method. Many proactive gossip protocols such as peer sampling service, data aggregation, and topologic maintenance have been implemented in such Gossip frameworks.

However, the monolithic design of these Gossip frameworks makes them inadapted to protocols that use a reactive gossip pattern (e.g. SCAMP [9]) or those implementing sophisticated optimisations such as feedback based dissemination decision [13] and premature gossip death prevention [14]. Furthermore, these Gossip frameworks neither support reconfiguration nor concurrent operation of multiple gossip protocols at runtime.

Event-driven communication systems aim to provide a flexible composition model based on event-driven execution. They are

developed to support general-purpose communication and but not specifically for gossip protocols. Examples of such communication systems are Ensemble [3], Cactus [4] and their predecessors Isis [7] and Coyote [8]. In these environments, a configurable service (e.g. a Configurable Transport Protocol) is viewed as a composition of several functional properties (e.g. reliability, flow control, and ordering). Each functional property is then implemented as a micro-protocol that consists of a collection of event handlers. Multiple event handlers may be bound to a particular event and when this event occurs, all bounded event handlers are executed.

Event-driven communication systems offer a number of benefits for developing gossip protocols. First, individual micro-protocols can be reused to construct families of related gossip protocols (implemented as services) for different applications instead of implementing a new service from scratch for each protocol. Second, reconfigurability can be achieved by dynamically loading micro-protocols and rebinding event handlers to appropriate events. Finally, the use of event handlers present a fine-grained decomposition of protocols.

However, event-driven frameworks are known to be notoriously difficult to program and configure as argued in [16]. In large part, this is because these frameworks do not by themselves include any domain-specific features (e.g. interaction patterns and common structure) for individual protocol types.

In order to address the major shortcomings discussed in this section, GossipKit adopts *a hybrid approach that combines domain-specific abstraction and the strengths of event-driven architecture*. The remaining sections of this paper present its design and prototype implementation.

3. GossipKit's Key Design Choices

To design GossipKit, we first investigated a number of existing gossip-based protocols and identified similarities and differences amongst them. In this section, we report on the results of this study and present the key design choices we made for GossipKit based on these results. More precisely we look at three aspects of gossip protocols: Section 3.1 explains the reason of using domain-specific interfaces for different types of gossip protocol to interface with external applications. Section 3.2 presents the common interaction pattern of gossip protocols that we have observed, and finally Section 3.3 argues the benefits of adopting an event-driven architecture for our gossip protocol framework.

3.1 Application-dependent Interfaces

As mentioned previously, gossip-based solutions have been proposed for a wide range of distributed applications. Different types of gossip protocol interact with the external world distinctively. For instance, a gossip-based routing protocol must provide an interface for external application systems to trigger the route request that will be gossiped, whilst a gossip protocol for peer sampling service needs to provide access to the collected peer samples. From our experience and analysis, it is unlikely to identify a common generic interface that can separate gossip protocols from the applications that utilise them. Instead we proposed to identify a *set* of generic but domain-specific interfaces that can each support a family of gossip protocols in a particular application domain. In order to do so, we have classified gossip protocols into categories in accordance with their functionality. This has enabled us to identify a common interface for gossip protocols within each category that can be used to interact with their external applications. Through domain-specific common interfaces, external applications can access various types of gossip protocols that operate in a single framework. Section 4.1 will describe the mapping between these domain-specific interfaces and control logic in detail.

3.2 Common Interaction Pattern

Although different types of gossip protocols provide divergent interfaces to external applications, we have found that, internally, they all follow the same interaction pattern. This common interaction pattern can be captured using a modular approach and combines the proactive gossip pattern that has been identified in [5] and [6], with the reactive gossip patterns observed on gossip protocols such as [9] and [14]. This common interaction model is shown in Fig. 1. In this figure, the modules involved in the interaction are presented as boxes, and interactions between modules as arrowed lines. The direction of the arrows indicates which module initiates the interaction, and the labels show in which sequence these interactions take place.

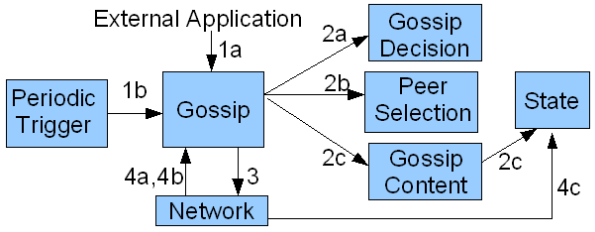


Fig. 1. Common Interaction Model

Initially, a gossip dissemination can either be raised periodically (e.g. a periodic pull or push of gossip message), or upon a receipt of an external request (e.g. an ad-hoc routing protocol requesting a reactive gossip protocol such as [14] to gossip a route request). These two interactions are represented as 1a and 1b in Fig. 1, respectively.

The second phase prepares the gossip action. Some gossip protocols may use various policies to decide whether to gossip at the current situation (2a). For instance, a reactive gossip protocol may decide not to gossip the same message twice or forward the message with a given probability. If a decision is made to forward the gossip message, the protocol instance must then select the peers it wishes to gossip with (2b). In addition, many gossip protocols will need to decide which content is to be gossiped (2c). In particular, a proactive gossip protocol typically requires to retrieve the gossip content from its local state if it needs to send periodically its state (push-style gossip) or reply to a request of its state (pull-style gossip). An example of gossip content could be the temperature sensed by each peer.

The third phase is gossip dissemination. It utilises the underlying network to send gossip messages to the selected peers (3).

Finally, on receipt of a gossip message from the network, a gossip protocol may react in three different ways, depending on the type of the received message: 1) it might forward the message to peers that it knows (4a) and this may involve the interactions in phase 2 (2a, 2b and 2c); 2) it might respond with its own state (4b) and similarly this can involve the interactions in phase 2; and 3) it might extract the state contained in the message and merges with its own state (4c).

Note that this overall interaction model can be invoked recursively — each module presented in Fig. 1 can itself be implemented as a gossip protocol that follows the interaction model. For instance, the Peer Selection module can be a gossip-based peer sampling service protocol.

In practice, various gossip protocols may be composed from completely different implementations of modules in Fig. 1, and these

coarse-grained modules can hardly be reused. In order to enable optimal reuse, the framework allows each module to be composed from a variety of finer-grained micro-modules.

More precisely, we have noticed that five modules (Gossip, Peer Selection, Gossip Decision, Gossip Content, and State) in Fig. 1 can often be decomposed into finer-grained and reusable *micro-modules*. Each individual micro-module implements a distinct algorithm, and different combinations of these micro-modules can form modules with more sophisticated behaviours. Consider the example presented in Fig. 2. This example shows three gossip-decision policies used in a gossip-based ad-hoc routing protocols ($Gossip1(p)$, $Gossip2(p,k)$, and $Gossip3(p,k,p_1,n)$) [14]. Instead of being implemented as independent coarse-grained decision modules, these three decision strategies can reuse the same three fine-grained micro-modules.

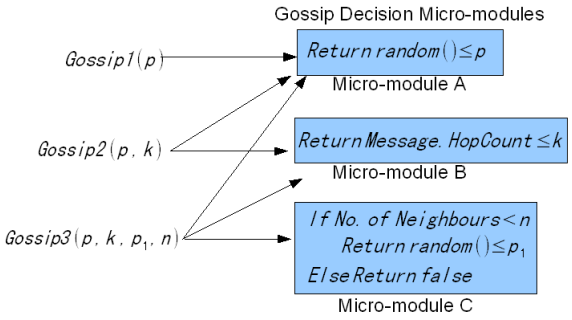


Fig. 2. Various Gossip Decision modules realised by different composition of micro-modules

More precisely, $Gossip1$, $Gossip2$, and $Gossip3$ differ by how they decide whether to forward the received routing request message (i.e. they require different versions of the Gossip Decision module): $Gossip1$ forwards the message with probability p; $Gossip2$ is the same as $Gossip1$ except that it forwards the message with probability 1 in the fist k hops; and $Gossip3$ is the same as $Gossip2$ except that it forwards message with probability $p_1 > p$ if it has less than n neighbouring peers.

These three different gossip decision strategies can be implemented by different combination of the three fine-grained Gossip Decision micro-modules shown on Fig. 2. $Gossip1$ can directly use micro-module A as its Gossip Decision module; $Gossip2$'s Gossip Decision module can be viewed as a composition of micro-module A and B by evaluating the return values of these two micro-module using boolean operation OR to obtain the decision for forwarding the message; and $Gossip3$'s Gossip Decision module can be composed from micro-module A, B, and C in the same way as $Gossip2$ does.

3.3 Event-driven Architecture

The common interaction pattern of gossip protocols we have just presented serves as the basis for our architecture design. Based on the study of gossip protocols, it is clear that a generic system architecture should satisfy the following two criteria.

First, our architecture should allow micro-modules to be easily configured and implement the various modules found in our common interaction pattern. This requirement can be fulfilled using event-driven frameworks such as Ensemble and Cactus. In these frameworks, micro-modules (e.g. Gossip Decision micro-modules shown in Fig. 2) can be viewed as event handlers that are bound to certain events, and the arbitrary composition of micro-modules can be simplified to uniform event-bindings. For instance, to compose a Gossip Decision module, Gossip Decision micro-modules can be bound to events raised by Gossip modules (Gossip Decision module is invoked by Gossip module as shown in Fig. 1). The Gossip module then evaluates the return values of the invoked Gossip Decision micro-modules using boolean operation OR, so as to obtain the decision for forwarding the message. Furthermore, one can simply change the event-bindings to obtain a different composition of micro-modules.

Second, the architecture should be easily extensible to support new gossip protocols on the basis of the common interaction pattern shown in Fig. 1. This is because our interaction pattern is based on the study of typical and representative gossip protocols. It does not cover however all existing gossip algorithms. New gossip protocols may require extra modules and interactions beyond the common interaction pattern. Therefore, it is important that the system architecture allows new modules and interactions to be added onto the pattern. This issue can be addressed by using event-driven systems. In an event-driven system, interactions between event handlers can be achieved through passing events and hence, minimises the explicit references between modules as argued in [8]. As a consequence, our framework can be easily extended by plugging in new micro-modules (i.e. event handlers) and reconfiguring the event binding to support new interaction patterns.

From the above analysis, we have therefore chosen an event-driven architecture for our framework in order to easily configure the composition of micro-modules and to improve extensibility of the common interaction model in Fig. 1. The details of the resulting architecture are presented in Section 4.

4. GossipKit's Architectural Overview

Our architecture consists of five components as shown in Fig. 3. In the figure, an interaction between two components is represented as a pair of connected interface and receptacle. The API components implement the domain-specific interfaces described in Section 3.1. The remaining components realise the common interaction pattern described in Section 3.2. The remainder of this section discusses these components and their interactions in detail.

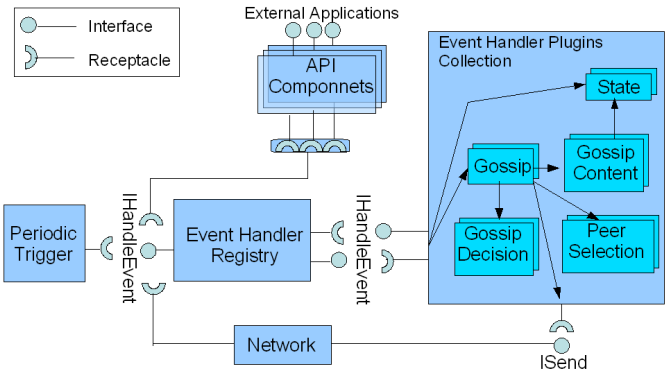


Fig. 3. GossipKit Architecture

4.1 API Components

API components aim to uncouple the gossip protocols implemented by the framework from external applications. Each type of API component provides a generic interface for external applications to access a particular category of gossip protocols. When an interface of an API component is triggered by the connected external application, it raises an event to the event handler registry. Fig. 4 provides an example of how API component interacts with external applications.

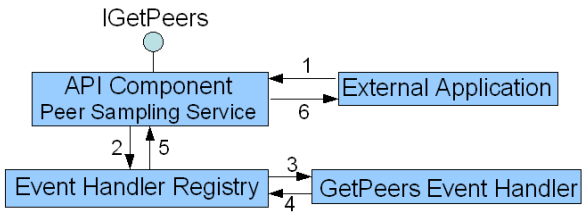


Fig. 4. Interaction of API Component with External Application

This figure shows the API component for peer sampling service protocols. This API component provides an IGetPeers interface for external application or other API components to retrieve peer information collected by the local peer. When IGetPeers is invoked (operation 1 in Fig. 4), the API component generates a GetPeers event to the event handler registry (operation 2). On receiving this event, the registry executes the proper event handler to handle the GetPeers event (operation 3, see section 4.3 below). The event handler then retrieves the peer sampling information stored locally, and returns the information to the API component as the event handling result (operation 4 and 5). Finally, the API component provides the peer sampling information to the external application as the return value of the IGetPeers interface (operation 6).

4.2 Periodic Trigger Component

The periodic trigger component is optional in the framework. It is only loaded when the framework is used to support proactive gossip protocols. This component periodically dispatches events to trigger specific event handlers that perform different styles of gossiping, such as pull, push or pull-push. The event-dispatching period (the gossip frequency) is predetermined at deployment phase, and can be reconfigured at runtime.

4.3 Event Handler Registry

The event handler registry serves as a broker between event handlers and event producers (components that raise events). The event handler registry maintains a table that records event handler IDs with their associated events (i.e. events that an event handler can handle). When an event handler's IHandleEvent interface is connected to the registry, the registry's table records the events bound to the event handler. The event handler registry also provides an IHandleEvent interface to event producers to trigger the events. On the invocation of an event, the event registry finds and executes the registered event handlers that are bound to this particular event type.

It is worth pointing out that the IHandleEvent interface can also be used by the event handlers themselves. This allows events raised internally within an event handler to be handled by others, thus providing a consistent event-based environment and facilitating interoperability between different gossip protocols.

4.4 Event Handler Plugins

As mentioned in Section 3.3, we considered modules that can be further decomposed to finer-grained micro-modules (i.e. Gossip, Peers Selection, Gossip Decision, Gossip Content, and State in Fig. 1) to be developed as a collection of event handlers. This is reflected by the event handler plugins in Fig. 3. In the figure, multiple micro-modules belonging to each particular module are designed as event handler plugins that are contained in the event handler plugin collection. Micro-modules for the Gossip module and the State module can be invoked by the event handler registry to handle events generated by the periodic trigger component, the API components, and the network component (see below). Micro-modules for the Gossip module can also send messages using the interface provided by the network component. Furthermore, each micro-module can invoke the IHandleEvent interface provided by the event handle registry to interoperate with other micro-modules.

4.5 Network Component

This component provides network level communication to other components, and as such is responsible both for sending messages generated by the Gossip module and for delivering message events received from the network to the event handler registry. Through this component, gossip protocols within the GossipKit framework can operate on transport layers such as UDP, TCP, or ad-hoc routing. The network component can also operate on virtual transport layers in order to utilise the features provided by various component-based virtual overlays such as GridKit [19].

5. Implementation

GossipKit's prototype implementation is based on the Java version of OpenCom [15], a lightweight, efficient and reflective component model. Java's portability enables GossipKit to operate on various platforms, from desktop computers through to PDA. We implemented the micro-modules and event handler plugins shown in Fig. 3 as individual OpenCom components, while we realised events with a normal Java class. This class contains: (i) a header that identifies the type of the event, (ii) a body containing data to be handled by the corresponding event handlers, (iii) a source ID identifying the peer that generated the event, and (iv) a target ID that defines the target peer that should receive the event.

It is worth emphasising the implementation of the periodic trigger component, which can be viewed as a task scheduler that can be utilised by multiple protocols to perform periodic gossiping with different frequencies. Its implementation only requires a single Java thread rather than spawning one thread for each proactive gossip protocol. If multiple proactive gossip protocols operate concurrently at runtime, the resource utilisation of the system can be significantly improved by minimising the use of resource-consuming multi-threading. This effectively reduces memory usage if GossipKit operates on mobile devices that are resource constraint.

6. Early Evaluation

We evaluated our GossipKit framework on two categories of gossip protocols: We implemented three peer-sampling services (SCAMP [9], PSS [17], and the topologic construction protocol described in T-Man [5]), and two reliable multicast protocols (Bimodal Multicast [2], and Lpbcast [1]). In the following, we focus our evaluation on the reusability of the GossipKit framework (Section 6.1). We then briefly discuss the configurability and reconfigurability of our framework in Section 6.2.

6.1 Reusability

We evaluated the reusability of GossipKit using a quantitative measuring approach suggested in [18]. This approach measures the size of the Java classes that make up different configurations of components. In Fig. 5, the first three configurations indicate the cost of each individual protocol in the framework (a tick means the protocol is selected in the configuration). The size of configuring multiple protocols is measured in Configurations 4-6. These measurements are compared against the side-by-side measurement of individual protocols. It can be seen that compiled Java code size is reduced by about 33% in Configuration 4 and 5, and 48% in Configuration 6. These results show that the GossipKit framework does not only promote code reuse for developing gossip protocols that belong to the same

category (SCAMP and PPS in Configuration 4 belong to the peer sampling category), but also for those belong to different categories (PPS and Bimodal Multicast in Configuration 5). Furthermore, the evaluation results indicate the reusable quantity increases as more gossip protocols are deployed in GossipKit (Configuration 6).

				Framework Size (KB)	Side by Side Size (KB)
Protocols / Config.	SCAMP	PPS	Bimodal Multicast		
Config 1	✓			20.39	
Config 2		✓		26.57	
Config 3			✓	37.31	
Config 4	✓	✓		31.19	46.96
Config 5		✓	✓	38.92	63.88
Config 6	✓	✓	✓	43.55	84.27

Fig. 5. Reusability Measurement.

6.2 Configurability and Reconfigurability

GossipKit offers a common component architecture to simplify the configuration of gossip-oriented middleware. It does so by providing module types and connection bindings between modules that remain the same regardless of the implemented protocols. However, the use of fine-grained micro-modules in GossipKit's event-driven architecture can make configuration a time-consuming process. Although an event-driven architecture simplifies the configuration of micro-modules into modules as discussed in Section 3.3, the manual configuration of event bindings for a large number of micro-modules still remains a time-consuming task, in particular when a user needs to deploy a number of gossip protocols to operate concurrently within GossipKit. From our experiences on the development of five gossip protocols, we have noticed that GossipKit eases the configuration process for these gossip protocols to a certain level. However, further study is required to evaluate whether GossipKit can support easy configuration of a broader range of gossip protocols.

GossipKit supports fine-grained reconfiguration to adapt to environmental changes — different protocol behaviours can be achieved by replacing a simple single component. For instance, a proactive gossip protocol that provides peer sampling service can be modified to support number averaging by replacing the stateful event handler, and the network component that supports communication for multiple gossip protocols can be replaced by another routing scheme. This form of component replacement relies on the mechanisms directly provided by OpenCOM. A detailed discussion of these mechanisms is however out of the scope of this paper.

7. Conclusion and Future Work

This paper has presented GossipKit, an event-based gossip protocol framework. This framework aims to facilitate the development of configurable and reconfigurable middleware that supports multiple gossip protocols potentially operating in parallel under different types of networks. We have presented an early prototype implemented using a reflective component model (OpenCom), and we have discussed some of the benefits we have observed when implementing several gossip protocols with our framework. Our early evaluation indicates that GossipKit promotes code reuse, simplifies configuration for deploying gossip protocol middleware, reduces the overhead for runtime reconfiguration, and minimises the resource usage at runtime to a certain level.

In the future, we plan to explore a broader range of gossip protocols in order to identify more domain-specific features and to improve the genericity of the common interaction model. We are also currently building a configuration tool to allow users to describe a selection and composition of micro-modules, and to automatically configure event bindings of event handlers in order to address the issue discussed in Section 6.2. Furthermore, we plan to utilise the self-organising features of gossip protocols to improve GossipKit towards a self-adaptive framework so that it can automatically reconfigure itself and adapt to changes in its environment.

References

[1] P. Eugster, R. Guerraoui, S. Handurukande, A.-M. Kermarrec, and P. Kouznetsov, *Lightweight Probabilistic Broadcast*. In IEEE International Conference on Dependable Systems and Networks(DSN2001), July 2001.

[2] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu and Y. Minsky, *Bimodal multicast*. TR99-1745, May 11, 1999.

[3] R. Renesse, K. Birman, M. Hayden, A. Vaysburd and D. Karr, *Building Adaptive Systems Using Ensemble*. Cornell University Technical Report, TR97-1638, July 1997.

[4] M. Hiltunen and R. Schlichting, *The Cactus Approach to Building Configurable Middleware Services*. Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000), Nuremberg, Germany (October 2000).

[5] M. Jelasity and O. Babaoglu, *T-Man: Gossip-based overlay topology management*. In EngineeringSelf-Organising Systems: Third International Workshop (ESOA 2005), Revised Selected Papers.

[6] O. Babaoglu, *Gossiping in Bologna*. http://www.cs.cornell.edu/Courses/cs514/2007sp/UniBo%20Project/Leiden-Gossip.ppt.

[7] K. Birman, A. Abbadi, W. Dietrich, T. Joseph and T. Raeuchle, *An Overview of the ISIS Project*. IEEE Distributed Processing Technical Committee Newsletter. January 1985.

[8] N. Bhatti, M. Hiltunen, R. Schlichting and W. Chiu, *Coyote: A System for Constructing Fine-Grain Configurable Communication Services*. ACM Transactions on Computer Systems, November 1998.

[9] A. Ganesh, A.-M. Kermarrec and L. Massoulie, *SCAMP: Peer-to-Peer Lightweight Membership Service for Large-Scale Group Communication*. In Proc. of the 3rd International workshop on Networked Group Communication, 2001.

[10] D. Agrawal, A. E. Abbadi and R. Steinke, *Epidemic algorithms in replicated databases*. In Proc. 16th ACM Symp. on Principles of Database Systems, 1997.

[11] R. van Renesse, Y. Minsky and M. Hayden, *A gossip-style failure-detection service*. In Proc. IFIP Intl. Conference on Distributed Systems Platform and Open Distributed Processing, 1998.

[12] I.Gupta, R. van Renesse and K.Birman, *Scalable fault-tolerant aggrgation in large process groups*. In Proc. Conf. on Dependable Systems and Networks, 2001.

[13] A. Demers, D. Greene, C. Hauser et al. *Epidemic algorithms for replicated database maintenance*. In Proc. of the sixth annual ACM Symposium on Principles of distributed computing, 1987.

[14] Z. Haas, J. Halpern and L. Li, *Gossip-based Ad-Hoc Routing*. Unpublished. http://citeseer.ist.psu.edu/article/haas02gossipbased.html

[15] M. Clarke, G. Blair, G. Coulson and N. Parlavantzasco *An efficient component model for the construction of adaptive middleware*. In Proc. of IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware). Germany, 2001.

[16] M. Hiltunen, F. Taiani and R. Schlichting, *Reflections on Aspects and Configurable Protocols*. The Fifth International Conference on Aspect-Oriented Software Development (AOSD.06), Bonn, Germany, March 20-24, 2006, pp.87-98 (12 p.).

[17] M. Jelasity, R. Guerraoui, A.-M. Kermarrec and M. Steen, *The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations*. Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware, Toronto, Canada, 2004, pp. 79-98.

[18] C. Flores-Cortes, G. Blair and P. Grace, *A Multi-protocol Framework for Ad-Hoc Service Discovery*. In Proc. of the 4th International Workshop on on Middleware for Pervasive and Ad-Hoc Computing (MPAC '06), co-located with Middleware 2006, Melbourne, Australia, 2006.

[19] P. Grace, G. Coulson, G. Blair et al. *GRIDKIT: Pluggable Overlay Networks for Grid Computing*. In Proc.of International Symposium on Distributed Objects and Applications(DOA), Larnaca, Cyprus, 2004.

Enabling Cyber Foraging for Mobile Devices

Mads Darø Kristensen
Center for Interactive Spaces, ISIS Katrinebjerg
Computer Science Department, University of Aarhus, Denmark
Email: madsk@daimi.au.dk

Abstract

This paper presents the LOCUSTS *framework. The aim of the* LOCUSTS *project is to enable easy use of cyber foraging techniques when developing for small, resource-constrained devices. Cyber foraging, construed as "living off the land", enables resource poor devices to offload tasks to nearby computing machinery, thereby enabling the small devices to 1) save energy and time, 2) take on tasks that would normally not be possible on such small devices, and 3) co-operate to perform tasks.*

This paper is concerned with foraging for processing power, i.e. remote execution of tasks, and discusses how distribution and migration of tasks can be done in a highly mobile environment.

The main contribution of LOCUSTS *is the focus on highly mobile cyber foraging. Here highly mobile means two things: 1) that the mobile devices are physically moving through the environment, which calls for task migration, and 2) that this mobility moves the devices into unknown environments where they would still like to be able to perform cyber foraging, which calls for the use of mobile code.*

1. Introduction

In recent years the usage of small mobile devices has increased dramatically. Today most people own a mobile phone, many have PDAs, and other forms of mobile devices, such as electronic gaming devices, are gaining in popularity. A shared characteristic of these devices is that they are in some way resource-constrained devices, if for no other reason then at least because they are all battery powered. Because of the scarcity of resources users of mobile devices often avoid doing resource intensive work when using these devices, because executing such tasks is very slow and consumes a disproportional amount of energy.

Cyber foraging, as described by Satyanarayanan [1] and Balan *et al.* [2], enables the mobile devices to take on more resource intensive tasks by leveraging unused resources on larger computers in the vicinity. Cyber foraging is foraging for a multitude of resource types – not just processing power. Among the resources that could be foraged for is network connectivity or bandwidth, storage, processing power, and much more. All of these resource types are equally important in a cyber foraging scenario. In the present article, however, only foraging for processing power will be considered.

There are many possible usage scenarios where cyber foraging can be utilised. Some visions for pervasive computing calls for *wearable computing* devices - i.e. small computing devices that may be worn by their users like clothes, e.g. see [3]. Users of such devices are obviously not interested in carrying around heavy equipment, and these devices must therefore be as lightweight as possible. This is counter to the user's wish to have as powerful a device as possible. The desired computing power can be added to these small wearable devices through techniques such as cyber foraging.

Consider the following scenario: a doctor doing house calls is wearing a small headset (similar in size and form to the well-known Bluetooth headsets for mobile phones). Using this headset he would like to be able to enter information about his patients into an electronic journal. This means that the headset is faced with the difficult task of continuous voice recognition. The headset is unable to perform this translation task itself, so instead of performing the actual voice recognition it merely records the utterances made by the doctor. Whenever the headset comes within range of usable computing resources (surrogates) it forwards some of the recordings to these machines who respond by returning the translated text. If the surrogate has an Internet connection it may even be given the task of updating the patient's journal directly. After translation the headset may discard the recording and

thus free storage for additional recordings.

A notable thing in the preceding scenario is that the application running on the mobile device works in two modes; *high fidelity* and *low fidelity*, as defined by Noble *et al.* in [4]. When no surrogates are within range the headset simply stores the recordings (low fidelity), and when surrogates can be used the recordings are immediately translated into text (high fidelity). This high/low fidelity aspect is inherent in all cyber foraging applications – when surrogates are available high quality work may be done, but this does not mean that the applications will only work in the presence of surrogates. For cyber foraging to be usable a low fidelity setting must also be possible, where the mobile device itself is running the application, albeit at a diminished fidelity. In the scenario low fidelity means that the headset only stores the recordings, but it would also be possible to ask the mobile device to do the processing itself, or even to do it in conjunction with other mobile devices that reside in the doctors personal area network.

To be able to perform the actions described above a number of things are needed. First off the mobile device must be able to monitor the network looking for any available surrogates. Once found the mobile device must be able to distribute tasks to surrogate machines, and, in the case that the user is moving while tasks are being performed, surrogates must be able to migrate tasks between each other so that the result may be returned. These are fairly complex operations, and it should not be the responsibility of the application programmer to implement this. LOCUSTS aims to create a toolbox so that a developer just needs to mark the pieces of code that could be distributed, and then the rest will be taken care of by the framework; distributing computation to and migrating tasks between surrogates as needed. Such frameworks have been proposed before [5] [6], but these approaches have no provisions for distribution of code or task migration, and as such they do not consider the high level of mobility and flexibility that LOCUSTS caters for.

In Section 2 the main challenges faced in cyber foraging are presented, then, in Section 3, the design of the LOCUSTS framework is presented and its architecture is described to show how these challenges will be met. Related work is discussed in Section 4 and the paper is concluded in Section 5.

2. Cyber Foraging Challenges

There are a number of challenges that must be addressed when designing a framework for cyber foraging. In this paper only the two challenges most central for the remote execution of tasks will be discussed:

- **Task distribution.** How can tasks be delegated to surrogates, and exactly *what* should be moved onto the surrogates.
- **Task migration.** When mobile devices are using surrogates the tasks that are distributed to surrogates must be migratable – i.e. it must be possible to move running tasks between surrogates and also to move a task back to the mobile device.

Apart from these remote execution specific challenges a number of other challenges are posed as well in a cyber foraging framework, challenges such as device discovery, capability announcement, data staging, etc. These are of course covered in the design of LOCUSTS, but are not described in any detail here.

One final critical challenge for cyber foraging is security. Surrogates must execute code on behalf of, possibly unknown and thus untrusted, mobile devices, and data must be transmitted over wireless links that are easy for an eavesdropper to monitor. Finally, the client must be able to trust that the surrogate actually performs the task that it is asked to. How can this be done in a secure manner? A fine balance between security and flexibility must be found here. A full description of how this is handled in LOCUSTS is beyond the scope of this paper.

3. The LOCUSTS Framework

The LOCUSTS framework aims to provision developers with a complete cyber foraging toolbox that can ease the process of developing applications that utilise cyber foraging. In the following the architecture of LOCUSTS will be briefly described in Section 3.1, then the chosen approach towards task distribution is described in Section 3.2. Finally, task migration is illustrated in Section 3.3.

3.1. Architecture

A simplified view of the current architecture of LOCUSTS is depicted in Figure 1. The LOCUSTS daemon is running as a separate process on both client and surrogate devices, and the individual applications can communicate with the local LOCUSTS instance. As shown, a cyber foraging enabled application consists of some local code, executed by the local device, and a number of distributable tasks. Whenever a task is executed, the local LOCUSTS instance is contacted so that it may find a suitable execution plan. This execution plan is created by the scheduler which relies on resource measurements, both local and remote, and

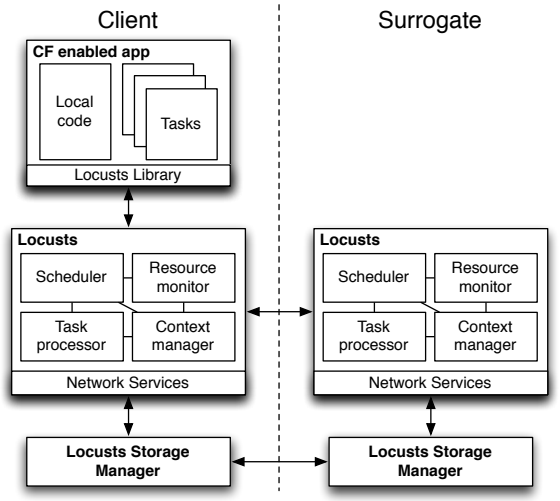


Figure 1. LOCUSTS architecture.

in some cases also on storage specific information such as input availability. When an execution plan has been derived the task may either be distributed to one or more surrogates, or it may be handed over to the task processor for local execution. At the bottom of the LOCUSTS client is the network services. These enable the mobile node to do necessary P2P operations such as peer discovery, network roaming etc. All devices can choose to act as surrogates and thus the software running on surrogates and clients is the same. When operating as a surrogate, a device basically offers three things: 1) to execute known tasks an behalf of clients, 2) allowing clients to author new tasks, and 3) full task migration support. Describing these three subjects fully is out of the scope of this paper, but a short description is given in the following sections.

The storage manager shown below the LOCUSTS daemon in Figure 1 provides a simple file system that can be accessed by tasks executed by LOCUSTS. The storage manager provides a virtual file system that can be accessed from within tasks. When executing a task on the local device, files in this virtual file system simply point to the local files, but when a task is delegated for remote execution, the storage managers of the client and the surrogate are *linked*, so that remote files may be read transparently. This small distributed system is kept as simple as possible and is designed specifically for the purpose of cyber foraging. It uses on-demand synchronisation of file data to reduce the amount of data transferred, and has built in support for temporary files that will only be synchronised if the task is migrated.

3.2. Task Distribution

Task distribution is at the core of cyber foraging; the delegation of heavy work to surrogates is the whole idea of cyber foraging. When designing a cyber foraging framework it must be decided exactly what is delegated, when it is delegated, and at which granularity.

The question about task granularity is a hard one to give a definite answer to, since it depends on a number of factors that may vary from system to system. The main factors to consider are network bandwidth and latency, processing power of the mobile device and the surrogate, the amount of energy used at the mobile device when communicating with the surrogate, and the velocity of the mobile device. When delegating tasks to a surrogate the mobile device needs to send the task to the surrogate, and likewise the surrogate must transmit a response back to the mobile device. This means that data must be transmitted over the wireless link between the mobile device and the surrogate. It must be considered whether the cost of this transmission, both in time and energy, is acceptable, i.e. whether the cost of distributing the task is smaller than the cost of doing the processing locally. To this end it makes sense to distribute only larger, longer running tasks, since the cost of delegating a small task will exceed the cost of local execution. But what designates a small task? This will vary from situation to situation depending on all the factors mentioned above.

To approach this challenge, decisions about when to distribute a task must be taken dynamically depending on the current resource availability. This is done by monitoring resource usage at the client and surrogates and using this information in the scheduler when planning future execution. This is also the approach taken in existing remote execution systems such as Spectra [5] and Chroma [6]. The LOCUSTS framework takes the same approach towards task distribution; monitoring resource usage and dynamically deciding where and when to distribute tasks. Aside from resource monitoring and subsequent planning LOCUSTS also works with the concept of resizable tasks. A resizable task is a task that can be solved to different degrees, in some ways similar to *fidelity* as introduced Noble *et al.* in [4]. But, apart from being able to solve the task at different fidelities, a resizable task in LOCUSTS may also be solved to a certain degree, meaning that a surrogate may choose to solve only a small fraction of the task before returning the task to the client. This will normally be done when the surrogate is subject to timing constrains given by the client, e.g. if a highly mobile client only allows the surrogate to use one second on the task to make sure that

it will receive the answer before going out of range. Related to this, LOCUSTS also works with the concept of migratable tasks as will be described in Section 3.3.

The next important question to answer is exactly *what* is distributed and how it is done. When a mobile device is running an application that is capable of utilising cyber foraging, a part of the application will always be running locally while other parts may or may not be distributed to surrogates. Just like when working with parallelising programs, the task of preparing a program for distribution requires some work by the developers of the program. The parts of the program that can be distributed must be identified and, possibly, altered to make the distribution possible.

After identifying the parts of a programs that can be delegated to surrogates a mechanism for actually distributing these tasks must be found. Existing cyber foraging systems use RPC for remote execution, and the surrogates must have the software behind these RPCs installed prior for the clients to be able to utilise them. LOCUSTS differs from these systems because it strives to do away with the need to have anything pre-installed on surrogates. A task in LOCUSTS is therefore more than just an RPC invocation, it also contains the actual source code of the task represented in a way such that any surrogate, regardless of architecture etc., will be able to execute it. This means, that the portions of the code that designate distributable tasks must be written in a specific, interpreted language so that it can be moved on to surrogates, and thus allowing the clients to *author* new tasks on the surrogate. Allowing clients to execute unknown code on surrogates of course leads to an abundance of security issues that will have to be addressed, but that is out of the scope of this paper. Currently the language used for distributable tasks is Python but other interpreted languages could be used as well.

3.3. Task Migration

Distributing very large tasks increases the benefits of remote execution, since it helps to diminish the overhead of sending tasks back and forth. But, in existing cyber foraging frameworks, working with large tasks requires the user to stay within range of a specific surrogate for an extended period of time. To alleviate this problem, provisions have to be made so that tasks may span multiple surrogates throughout their lifetime. The solution to the problem is task migration. Task migration enables surrogates to move running tasks to other surrogates or even back on to the mobile device. Using migration a client no longer needs to stay within range of a surrogate while performing a task, and it is thus possible to distribute larger tasks, which alleviates the considerable overhead of remote execution. Task migration is depicted in Figure 2.

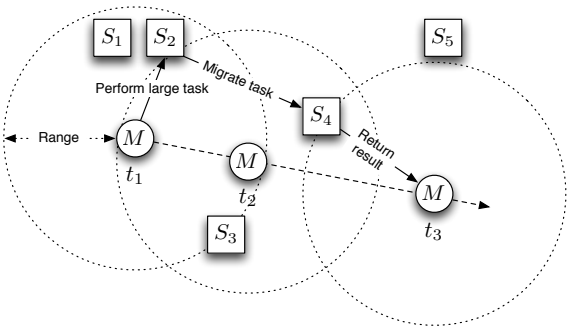


Figure 2. Migration of a single task. In this scenario a mobile device M is depicted at time t_1, t_2, and t_3. A task is initiated on surrogate S_2 at time t_1, migrated to S_4 at time t_2 when M moves out of range, and finally the result is returned by S_4 at time t_3.

The ways that such task migration could be implemented range from simple surrogate-to-surrogate proxies to task checkpointing. Both methods are used in LOCUSTS. Proxies are used in some circumstances when high speed network connections exist between surrogates. Take for example the scenario in Figure 2. The task is initiated at S_2 but when M moves out of range of S_2 the task is migrated to S_4. If, for some reason, it makes sense to let S_2 keep the task S_4 will simply be asked to proxy for S_2. In the eyes of the client M surrogate S_4 is the one executing the task and all communications regarding the task goes through S_4. Alternatively, the task could be moved entirely to S_4. In this case the running task would be checkpointed by S_2 and its code and state sent to S_4. A multitude of factors must be considered when choosing which kind of migration to use – factors such as network bandwidth between the surrogates, current resource usage at the surrogates, checkpoint size, estimated finish time of the task etc.

This preceding description of task migration touches lightly on a very complex matter – the full complexity of task migration and how it is implemented in LOCUSTS is outside the scope of this papers.

4. Related Work

Remote execution of tasks is a well-studied research field, but remote execution of tasks done by mobile, resource constrained devices is less so. Spectra, described by Flinn *et al.* [5], and Chroma, described by Balan *et al.* [6], are two related examples of remote execution frameworks that consider these factors. These

systems include very sophisticated schedulers that try to dynamically find the best execution plans for a given task execution. They measure resource usage on different levels, both at the client, at the surrogates, and in the network, and thus try to choose the best possible placement of the tasks. Spectra and Chroma thus solve many of the problems when using remote execution in a cyber foraging setting. One shortcoming of these systems, when considered in the highly mobile usage scenarios envisioned for the LOCUSTS framework, is that they do not provide any means for dynamically distributing the *code* of the tasks. Surrogates must therefore be prepared beforehand to enable the execution of tasks. Furthermore, they do not consider migration of tasks between surrogates. Likewise, Spectra and Chroma use the Coda [7] filesystem for data staging, which, in a highly mobile scenario such as the ones LOCUSTS aims to support, would be too complicated.

The Coign system, described by Hunt and Scott in [8], makes distribution of tasks possible without even altering the source code of the application. But, as it is also noted by Flinn *et al.*, a little application-specific knowledge can go a long way when preparing an application for distribution. In many cases the inclusion of distribution alters entirely the way to think about a given program, e.g. it may make sense to execute parts of an initially linear program in parallel (possibly even on multiple surrogates). Such optimisations would be hard to detect for an automated distribution algorithm. Ideally a cyber foraging framework should cater for both kinds of distribution, falling back to automatic distribution when no instrumented version of an application is available. The Coign system only works on applications consisting of Microsoft COM components, and is therefore limited to distributing applications running on different versions of Microsoft Windows. This is a big limitation in a cyber foraging setting where many different kinds of mobile devices must be supported. Furthermore, Coign is *not* a cyber foraging framework – it is only concerned with the partitioning of applications.

5. Conclusion

This paper presents the LOCUSTS framework. The LOCUSTS framework extends cyber foraging to encompass highly mobile foraging for resources in unknown environments. The focus on unknown, or unprepared, environments leads to new challenges in task distribution since unknown code has to be distributed to untrusted surrogates. The focus on mobility means that task migration becomes a necessary part of LOCUSTS. This has to our knowledge not been studied in detail before in a cyber foraging setting.

LOCUSTS is still a work in progress and much work and experimentation needs to be done. The most challenging future questions are: 1) how can code be distributed safely in an untrusted environment? 2) how will task migration perform compared to converting the problems into smaller tasks? 3) Which features are needed in a minimal distributed file system to fully support usage in a cyber foraging setting?

Acknowledgements

This paper has been funded by a research grant from the Danish Research Council for Technology and Production Sciences.

Furthermore, I would like to thank Niels Olof Bouvin for proof-reading and providing helpful insights into the subject matter.

Finally, I would like to thank the anonymous reviewers for their excellent and very insightful comments. I hope to have honoured most of their requests for improvement in this version of the paper.

References

[1] M. Satyanarayanan, "Pervasive computing: vision and challenges," *Personal Communications, IEEE*, vol. 8, no. 4, pp. 10–17, 2001.

[2] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang, "The case for cyber foraging," in *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC*. New York, NY, USA: ACM Press, 2002, pp. 87–92.

[3] S. Mann, "Wearable computing: a first step toward personal imaging," *Computer*, vol. 30, no. 2, pp. 25–32, 1997.

[4] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker, "Agile application-aware adaptation for mobility," in *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, vol. 31, no. 5. New York, NY, USA: ACM Press, December 1997, pp. 276–287.

[5] J. Flinn, S. Park, and M. Satyanarayanan, "Balancing performance, energy, and quality in pervasive computing," *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pp. 217–226, 2002.

[6] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb, "Simplifying cyber foraging for mobile devices."

[7] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: a highly available file system for a distributed workstation environment," *Transactions on Computers*, vol. 39, no. 4, pp. 447–459, 1990.

[8] G. C. Hunt and M. L. Scott, "The coign automatic distributed partitioning system," in *Operating Systems Design and Implementation*, 1999, pp. 187–200.

Peer-to-Peer Strategies for Location-based Publish/Subscribe with Persistent Events in Wireless Settings

Patrick Eugster[1], Benoît Garbinato[2], Adrian Holzer[2], Jun Luo[3]

[1] Purdue University, West Lafayette, USA
[2] Université de Lausanne, Lausanne, Switzerland
[3] University of Waterloo, Ontario, Canada

Abstract

In this paper, we compare two peer-to-peer implementation strategies for persistent publications in location-based publish/subscribe. Both approaches use a scoped flooding algorithm to effectively disseminate data through the network – the first naive alternative propagates only published events, while the second smarter approach disseminates subscriptions during a warm up *phase and then routes matching publications with a multisend communication primitive. We show in preliminary performance evaluations that depending on the applicative scenario, the cost of the warm up phase cannot be covered and the seemingly naive approach can outperform the smart one.*

1 Introduction

Publish/subscribe programming models offer many characteristics which seem intuitively appealing for bulding applications involving mobile clients and wireless communication, i.e., running in MANETs. Several authors have advocated for publish/subscribe abstractions in these settings. To address the *geographic* dependence of communication in MANETS, i.e., the fact that nodes move about, these abstractions proposed virtually all include a notion of "location context" attached to publications and/or subscriptions which is logically decoupled from the events themselves and reflects spatial proximity semantics (e.g., [1, 2, 3, 4]).

The second main characteristic of MANETS is their *chronologic* dependence of communication, resulting from the fact that communication enjoys no wired infrastructure but has an ad-hoc and thus transient nature. *Persistent* publications are a means to address this second dimension of dependency, consisting in associating explicit lifetimes to published events rather than propagating these events in a "fire-and-forget" manner. These criteria go hand-in-hand with location semantics, since not all nodes in the prescribed perimeter might be reachable at the very moment of publication. [5, 6] offer such a feature but do not provide a means to restrict events to a certain proximity if needed.

An issue when implementing such a service is the *matching strategy*. Two kinds of approaches have been proposed: (1) publication-centric approaches [2],[1] where only publications are disseminated and where matching is done on the subscriber side and, more recently (2) subscription-centric approaches [7, 8],[2] where publishers perform the matching and route publications using the previously received subscriptions. Intuitively, the former is usually based on unconditional flooding and thus tends to congest the network more. The latter on the other hand, needs a warm up phase in order to gather routing information.

In this paper, we compare both strategies and try to evaluate the performance of both alternatives. To do so, we present two fully decentralized implementation alternatives for persistent publications in location-based publish/subscribe, one using a publication-centric strategy and another one using a subscription-centric strategy. The preliminary results obtained using a simple network topology, convey the fact that there are applicative scenarios in which the cost of the warm-up phase of a subscription-centric approach cannot be covered by the efficiency of its routing strategy and where a publication-centric strategy is preferable.

The rest this paper is organized as follows: Section 2 describes the location-based publish / subscribe paradigm while Section 3 presents the proposed strategies, Section 4 discusses the performance evaluations whereas Section 5 concludes the paper.

[1] For lower overhead, publications are only valid in a certain range.
[2] For better scalability, routing is delegated to forwarding nodes.

2 Location-based Publish / Subscribe

With location-based publish/subscribe [1, 2, 3, 4], matching criteria include dynamic event content as in industrial-strength solutions for wired settings, but in addition comprises *geographical context*. With our notion of context, an event is typically restricted to a defined range around the publisher (subscriber). This range defines the publication (subscription) space.[3] To be called a location match, both subscriber and publisher must be located in the intersection of both spaces. So in order to be called a match, a publication/subscription couple has to meet both conditions: (1) the location match and (2) the content match.

Persistent publications. In many application scenarios, subscribers should eventually deliver geographically and content-wise matching publications even though they might not be reachable at publication time. To a similar end, conventional publish/subscribe systems, for wired settings, offer some notion of durable subscribers that can be registered with a topic. Such a subscriber gets notified of publications that occurred while it was unreachable, as soon as it comes back online. As there is no notion of topic in LPSS to restrain "applicability" of events, we introduce persistent publications to address the issue and allow subscribers that enter a publication range to deliver previously published events. Note that unlike the wired system solution, here a persistent publication is linked to its publisher.

Location-based publication. We represent a publication p=$\langle e, h, t, \Delta t, \Delta x \rangle$ abstractly as an event e distributed at time t within a determined geographical range Δx around the position of a publisher represented by host h. This delimits an area termed the *publication space*. An event e itself is a record of attributes $[a_0 = v_0, ..., a_k = v_k]$. A publication is valid by default for a predetermined timespan Δt or until it is explicitly unpublished. Any element of a publication p can be accessed by dereferencing p e.g. $p.\Delta x$.

Location-based subscription. A subscription is defined as s=$\langle h, \Delta x, \Delta e \rangle$ and is a request to receive events published by producers located within a determined geographical range Δx around the position of the node h hosting the corresponding subscriber on host . This delineated area is known as the *subscription space*. Content-based selection of events can be achieved via an *event template* Δe, which represents a record of attributes alike events themselves. The content match condition is met when the published event contains at least all attributes specified in the subscribed event template, and matching values for those. This is similar to what most common content-based publish/subscribe platforms offer.

[3] We assume that each node has access to its own location via a location provider such as a GPS receiver for example.

3 Peer-to-Peer Location-based Publish / Subscribe Implementation Strategies

In this section, we present two implementation alternatives of the peer-to-peer location-based publish/subscribe service (LPSS). The first alternative can be characterized as *publication-centric* as it only propagates publications through the network similar to [8, 2]. The second alternative can be qualified as *subscription-centric* as it first propagates subscriptions through the network and then routes matching publications similar to [7]. Our service offers five primitives:

LPS-PUBLISH(p) - publishes a publication p.

LPS-SUBSCRIBE(s) - creates a subscription s.

LPS-UNPUBLISH(p) - removes a persistent publication p.

LPS-UNSUBSCRIBE(s) - removes a subscription s.

LPS-STANDUP(p,s) - works as a callback and notifies the host that a publication p matches the subscription s.

Before further detailing the alternatives, we present the underlying services they rely upon, namely the Scoped Flooding Service (SFS) and the Multisend Servics (MS).[4]

Definitions. In the following, we will use the notions of *neighborhood* and *group*. A neighborhood at time t for a given node denotes the set of peers within the considered node's communication range at time t. We distinguish two types of groups: a publication group for a given publication at time t represents the set of peers within the corresponding publication space at t and a subscription group for a considered subscription at time t denotes the set of peers within the corresponding subscription space at time t.

Scoped Flooding Service (SFS). The Scoped Flooding Service is used to disseminate messages to all peers within a defined geographical range of the source and offers two primitives.

SFS-BROADCAST(m, x, Δx) - disseminates message m to all hosts located within the range Δx of position x.

SFS-DELIVER(m) - works as a callback and notifies the host when a message m is received.

[4] Note that both SFS and MS use the MAC layer to broadcast messages over the physical network

Multisend Servics (MS). The multisend service is a one-to-many communication abstraction where the sender explicitly denotes the destinations. Our simple algorithm creates a route to all destination nodes within a given range starting with the closest to the sender and finishing with the furthest away. For the first node the route with the least communication hops that separates it from the sender is chosen. All nodes that have to relay the message are added to a forwarder set and all nodes within the one hop range from such a forwarder are placed in a set for reached nodes. Then the route to the next unreached destination is calculated as the shortest path to the closest forwarder, and the forwarding and reached nodes are again added to their respective sets and so on. According to its functionality, the service offers two primitives.

MS-SEND($m, \{h_1, ..., h_n\}$) - multisends a message m to the set of hosts $\{h_1, ..., h_n\}$.

MS-DELIVER(m) - is used as a callback when a message m is received.

3.1 Publication-centric Solution

Hereafter, we present the publication-centric implementation of the decentralized LPSS. We detail the two main primitives provided by the LPSS interface, namely the LPS-PUBLISH and LPS-SUBSCRIBE primitives. We also detail the REFRESH task which is key to event persistency (see Algorithm 1).

The publication primitive. When a publication p is created, LPS-PUBLISH is called with p as parameter (line 2) and p is stored in the $myPubs$ set for newly created publications, until it is either unpublished explicitly through the LPS-UNPUBLISH primitive, or automatically by the REFRESH process after its time-to-live has elapsed.

The subscription primitive. When a subscription s is created, LPS-SUBSCRIBE is called with s as parameter (line 4) and s is stored in the $mySubs$ set which stores all of the peer's subscriptions until it is removed by the LPS-UNSUBSCRIBE primitive. When a host receives a publication through the SFS-DELIVER callback, it adds it to the set of received publications until it is processed by the REFRESH task.

The refresh task. This task is executed every Δt and republishes the peer's persistent publications with the updated location information ($myLoc$) to ensure that all subscribers entering the publication space before p is unpublished will eventually deliver it, discards out-of-date publications received from other peers, and unpublishes the peer's own out-of-date publications (line 9) and matches all received publications with the peer's own subscription stored in $mySubs$. When a match is found the publication is delivered using the LPS-STANDUP callback (line 19).

```
 1: uses Scoped Flooding Service (SFS)

 2: To execute LPS-PUBLISH(p) :
 3:     myPubs ← myPubs ∪ p                          {adds p to own pub list}

 4: To execute LPS-SUBSCRIBE(s) :
 5:     mySubs ← mySubs ∪ s                          {adds s to own sub list}

 6: upon SFS-DELIVER(p) do
 7:     pubs ← pubs ∪ p                              {adds p to received pub list}

 8: task REFRESH
 9:     repeat every Δt
10:         for all p ∈ myPubs do                    {for all created publications}
11:             SFS-BROADCAST(p, myLoc, p.Δx)        {forwards p}
12:             if p.Δt ≠ UNLTD_TTL then
13:                 p.Δt ← p.Δt − Δt                 {decreases the ttl}
14:                 if p.Δt≤ 0 then                  {if p is obsolete}
15:                     myPubs ← myPubs\{p}          {removes p}
16:         for all p ∈ pubs do                      {for all received publications}
17:             MATCH(p)                             {checks for subscriptions matching p }
18:             pubs ← pubs\{p}                      {removes p }

19: function MATCH(p):
20:     for all s ∈ mySubs do                        {for all of the host's subscriptions}
21:         if LOCATIONMATCH(p, s) and CONTENTMATCH(p, s) then
22:             if ⟨p, s⟩ ∋ matches then             {if the match is new}
23:                 LPS-STANDUP (p, s){delivers the matching publication}
24:                 matches ← matches ∪ ⟨p, s⟩       {adds the match}
```

Algorithm 1. Publication-centric LPSS

3.2 Subscription-centric Solution

Hereafter, we present the subscription-centric implementation of the decentralized LPSS. Like in the publication-centric solution, we will detail the LPS-PUBLISH and LPS-SUBSCRIBE primitives as well as the REFRESH process (see Algorithm 2).

The publication process. In this alternative the publication process is the same as in the publication-centric algorithm.

The subscription process. When a subscription s is created, LPS-SUBSCRIBE is called with s as parameter (line 2) and s is stored in the $mySubs$ set for newly created subscription just like in the publication-centric solution. When a host receives a subscription the SFS-DELIVER callback is triggered and the subscription is placed in a set for received subscriptions until it is processed by the REFRESH

task. When a match is received, the MS-DELIVER callback is called with a message m as parameter. This message contains a publication and a list of matching subscriptions. For every subscription in the hosts $mySubs$ set, the LPS-STANDUP callback is triggered with the matching publication-subscription couple as parameter.

The refresh process. This process is executed every Δt and is in charge of four actions: (1) rebroadcasting the peer's subscriptions with updated location information to ensure that all publishers entering the subscription space before s is unsubscribed will eventually deliver s, (2) discarding out-of-date received subscriptions, (3) unpublishing the peer's out-of-date publications (line 24) and (4) matching each of the peer's publications against all received subscriptions, and then multisending each such publication to all nodes with matching subscriptions using MS-SEND (line 12).

```
 1: uses Location Flooding Service (SFS), Multisend Servics (MS)

 2: To execute LPS-PUBLISH(p):
 3:     myPubs ← myPubs ∪ p                    {adds p to the publication list}

 4: To execute LPS-SUBSCRIBE(s):
 5:     mySubs ← mySubs ∪ s                    {adds s to own sub list}

 6: upon SFS-DELIVER(s) do
 7:     subs ← subs ∪ s                        {adds s to received sub list}

 8: upon MS-DELIVER(m) do
 9:     for all s ∈ m.subs do
10:         if s ∈ mySubs then                 {tests if the s is in own sub list}
11:             LPS-STANDUP(s,m.p)             {delivers the matching publication}

12: function MATCH(p):
13:     m ←⊥                                   {message to send}
14:     m.p ← p                                {sets the message's publication}
15:     dests ←⊥                               {list of destinations}
16:     for all s ∈ mySubs do
17:         if LocationMatch(p, s) and contentMatch(p, s) then
18:             if ⟨p, s⟩ ∋ matches then       {if the match is new}
19:                 dests ← dests ∪ s.h        {adds h to dests}
20:                 m.subs ← m.subs ∪ s        {adds s to matching sub list}
21:                 matches ← matches ∪ ⟨p, s⟩  {adds the match}
22:     MS-SEND(m, dests)                      {multisends m to destinations}

23: task REFRESH
24:     repeat every Δt
25:         for all p ∈ myPubs do              {for all created publication}
26:             MATCH(p)                       {checks for subscription matching p }
27:             if p.Δt ≠ UNLTD_TTL then
28:                 p.Δt ← p.Δt − Δt           {decreases the ttl}
29:                 if p.Δt≤ 0 then            {if p is obsolete}
30:                     myPubs ← myPubs\{p}    {removes p}
31:         for all s ∈ mySubs do              {for all created subscriptions}
32:             SFS-BROADCAST(s, myLoc, s.Δx)  {forwards s}
33:         subs ←⊥                            {resets the set of received subscriptions}
```

Algorithm 2. Subscription-centric LPSS

4 Simulation Evaluation

Our performance evaluation is a preliminary study of the behaviour of our algorithms. Its aim is to evaluate the usefulness and relevance of further studying those algorithms. To do so, we assume a simple grid network topology depicted in Figure 1, composed of fixed nodes.

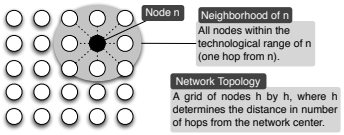


Figure 1. Network Topology

Nodes can broadcast messages to their neighbors. In our topology neighbors are those nodes immediately adjacent to a certain node in the grid. Also instead of a geographical range, we use an amount of network hops to determine the group boundaries. In the following, we first want to quantify the usefulness of scoped flooding in an ad hoc communication environment compared to standard flooding, then we are going to compare scoped flooding with our multisend algorithm to measure the advantages of multisend without its warm up phase, and finally we want to measure the expected performance of both of the algorithms presented in the former section, the first using only scoped flooding to disseminate publications and the second using scoped flooding to disseminate subscriptions and multisend to route matching publications.

4.1 Standard Flooding vs Scoped Flooding

The advantage of scoped flooding over standard flooding in terms of message load is fairly trivial. The more the flooding is scoped the less messages are unnecessarily propagated through the network. The number of forwarders grows exponentially as the range increases following the equation below:

$$forwarderNumber = (2 * (range - 1))^2 \qquad (1)$$

Furthermore, as scoped flooding avoids flooding the whole network with every message, it allows a better usage of the overall bandwidth. For example if two remote nodes, node A and node B, separated by 10 network hops diffuse information to nodes in a range of 4 hops, 2 network partitions will be created and both A and B will be able to fully use the respectively available network bandwidth.

4.2 Scoped Flooding vs Multisend

As shown previously, scoped flooding has a definite advantage over standard flooding in an ad hoc network where

the communication model implies that messages are targeted at recipients located within a certain geographical range from the sender. But the performance in terms of message load obtained with a multisend algorithm are much better, especially if only a subset of the nodes within a group must be reached (see Figure 2).[5] The cost of multisend is twofold: (1) the receivers are not anonymous, and (2) the sender needs to maintain a routing table.

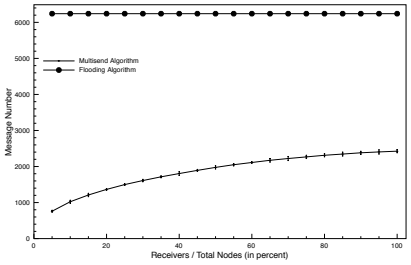


Figure 2. Flooding vs Multisend.

4.3 Publication-centric vs Subscription-centric

Intuitively, if there are few subscriptions and many publications, the costly warm up phase will be repaid by the efficient routing scheme. To quantify the break-even point, we ran simulations using a fixed number of publications (100 publications) in a fixed range (40 hops) and we evaluated both alternatives changing the number of subscriptions. We also evaluated different levels of matching subscriptions (1%, 10%, 50%, 100%). Our results show that if the number of publications and subscriptions is equivalent and the number of matches is very low, both alternatives perform equally well(partner search senario). And if the number of subscriptions is less than 70% of the publications then the subscriptions centric approach will outperform the publication-centric one no matter the level of matching (location-based game senario). But if the subscriptions outnumber publications, the publication-centric approach is most efficient in terms of message overhead (emergency message senario).[6]

5 Conclusion

Finding the right routing scheme for a specific service is not trivial. In this paper, we showed that two different solutions can both be efficient depending on the applicative scenario. In future work, we will thoroughly evaluate these solutions using mobile ad hoc network simulation tools.

[5]Network settings: 40 hops range, 1681 nodes

[6]Such as an ambulance sending messages in a traffic jam.

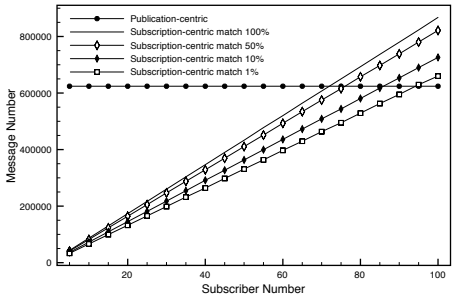


Figure 3. Pub-centric vs Sub-centric

References

[1] L. Fiege, F. C. Gaertner, O. Kasten, and A. Zeidler, "Supporting mobility in content-based publish/-subscribe middleware," in *Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference (Middleware'03)*, 2003.

[2] R. Meier and V. Cahill, "Steam: Event-based middleware for wireless ad hoc network." in *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02) Workshops*, 2002.

[3] C. Sorensen, M. Wu, T. Sivaharan, G. Blair, P. Okanda, A. Friday, and Duran-Limon, "A context-aware middleware for applications in mobile ad hoc environments," in *Proceedings of the 2nd Workshop on Middleware for Pervasive and Ad-hoc Computing*, 2004.

[4] P. T. Eugster, B. Garbinato, and A. Holzer, "Location-based publish / subscribe," in *Proceedings of the 4th International Symposium on Network Computing and Applications (NCA'05)*, Cambridge, 2005.

[5] I. Podnar and I. Lovrek, "Supporting mobility with persistent notifications in publish/subscribe systems," in *Proceedings of the third International Workshop on Distributed Event-Based Systems (DEBS'04)*, 2004.

[6] D. Frey and G.-C. Roman, "Context-aware publish subscribe in mobile ad hoc networks," in *Proceedings of the 9th International Conference on Coordination Models and Languages (Coordination'07)*, 2007.

[7] E. Yoneki and J. Bacon, "Distributed multicast grouping for publish/subscribe over mobile ad hoc networks." in *Proceedings of IEEE WCNC'05*, 2005.

[8] R. Baldoni, R. Beraldi, G. Cugola, M. Migliavacca, and L. Querzoni, "Structure-less content-based routing in mobile ad hoc networks." in *Proceedings of IEEE ICPS'05*, 2005.

Probabilistic Publish-Subscribe for Mobile Ad Hoc Networks

José Mocito
U. Lisboa
jmocito@di.fc.ul.pt

José Côrte-Real
U. Lisboa
cortereal@lasige.di.fc.ul.pt

Luís Rodrigues
U. Lisboa
ler@di.fc.ul.pt

Abstract

This paper proposes a probabilistic publish-subscribe approach for Mobile Ad Hoc Networks (MANETs). In our approach publishing and subscribing are implemented using random walks in the network. This strategy avoids the cost of continuously maintaining complex routing structures that are unstable due to node mobility. While previous research on gossip based publish-subscribe for MANETS assumed the collaboration of all nodes in the network our solution operates on an overlay constructed on top of DSR, allowing for only a small fraction of the nodes to participate in the publish-subscribe system.

1. Introduction

The publish-subscribe communication paradigm is a very powerful tool for designing flexible, reactive and highly decoupled distributed systems, which makes it well suited to highly dynamic systems like Mobile Ad Hoc Networks (MANETs).

Several solutions for implementing publish-subscribe infrastructures have been proposed ranging from structured and deterministic [5, 8, 1] to unstructured and probabilistic [3]. In the former, participating nodes are organized in a specific structure (e.g. tree-based) which is then used to route publications to the respective subscribers. The latter uses limited subscription information and probabilistic decisions to provide high event delivery with low overhead in highly dynamic environments.

In this paper we propose a probabilistic publish-subscribe approach for MANETs which uses random walks for subscribing and publishing events in the system. Inspired by previous work [1], our solution benefits from the existence of routing information provided by a unicast routing protocol like DSR [4] to aid in the optimization of the overlay network used for publishing and subscribing. Moreover, by relying on an underlying unicast protocol, our algorithm can be executed in just a small subset of network nodes, as long as every node executes the routing protocol.

The rest of the paper is organized as follows. In Section 2 we describe the core components of the publish-subscribe approach. Preliminary experimental results are presented in Section 3. An overview of open issues regarding our approach is provided in Section 4. Finally, Section 5 presents the concluding remarks and outlines future work.

2. Publish-Subscribe Protocol

We now provide a description of the publish-subscribe protocol. Each participant in the system can be a publisher, subscriber, forwarder or any combination of the three, and maintains both an active view *AView* and a passive view *PView*. The purpose of the *AView* is to store the identifiers of participants to which the current participant can send/forward messages. The *PView* works as a backup and holds identifiers of the remaining participants for which routing information is available (from DSR).

In order to allow only a subset of network nodes to participate in the publish-subscribe system we build an overlay network on top of DSR that uses the routing table information to optimize the links in the network graph. This can be accomplished by keeping in the *PView* every participant with routing information and performing exchanges with the *AView* based in some criteria.

To cope with the highly dynamic environment of MANETs and to address scalability concerns, our algorithm uses a probabilistic approach based on random walks [6] to forward published events to the interested subscribers.

The protocol can be divided into three main components: overlay network construction and maintenance, event subscription and event publication.

2.1. Overlay Network Construction and Maintenance

The overlay network is the cornerstone of our proposal. Its topology will determine the efficiency of the random walk based publishing and subscribing activities. The path of an unbiased random walk is usually determined by the *in-degree* of participants in the network, i.e. the number

of incomming connections of each participant. In simplistic terms, participants with higher *in-degrees* will be visited by random walks with greater probability than participants with lower *in-degrees*.

Also important to the overall performance of the system is the *stretch*, i.e. the average ratio of the shortest path latency in the overlay network to the shortest path latency in the physical network. The goal here is to minimize the *stretch* without compromising the topological properties of the overlay.

2.2. Event Subscription

When a given participant wants to subscribe a certain type of event it initiates several random walks that contain the event type. Participants visited by the random walks store an association between the event type and the subscribing participant. These associations are later used to forward the events to the respective subscribers. Note that, because our approach uses DSR as the underlying unicast primitive no routing information is kept by the publish-subscribe protocol. Only the address of the respective participant is stored.

2.3. Event Publishing

When publishing an event the publisher also initiates multiple random walks containing the event. Each visited participant checks its association table for interested subscribers which, if found, will be sent the event.

Because subscribers also participate in the forwarding of events, if they have information about other subscribers for the same event, they will also forward it to them.

The rationale behind this approach for event publication is the fact that random walks will intersect in higher *in-degree* participants with greater probability. Therefore, we can take advantage of this emerging property in order improve the delivery ratio of publications.

3. Preliminary Experimental Results

To quickly assess the benefits of an approach based on the previous description we performed several preliminary experiments. To illustrate the rationale behind our proposal we present two of them.

Both experiments were performed in the *NS-2* network simulator.

In the first experiment 50 nodes were placed randomly in a 1000x1000 meters area, with a movement model following the random waypoint approach, at a speed of 10 m/s. Every node participates in the publish-subscribe system, but only one participant publishes and five participants subscribe. Subscriptions are performed by a variable amount

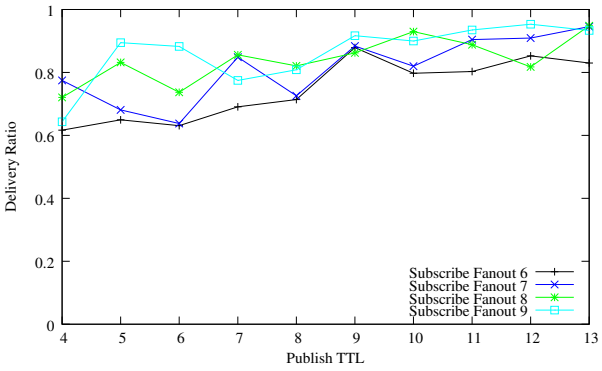


Figure 1. Delivery ratio in a scenario were publications are performed at rate of 1 per second.

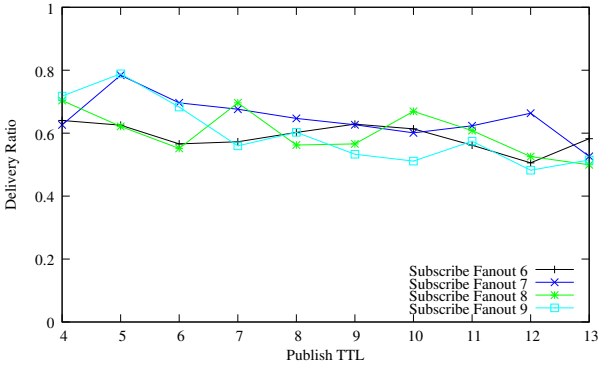


Figure 2. Delivery ratio in a scenario were publications are performed at rate of 20 per second.

(*fanout*) of random walks with a TTL value of 2. Publications are performed by two random walks with a variable TTL.

In Figures 1 and 2 we show the delivery ratio for different combinations of subscription *fanout* and publication TTL, with the publication rate being 1 event per second and 20 events per second, respectively.

As we can see in the first scenario (Figure 1) the delivery rates range from approximately 60% with a publish TTL of 4 and a subscription fanout of 6 to approximately 96% with a publish TTL of 12 and a subscription fanout of 9. In the second scenario (Figure 2) the delivery rates range from close to 50% with a publish TTL of 12 and a subscription fanout of 9 to almost 80% with a publish TTL of 5 and a subscription fanout of 9.

The performance clearly suffers from the increase in the publishing rate, which is justified by an increase in the number of messages in the underlying MANET, resulting in an

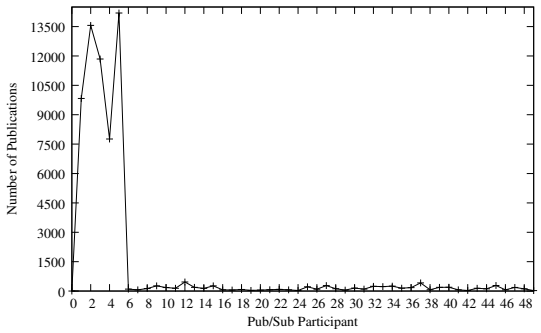


Figure 3. Distribution of publications across the participants of the publish-subscribe system.

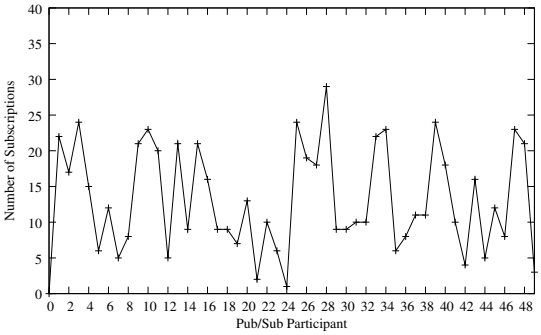


Figure 4. Distribution of subscriptions across the participants of the publish-subscribe system.

increase in the packet drop rate. These results illustrate how the combination of the number of random walks and their length influence the ability to properly deliver published events to the respective subscribers.

In the second experiment we studied the impact of random walks in the spreading of publication and subscription information. The simulation consisted of a network with 250 nodes placed in a square of 2000x2000 meters, moving at 2 m/s following a random waypoint movement model. Only a fraction of the nodes (50) were made participants of the publish-subscribe system. The simulation was performed using one participant as a publisher and five other participants as subscribers. The publish TTL was 13 with a single random walk and the subscription TTL was 9 with two random walks.

In this experiment we obtained an average delivery ratio of 86%. Figures 3 and 4 show the plots for the number of publications and subscriptions received by each participant, respectively.

Figure 3 clearly shows a significant bias in the publishing activity, with participants 1 to 5 receiving most of the publication information. However, Figure 4 shows a more distributed amount of subscription information, which may justify the average delivery ratio of only 86%. This indicates that a more biased random walk approach is required in the subscription phase to ensure a higher ratio of intersections with publication random walks.

4. Open Issues

In the previous sections we described the general behavior of each component of the approach and provided some experimental results of a preliminary implementation of the publish-subscribe system. In this section we provide a small discussion about open issues that we identified in our approach.

4.1. Overlay Network Quality

As we saw in Section 2, subscription and publication activities are performed using random walks. Depending on the desired outcome of the probabilistic dissemination of events, we may wish to influence the topology of the overlay network to promote a regular or biased topology. The former case will lead to a more fair scenario, where all participants will potentially have similar loads. The latter case will bias the *in-degrees* of participants and therefore promote more load on participants with higher *in-degrees*. As we will see later in this section, biasing random walks may be useful in some situations.

4.2. Mobility

In MANETs node mobility is usually the main source of topology changes due to sporadic or permanent disconnections, or to readjustments in routing paths. Because our approach is built on top of DSR, topology changes in the MANET are handled transparently. However, these changes may have a negative impact on the performance of the publish-subscribe system, which is built on top of the overlay. To counter this effect we need a mechanisms that adapts the overlay topology at run-time.

A way to accomplish this would be for each participant to periodically compare its views (*AView* and *PView*) regarding routing information provided by DSR and choose for its *AView* the participants that minimize the *stretch* of the overlay. This feature can, however, produce clustering or even partitioning effects in the overlay network, which would significantly degrade the system's performance.

Therefore, it is our belief that an overlay maintenance and optimization strategy must ensure some diversity in each participant's *AView* regarding participants hop-count. Although it should try to have as much closer participants

as possible it must also contain some that are further away in the underlying MANET to avoid the problems just described.

4.3. Load Balancing *vs* Delivery Ratio

As we have seen in Section 3 a high load on the publish-subscribe system may produce a decrease in the delivery ratio of the publish-subscribe system. Therefore, using overlay topologies that favour load balancing can decrease collisions and contention in the underlying MANET and improve the performance of the overall system. However, random walks tend to meet more frequently in biased network topologies, which would lead to an improved delivery ratio.

Bar-Yossef [2] proposes a method, which is based in *self-loops*, to erase the heterogeneity in node degrees. Each node adds *selfloops* (connections to itself) until all have the same degree. This way, a simple random walk would always sample nodes following an uniform distribution. Moreover, *selfloops* are "cheap" in a MANET environment, because they are done locally and do not require any radio communication. Other approaches usually use information about neighboring nodes to bias the forwarding probabilities. This however, has the disadvantage of requiring additional information to be exchanged with neighboring nodes.

A survey of biased random walks for uniform node sampling and selection can be found in [7]. It is our goal to experiment with these approaches to find the one(s) that best fit our needs.

Although regular network topologies can be useful for improving load balancing and connectivity, sometimes it seems reasonable to promote unbalance in order to improve convergence. For instance, if we increase the *in-degree* of a set of nodes, they will be more likely visited by random walks. Intuitively, this would also improve the ability of publication random walks to intersect subscription random walks, as both would visit these nodes with greater probability than the remaining.

We can have biased random walks either by building an irregular overlay network, or by tweaking the forwarding probabilities associated with the random walk. In the former case, the *contact node* may provide the means to bias the *AView* of the joining node, however either the *contact node* is always the same or all the *contact nodes* must coordinate to agree on which participants will be promoted. In the latter case neighboring participants may coordinate with each other, for instance by carefully shuffling views, to increase the *in-degree* of specific participants.

5. Conclusions and Future Work

In this paper we have proposed a probabilistic publish-subscribe approach for Mobile Ad Hoc Networks. Our proposal has several advantages over previous solutions, namely the ability to execute on only a subset of nodes as long as the remaining ones execute DSR, and also the improved scalability and adaptability because it uses only probabilistic dissemination. We presented preliminary results that provide some insight on the issues we need to address in the final version of our system. Moreover, we showed how the parametrization of the publish-subscribe system components may provide adapted behaviors that favour load balancing or improved delivery ratios.

For future work we intend to do an analysis of the different approaches biased and unbiased random walks in irregular graphs, both formally and experimentally, and develop an algorithm that makes use of the best approach to provide a scalable and efficient publish-subscribe protocol for MANETs.

References

[1] M. Avvenuti, A. Vecchio, and G. Turi. A cross-layer approach for publish/subscribe in mobile ad hoc networks. In T. Magedanz, A. Karmouch, S. Pierre, and I. S. Venieris, editors, *Mobility Aware Technologies and Applications*, volume 3744 of *Lecture Notes in Computer Science*, pages 203–214. Springer, 2005.

[2] Z. Bar-Yossef, R. Friedman, and G. Kliot. Rawms -: random walk based lightweight membership service for wireless ad hoc network. In *MobiHoc '06: Proceedings of the seventh ACM international symposium on Mobile ad hoc networking and computing*, pages 238–249, New York, NY, USA, 2006. ACM Press.

[3] P. Costa and G. P. Picco. Semi-probabilistic Content-based Publish-subscribe. In *Proceedings of the 25^{th} International Conference on Distributed Computing Systems (ICDCS05)*, pages 575–585, Columbus (OH, USA), June 2005. IEEE Computer Society Press.

[4] D. A. M. David B. Johnson and J. Broch. *Ad Hoc Networking*, chapter DSR: The Dynamic Source Routing Protocol for Multi-Hop Wireless Ad Hoc Networks, pages 139–172. Addison-Wesley, 2001.

[5] Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile environment. *Wireless Networks*, 10(6):643–652, 2004.

[6] L. Lovász. *Combinatorics, Paul Erdos is Eighty*, volume 2, chapter Random Walks on Graphs: A Survey, pages 1–46. Bolyai Society, Keszthely, Hungary, 1993.

[7] V. Vishnumurthy and P. Francis. On heterogeneous overlay construction and random node selection in unstructured p2p networks. In *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM 2006)*, pages 1–12, Barcelona, Spain, Apr. 2006. IEEE.

[8] E. Yoneki and J. Bacon. Distributed multicast grouping for publish/subscribe over mobile ad hoc networks. In *2005 IEEE Wireless Communications and Networking Conference*, volume 4, pages 2293–2299, New Orleans, LA, USA, Mar. 2005. IEEE.

Predictive Publish/Subscribe for Delay Tolerant Mobile Ad Hoc Networks*

Paolo Costa
Vrije Universiteit, Amsterdam
The Netherlands
costa@cs.vu.nl

Cecilia Mascolo
University College of London, UK
c.mascolo@cs.ucl.ac.uk

Mirco Musolesi
University College of London, UK
m.musolesi@cs.ucl.ac.uk

Gian Pietro Picco
University of Trento, Italy
picco@dit.unitn.it

Abstract

Many infrastructure-less mobile applications demand the ability to withstand long-lasting network partitions and disconnections. Solutions supporting opportunistic communication have been studied in the context of delay-tolerant mobile ad hoc networks. However, they typically assume that the sender determines the intended recipients, using a unicast or multicast address. Instead, several applications require a form of publish/subscribe, where it is the receiver that specifies the messages it is interested in; the sender simply injects messages into the network, which routes them based on their content. Although publish/subscribe protocols and systems exist for fixed and mobile networks, to the best of our knowledge no proposal addressed highly partitioned and intermittently connected MANETs.

In this paper we report about our experience with SOCIAL-CAST*, a routing framework for publish-subscribe that exploits predictions based on metrics of social interaction (e.g., patterns of movements among communities) to identify the best information carriers. The protocol takes into account contextual information (e.g., connectivity changes) to produce estimates about the network evolution. These estimates are used to forward messages not only to the intended receivers, but also towards good message carriers, i.e., hosts with a high probability to deliver messages across partitions. We highlight the principles underlying our protocol, illustrate its operation, and evaluate its performance using a mobility model validated with real mobility traces.*

1 Introduction

Imagine an emergency situation (e.g., a fire in a road tunnel) where wireless communication supports the activities of rescue teams. Different team members have different mobility characteristics. For instance, medical personnel are likely to remain stationary for a long time, curing the injured people. Firemen are likely to move more frequently, although most likely around their assigned area. Supervisors are likely to stay out of immediate emergency areas, albeit frequently moving between critical zones. Therefore, team members form highly dynamic mobile ad hoc networks (MANETs) that are likely to be intermittently connected and frequently partitioned, due to the relative movement of members and to environmental factors (e.g., different connectivity inside and outside the tunnel). Application messages cannot be delivered along the whole end-to-end path in one shot, since the links forming this path are not simultaneously active. Instead, messages may be stored during partitions and forwarded opportunistically when connectivity is restored. For instance, information about the fact that a path is now safe may be present among firemen at one end of the tunnel, and it maybe of interest to a medical personnel. However, the two may be temporary disconnected; communication can occur only by relying on hosts (e.g., supervisors) buffering messages while in the firemen range, and delivering them opportunistically when in the range of medical personnel.

The scenario we described can be defined as a *delay-tolerant* [9] MANET, a dynamic infrastructure-less network where groups of hosts can be temporarily or permanently be partitioned. Applications of this kind of networks cover a broad spectrum including support for rural networks, interplanetary communications, wildlife monitoring, and infomobility. Research in the field of delay-tolerant network has focused on traditional communication paradigms where the sender knows *a priori* the intended receivers, and messages are routed based on their (unicast or multicast) recipient address [15]. Nevertheless, many of the applications above—including the example at the beginning of this paper—call for a paradigm where the sender does not know who should receive a message, and is, instead, up to the receiver to spec-

*An extended version of this paper is currently under submission and available as techinical report [7].

ify the messages of interest, based on their content.

This form of communication demands for *publish-subscribe* , where messages injected in the network by senders are routed by the network towards the interested hosts. Several proposals and system exists for publish-subscribe in fixed networks (e.g., [4, 5, 14]) and, recently, some have been proposed also for MANETs (e.g., [2, 8, 12]). However, to the best of our knowledge, no proposal has yet addressed the specific characteristics of highly partitioned and intermittently connected MANETs.

In this paper we present a predictive publish-subscribe protocol called SOCIALCAST, designed to deal with the aforementioned delay tolerant mobile ad hoc scenarios. In a nutshell, SOCIALCAST complements the information about the receivers' interests with information about the changes in the *context* observed by nearby hosts. Our framework is general enough to encompass a broad definition of context, e.g., including a host's residual energy, physical location, or application-specific data. Kalman filter forecasting techniques [3] are used to predict the future evolution of these parameters based on previous observations. These predictions are used to estimate which hosts are potentially good message carriers, i.e., may enable indirect connectivity by moving into partitions containing subscribers. In some situations, the forwarding host may actually be a good carrier as well, and, therefore, buffer the message in the hope of forwarding it opportunistically later.

2 System Model and Assumptions

We assume a network composed of N hosts. For simplicity of treatment we assume they all have the same capabilities, in particular to store messages in a buffer of maximum size β. Hosts are mobile and interconnected by wireless links. The mobility of a host is determined by the user carrying it.

A user, and therefore a host, may act as an information *publisher* or *subscriber*[1]. Publishers and subscribers are in general not aware of each other. A host subscription identifies the host's interest (e.g., "Rugby" or "Computer Science"). We assume that each user in the system has at least one interest. When a message is published (e.g., "Six Nations Results"), it is tagged with the related interest. The goal of our protocol is to deliver the message to the hosts with at least one interest matching the one in the message. As such, delivery is driven by the message content. In this work we base matching on interests specified as message topics, but we conjecture that extensions allowing for more sophisticated and direct matching against the message content can be easily integrated in our approach.

Key to this work is the assumption that users with common interests tend to meet with each other more often than with other users. This can be observed in practice in our everyday life. Examples are people interested in information concerning the department where they work, or friends sharing the same sport interest. In other words, we assume that the mobility of users is driven by their social behaviour that, in turn, is determined by their common interests.

Apart from the aforementioned social behaviour, hosts can move with arbitrary (not necessarily random) directions and speeds, and in doing so they may cause an arbitrary number of network partitions. Furthermore, for what concerns communication we rely solely on the basic ability of a host to communicate within its 1-hop neighbourhood, by broadcasting a message to all the neighbours or unicasting it to a specified one.

[1] A host can be, at the same time, a publisher and a subscriber.

3 Routing in SOCIALCAST

In this section, we describe the main characteristics of our routing protocol. This relies on the notion of *utility* for the selection of message carriers in order to enable store-and-forward communication. The utility of a host h with respect to interest i represents how good of a carrier h is for messages matching i. The utility values in SOCIALCAST are linked to movement patterns and colocation with other hosts: as the basic assumption is that hosts which have the same interests spend time co-located, the SOCIALCAST routing aims at exploiting as carrier for messages hosts which have been co-located often with the interested subscribers. The calculation of utilities are described in detail in the next section.

Routing in SOCIALCAST consists of three phases: interest dissemination, carrier selection, and message forwarding. The distinction in phases is only for illustration purposes, as in practice each phase is executed one after the other. The whole sequence is repeated periodically after T units, without requiring synchronisation across hosts.

During *Interest Dissemination*, each host broadcasts the list of its interests to its 1-hop neighbours, along with the corresponding list of utility values. These are first locally re-computed based on the current host context before dissemination. This information is stored in the routing tables of the neighbours, and is key in determining message forwarding decisions. In this phase, the identifiers of the last λ messages received are also piggybacked on the utility message.

During *Carrier Selection*, the utility of the local host, U_i, is recomputed for all interests i. This utility U_i is compared, for each interest i, against the highest among those communicated by neighbours, say $U_{h,i}$ as reported by a neighbour h. If $U_i < U_{h,i} + \epsilon$, this means that, for interest i, h is a better carrier than the local host. ϵ is an hysteresis threshold which forbids that the message is bounced back and forward between hosts with similar fluctuating utilities. Otherwise,

the local host is still the best carrier for messages tagged with i.

During *Message Dissemination*, the content of the buffer is re-evaluated against the new subscriptions and utilities, and messages are forwarded to the interested hosts and/or the best carrier. A copy of messages matching an interest i is immediately sent to all neighbours whose subscriptions contain i. Note how this ensures that nearby interested hosts receive messages, but does *not* imply that these also become a carrier for messages. In other words, messages are delivered to the application but *not* inserted in the hosts' buffer. Indeed, carrier role (and buffer insertion) are determined by the outcome of the previous phase. If the local host is still the best carrier, no action needs be taken. Otherwise, *all* messages tagged with i are removed from the local host's buffer. and sent to h, the best carrier, where they are inserted in its buffer. An issue arises if h is also a subscriber for i. In this case, the matching messages can be properly flagged to inform the receiving carrier h that they must be inserted in its buffer instead of being simply delivered to the application.

To avoid unnecessary traffic, a message is forwarded only if the recipient has not previously received that message. This can be easily verified by checking the list of the last λ messages piggybacked during the dissemination phase. Moreover, to prevent messages from remaining forever in the system, we rely on a time-to-live (TTL) based on hop counts. Clearly, other solutions are also possible. For instance, in some applications it could be useful to have the publisher explicitly specify an expiration time, (e.g., a concert advertisement is useful only before the time it starts).

Based on the protocol we described thus far, *Message Publishing* becomes essentially the insertion of a message into the local buffer. Indeed, our routing protocol works based on whatever the content of the buffer is, regardless of how it got inserted in it. Therefore, a message inserted by a publish operation will be forwarded to the interested subscribers as well as "moved" to a better carrier, if and when encountered. To ensure high delivery, a publish operation actually inserts γ copies of the message. Each copy is routed independently, i.e., whenever a better carrier is encountered only *one* copy is removed from the local buffer and sent to the new carrier, to ensure that the copies are spread over time and space across the system. Note that the publisher is the only host that duplicates messages, and does so only at publish time. Therefore, at any time the network contains at most γ copies of the message.

4 Computing Utilities from Context

Many parameters determine if a node is a good message carrier. A host with a high *change degree of connectivity* frequently changes its neighbor set (e.g., because it is moving, or is stable in a very dynamic area), and enjoys more forwarding options. The probability of *subscriber co-location* is the likelihood of meeting another host subscribed to interest i, thus enabling direct delivery of matching messages. A host's *residual energy* indicates if it is going to stay alive long enough to meet other hosts. Finally, the *free buffer space* tells if the host can carry the message altogether.

Knowledge about the current values of these context attributes is helpful, but only to a limited extent. What really matters are the values the attributes are likely to assume in the future. We compute these *predicted* values using techniques based on Kalman filters [10]. These techniques do not require the storage of the entire past history of the system and are computationally lightweight, making them suitable for a resource-scarce mobile setting.

We cannot repeat here the mathematical details involved: a comprehensive presentation of these techniques is found in [3]. However, it is fundamental to say how a host computes the *input values* to the Kalman filter, i.e., the value of the utility at time t, for which the filter computes the predicted value at time $t + T$. Hereafter, we focus only on change degree of connectivity and probability of subscriber co-location, because they are the attributes most relevant to our work. However, the framework is general and open to inclusion of any other context attribute.

The change degree of connectivity of a host h is

$$U_{cdc_h}(t) = \frac{|n(t-T) \cup n(t)| - |n(t-T) \cap n(t)|}{|n(t-T) \cup n(t)|} \quad (1)$$

where $n(t)$ is h's neighbor set at time t. The formula yields the number of hosts that became neighbors or disappeared in the time interval $[t-T, t]$, normalized by the total number of hosts met in the same time interval. A high value means that h recently changed a lot of its neighbors.

The co-location of h with a subscriber for interest i is

$$U_{col_{h,i}}(t) = \begin{cases} 1 & \text{if } h \text{ is co-located with a subscriber for } i; \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

A value of 1 means that h has been co-located with subscribers for i at time t.

These values are fed into Kalman filter predictors, which yield the predictions $\widehat{U}_{cdc_h}$ and $\widehat{U}_{col_{h,i}}$ of these utilities at time $t + T$. These are then composed into a single utility value using results from multi-criteria decision theory [11], as

$$U^{rec}_{h,i} = w_{cdc_h}\widehat{U}_{cdc_h} + w_{col_{h,i}}\widehat{U}_{col_{h,i}} \quad (3)$$

which represents how good of a carrier h is for messages matching i. The weights w denote the relative importance of each attribute and their value depends on the application scenario. Instead, next we show how we use the utility value above for routing messages.

Parameters	Default value
Simulation area	$4\ km \times 4\ km$
Number of hosts	100
Hosts speed	$[1-6]\ m/s$
Transmission range	$250\ m$
Percentage of publishers	50 %
Percentage of subscribers	50 %
Publishing interval	$60\ s$
Number of interests	10
Simulation duration	8 hours
w_{cdc}	0.25
w_{col}	0.75
Buffer size (β)	∞
Number of copies (γ)	3
Retransmission interval (T)	20 s
Hysteresis threshold (ϵ)	0.2

Figure 1. Simulation parameters

5 Evaluation

In this section we report about the evaluation of SocialCast based on a social mobility model. We evaluated the performance of our protocol using OMNeT++ [1], an open-source discrete event simulator written in C++.

Mobility Model. Traditionally, mobile wireless networks simulators assume a mobility model in which hosts move randomly in the space. This, however, does not suit our needs: since SOCIALCAST exploits prediction of co-location and movement, the use of a purely random mobility model would prevent an effective analysis of the protocol. To this end, we adopted the Community based mobility model [13], characterised by mobility patterns founded on social networks and validated against real traces provided by Intel Research [6].

Default Parameters. In real life, people sharing similar interests happen to be co-located more frequently among each others than with others. This property is crucial to our protocol as it can be exploited to perform accurate predictions over future movements of hosts. To reproduce this behaviour in our simulator, we map one interest to each community of the synthetic social network described in the previous section, such that hosts have more probability to be co-located with other hosts having the same interests. To avoid any bias, we assumed that a host can subscribe to at most one interest. Finally, publishers are uniformly chosen among all the hosts in the simulation space. A summary of all the simulation parameters is presented in Figure 1.

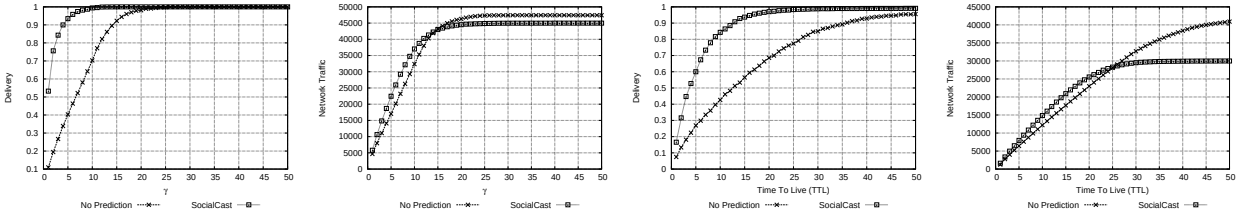
Simulation Results. We now present the results of our simulations of SOCIALCAST [2]. In all our experiments, we mainly concentrate on message delivery and network traffic. The former accounts for protocol *effectiveness* and is defined as the ratio between the actual number of messages delivered to the interested subscribers and the ideal one. The network traffic, instead, is constituted by the number of forwarded messages and measures the *efficiency* of the protocol.

To provide more insights, we compare SOCIALCAST with a variant in which prediction is not used and where the next carrier is selected on a random basis.

Number of Replicas. The first parameter we studied is the number γ of replicas in the system. This is a key parameter, because it has a large impact on the network traffic. Results in Figure 2(a) show that, through prediction, SOCIALCAST is able to achieve high message delivery with less replicas than the ones needed if prediction is not used. Indeed, 5 replicas are sufficient for SOCIALCAST to reach more than 90% of subscribers while without prediction three times that number of replocase is needed to obtain similar performance. Notably, although delivery is greatly improved in SOCIALCAST (e.g., with $\gamma = 5$ prediction boosts delivery from 40% up to 93%), the network traffic is not increased (see Figure 2(b)). The reason stems from the fact that network traffic strongly depends on the number of replicas. Therefore, since both SOCIALCAST and its variant share the same γ, the traffics are similar. However, leveraging off predictions, SOCIALCAST can select better carriers which enable reaching more subscribers, thus achieving better performance without increasing the traffic.

Time To Live (TTL). The Time To Live of a message (TTL) represents the dual parameter of γ, as they provide complementary information. Indeed, γ controls how many instances of the same message are around, while TTL defines for how far, in terms of hops, a message will be around. Clearly, given a fixed γ, by increasing the number of possible hops, there are more chances to reach other subscribers. Unfortunately, this comes at the price of a higher overhead because the message will stay around longer. Figures 2(c) and 2(d) show the performance of our protocol against different values of TTL. As expected, prediction enables decreasing the TTL because a message is forwarded only when needed, i.e., when a better carrier or a subscriber is encountered. Conversely, without prediction, messages are forwarded in a random fashion and hence more hops are needed to successfully contact the subscribers. This is confirmed in Figure 2(c): 15 hops per message are enough to SOCIALCAST to reach more the 90% of subscribers while the variant without prediction requires at least 35 hops per message. Clearly, the two traffic curves show similar trends because the TTL (as well as γ) directly influences the traffic. Notably, however, the network traffic generated by SOCIALCAST saturates for $TTL > 25$. Indeed, when all the best carriers and the subscribers have been reached (i.e., the delivery hits 100%), the messages are no further replicated. This happens for a value of TTL equal to 25. The traffic generated by the variant without prediction, instead, in-

[2]Due to space constraints, we cannot report the whole set of experiments we carried out. The interested reader can refer to [7].



(a) Delivery (number of replicas). (b) Overhead (number of replicas). (c) Delivery (time-to-live). (d) Overhead (time-to-live).

Figure 2. Delivery and overhead against number of replicas and time-to-live.

creases linearly with the TTL because it continuously forwards messages, also when not needed. This is a result of paramount importance because it demonstrates that our protocol does not waste resources generating additional traffic when not needed.

All the aforementioned results confirm the suitability of SOCIALCAST for the scenarios we target and demonstrate the improvements introduced by our prediction mechanisms. Indeed, thanks to prediction, SOCIALCAST is able to perform more accurate selection and to provide a more efficient usage of resources, both in terms of network traffic and memory. In addition, the ability to predict over the colocation as well as the host mobility allows for maintaining a very high and steady event delivery with a reasonably low traffic and latency.

6 Conclusions and Future Work

This paper presents a novel approach to publish-subscribe in delay-tolerant mobile ad hoc networks, based on an informed selection of the best carriers for messages matching content-based interests. This selection is made by taking into account predictions about contextual parameters (e.g., mobility patterns and connectivity), based on previous observations. We have evaluated our approach in realistic scenarios with disconnections to demonstrate the advantages of the prediction and store and forward strategies in terms of message delivery and overhead. Future work will address the inclusion of additional contextual information in our predictions, as encompassed by the general framework we described in Section 4. We also plan to port our work onto the architecture defined by the DTN Research Group [9] and to perform more evaluation on a real test-bed.

References

[1] OMNeT++ Web page. `www.omnetpp.org`.

[2] R. Baldoni, R. Beraldi, G. Cugola, M. Migliavacca, and L. Querzoni. Content-based routing in highly dynamic mobile ad hoc networks. *Journal of Pervasive Computing and Communication*, 1(4), 2005.

[3] P. J. Brockwell and R. A. Davis. *Introduction to Time Series and Forecasting*. Springer, 1996.

[4] F. Cao and J. P. Singh. Efficient event routing in content-based publish/subscribe service network. In *Proceedings of the 23^{th} IEEE Conference on Computer Communications (INFOCOM 2004)*, 2004.

[5] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Systems*, 19(3):332–383, 2001.

[6] A. Chaintreau, P. Hui, J. Crowcroft, C. Diot, R. Gass, and J. Scott. Impact of Human Mobility on the Design of Opportunistic Forwarding Algorithms. In *Proceedings of INFOCOM'06*, April 2006.

[7] P. Costa, C. Mascolo, M. Musolesi, and G. P. Picco. Socially-aware routing for publish/subscribe in delay-tolerant mobile ad hoc networks. Technical report, University College of London, 2007. Submitted for publication.

[8] P. Costa and G. P. Picco. Semi-probabilistic Content-Based Publish-Subscribe. In *Proceedings of the 25^{th} IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, Columbus (Ohio, USA), June 2005.

[9] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of SIGCOMM'03*, August 2003.

[10] R. E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME Journal of Basic Engineering*, March 1960.

[11] R. Keeney and H. Raiffa. *Decisions with Multiple Objectives: Preference and Value Tradeoffs*. Wiley, 1976.

[12] R. Meier and V. Cahill. STEAM: Event-Based Middleware for Wireless Ad Hoc Networks. In *Proc. of the 1^{st} Int. Wrkshp on Distributed Event-Based Systems*, July 2002.

[13] M. Musolesi and C. Mascolo. Designing Mobility Models based on Social Network Theory. *ACM SIGMOBILE Mobile Computing and Communications Review*. To appear.

[14] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An information flow based approach to message brokering. In *International Symposium on Software Reliability Engineering (ISSRE'98)*, 1998.

[15] Z. Zhang. Routing in Intermittently Connected Mobile Ad Hoc Networks and Delay Tolerant Networks: Overview and Challenges. *IEEE Communications Surveys and Tutorials*, January 2006.

Towards a Peer-to-peer Middleware for Context Provisioning in Spontaneous Networks

Tuan Dung Nguyen, Siegfried Rouvrais
GET / ENST Bretagne
Technopôle de Brest Iroise, 29238 Brest, France
{td.nguyen,siegfried.rouvrais}@enst-bretagne.fr

Abstract

Context-awareness has been widely considered important for applications to provide adaptable services to changes in dynamic environments. Classical approaches have mainly focused on gathering and representing internal contextual information from integrated sensors. Recent work has shown that useful external contexts can also be acquired from the surrounding environment in a distributed manner. However, the dynamism and heterogeneity of spontaneous networks raise new challenges for the provisioning of such information. This paper describes our ongoing work on a peer-to-peer middleware for abstracting contexts as logical peers in independent overlay networks. Context data retrieval then becomes transparent to applications and other middleware services. The proposed middleware will support efficient construction and maintenance of these overlays and offer common interfaces to applications. We discuss in detail its architecture and the open issues to be resolved.

1 Introduction

The increasing popularity of wireless computing devices has resulted in the spontaneous networking paradigm where a large number of nodes (e.g. PDAs, smart phones) collaborate to provide services without neither predefined infrastructures nor any manual interventions. Such environments are characterized by node's heterogeneity, potential mobility, unpredicted disconnection and limited resources. Systems and applications should be able to provide appropriate services based on user's current context (e.g. location, time, social situation).

Dey [3] defined context as any information that can be used to characterize the situation of an entity (e.g. person, place, and object). This information helps applications to dynamically adapt their behavior according to their requirements and to changes in the environment. Contexts can be either internal (e.g. user's profile, device's battery level) or external (e.g. nearby resources, temperature). They can also be static (e.g. user agenda) or dynamic (e.g. network bandwidth). For instance, a car may need various contextual information such as weather, nearby petrol stations, road status.

Context provisioning mainly consists of contextual information acquisition, sharing and interpretation. Internal contexts are often captured by inherently integrated physical and/or logical sensors. External contexts are acquired from the surrounding environment either in a centralized or distributed manner. In the former approach, specific entities are used for context storage and lookup while in the latter case, context data is provided by any entities in proximity of an ad hoc network [13]. The centralized approach is not relevant to spontaneous networks due to the lack of infrastructure and the frequent mobility.

Overlay networks are fundamental building blocks of peer-to-peer systems where nodes are connected by logical links on top of underlying physical networks. Their self-organization and sharing capability is a good support for dynamic environments.

Our work is highly motivated by the following requirements for context provisioning in spontaneous networks:

- **Context representation**. Various contexts can be gathered from different sources. An adequate representation should respect the separation of concerns design principle for context management and enable transparent context provisioning to upper applications.
- **Context sharing**. Resource-constrained and heterogeneous devices have only a limited number of sensors for capturing contextual information. Peer-to-peer context sharing allows nodes to obtain needed contexts not only from their integrated sensors but also from their neighbors.

The contribution of this paper is a peer-to-peer middle-

ware for managing different contexts on independent overlays. This framework allows resources-constrained nodes to maintain only currently needed contexts according to their available resources and their own non-functional requirements.

The rest of this paper is organized as follows. Section 2 presents our middleware architecture for multiple context overlays. Next, Section 3 reviews related work and finally, Section 4 concludes the paper by discussing some open questions.

2 Dynamic overlays for context provisioning

A physical device can have several integrated sensors for capturing contexts. However, as illustrated in Figure 1, a device can have a number of context peers (CxtPeer) corresponding to its needed contexts. According to application's needs and available resources, logical peers can be created and destroyed on-the-fly. Peers are connected in independent self-organized overlays. The context data stored in each peer can be provided by integrated sensors or by querying other peers. The content can be cached in a peer during a defined period for later lookup operations.

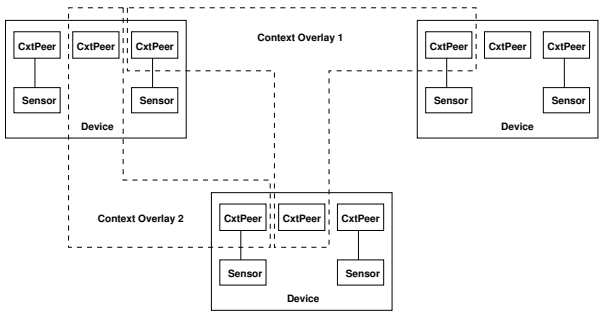


Figure 1. Dynamic overlays of context peers

Figure 2 presents our middleware architecture. The main components are described as follows.

- **Context Manager**. This component is responsible for dynamic context peer creation and destruction. New context peers can be created from a proposed specification language. It also provides applications a common interface to query context data. The context query processing can be synchronous or asynchronous, i.e. applications can submit a context query and receive the result after some acceptable delay.

- **Context Overlay Manager**. This component is responsible for overlay maintenance due to node's mobility in the proximity. Here, CxtPeer components are grouped based on its context semantic. The maintenance can also be triggered on the creation/destruction of a new context peer.

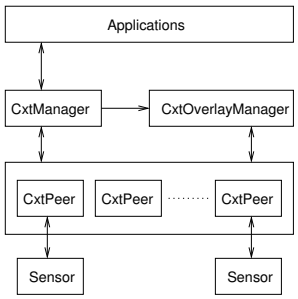


Figure 2. Middleware architecture

The software model of our proposed system is depicted in Figure 3. CxtPeer maintains list of neighbors representing a local view of the corresponding overlay. These neighbor lists are dynamically maintained by CxtOverlayManager. A CxtPeer has a Context data which can be associated or not with an integrated sensor.

When a context lookup query arrives at CxtManager, it is forwarded to the corresponding CxtPeer. Then the context data can be retrieved from a cached content, an integrated sensor or by querying other peers. These operations are carried out in the middleware layer and transparent to applications. By this way, some context data can be retrieved thanks to node's mobility even in partitioned networks.

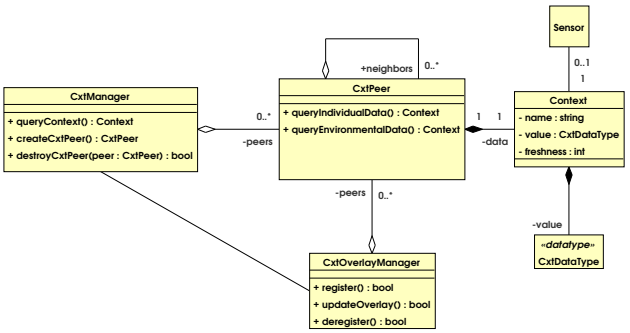


Figure 3. Context provisioning model

Fractal [4] is a modular, extensible and language-independent component model to design, implement, deploy and reconfigure middleware systems and applications. Our middleware model based on the Fractal is depicted in Figure 4. CxtPeer is a primitive component representing a context. These components are managed in a composite component called CxtManager. Its content part is composed

of a finite number of CxtPeer components. Its controller interfaces (namely CC, LC) permit the creation and destruction of primitives peers on-the-fly. A CxtPeer has a server interface providing the corresponding contextual information. Its client interface is connected to the client interface of its surrounding component CxtManager. Different context overlays are maintained dynamically by distributed binding between server and client interfaces of CxtManager components (e.g. using Fractal RMI [4]). These maintenance operations are carried out by CxtOverlayManager component.

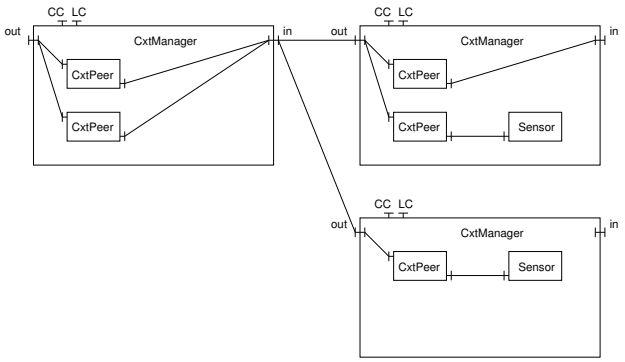


Figure 4. Fractal component-based model

3 Related work

The tuple-space model was first presented with Linda [5] and then adapted to mobility with LIME [12], TinyLIME [2] or ErgoSpaces [10]. These efforts have proved the interest of this data-sharing model to dynamic environments. However, existing works lack the capacity to efficiently manage different context sources according to various application's needs.

Using peer-to-peer overlays for context lookup was first presented in [9]. The authors proposed ContextBus, an architecture in which context producers are grouped based on the semantic of their data. An improved version namely Semantic Context Space (SCS), was also proposed to reduce the maintenance cost when the number of semantic groups increases using a one-dimensional ring space [7, 8]. Unlike this work, we propose a more abstract concept because any devices can have a context peer even if they do not directly produce context data. Moreover, we support dynamic overlay management and effective mechanisms for overlay maintenance.

Research work has recently put great attention on the coexistence of multiple overlays. Gridkit [6] is a component framework for pluggable overlay networks even at runtime. Behnel et al. [1] combined Gridkit with an overlay design modeling framework to enable the generation of specific code from platform-independent design of overlay networks. Maniymaran et al. [11] presented a mechanism to reduce the multiple overlays maintenance overhead without sacrificing the performance. The authors proposed that the maintenance of one overlay can be leveraged to partly maintain another overlay with no extra cost. These approaches form a good base for the overlay construction and maintenance in our framework. However, the coexistence of multiple context overlays needs further investigation.

4 Discussion and open questions

We have presented a peer-to-peer middleware architecture for context provisioning in spontaneous networks. The proposed system offers a means for context abstraction and sharing in such dynamic environments. We plan to evaluate our proposition with a prototype working on real PDAs. By comparing it to existing tuple-space based works, we will demonstrate its capacity to answer to its design requirements and its overall overhead is acceptable. We have also identified the three following issues to be investigated.

1. **Context peer specification and deployment**. We are investigating a specification language and a component framework for the dynamic management of overlays. The language will allow applications to specify new context peers based on their current requirements. The component-based framework enables on-demand deployment of different context peers. To avoid ambiguity among different contexts, ontology-based approaches can be investigated in future work.

2. **Context overlay construction and maintenance**. As context peers are connected in unstructured overlays, the maintenance operations involve two phases: a peer sampling phase providing candidates for the neighbor list establishment based on the semantic of overlays. We plan to exploit epidemic-style approaches [14] for peer sampling service in spontaneous networks. Then the coexistence of multiple overlays can be leveraged to improve the maintenance efficiency by exploiting the common peer sampling service and the semantic relations among overlays, i.e. binding of subcomponents in a common composite component. The maintenance principle consists of updating neighbors in each overlay through dynamic interface binding of distrbuted components and assuring data consistency among context peers.

3. **Context overlay lookup and composition**. Context provisioning to applications is realized through lookup operations in overlays. Therefore, we need effective lookup mechanisms to improve system performance.

Moreover, a query for higher-level contexts may concern several context peers which results in the composition of multiple overlays. So far, few work has put attention on this issue and further investigation is needed.

References

[1] S. Behnel, A. Buchmann, P. Grace, B. Porter, and G. Coulson. A Specification-to-Deployment Architecture for Overlay Networks. In *Proc. 8th OTM Int. Symp. on Distributed Objects and Applications (DOA'06)*, pages 1522–1540, Montpellier, France, Oct. 2006. Springer-Verlag.

[2] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. L. Murphy, and G. P. Picco. Mobile Data Collection in Sensor Networks: The TinyLime Middleware. *Pervasive and Mobile Computing*, 4(1):446–469, Dec. 2005.

[3] A. K. Dey. Understanding and Using Context. *Personal Ubiquitous Computing*, 5(1):4–7, 2001.

[4] Fractal. `http://fractal.objectweb.org/`, 2006.

[5] D. Gelernter. Generative Communication in Linda. *ACM Computing Survey*, 7(1):80–112, Jan. 1985.

[6] P. Grace, G. Coulson, G. Blair, L. Mathy, W. K. Yeung, W. Cai, D. Duce, and C. Cooper. GRIDKIT: Pluggable Overlay Networks for Grid Computing. In *Proc. 6th OTM Int. Symp. on Distributed Objects and Applications (DOA'04)*, pages 1463–1481, Agia Napa, Cyprus, Oct. 2004. Springer-Verlag.

[7] T. Gu, H. K. Pung, and D. Zhang. A Peer-to-Peer Overlay for Context Information Search. In *Proc. 14th IEEE Int. Conf. on Computer Communications and Networks (ICCCN'05)*, San Diego, CA, USA, Oct. 2005. IEEE CS Press.

[8] T. Gu, H. K. Pung, and D. Zhang. A P2P Context Lookup Service for Multiple Smart Spaces. In *Proc. 4th ACM/USENIX Int. Conf. on Mobile Systems, Applications, and Services (MobiSys'06)*, Uppsala, Sweden, June 2006. ACM Press. Poster paper.

[9] T. Gu, E. Tan, H. K. Pung, and D. Zhang. A Peer-to-Peer Architecture for Context Lookup. In *Proc. 2nd Int. Conf. on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous'05)*, San Diego, CA, USA, July 2005. IEEE CS Press.

[10] C. Julien and G.-C. Roman. Egospaces: Facilitating rapid development of context-aware mobile applications. *IEEE Transaction on Software Engineering*, 32(5):281–298, May 2006.

[11] B. Maniymaran, M. Bertier, and A.-M. Kermarrec. Build One, Get One Free: Leveraging the Coexistence of Multiple P2P Overlay Networks. In *Proc. 27th IEEE Int. Conf. on Distributed Computing Systems (ICDCS'07)*, Toronto, Canada, June 2007. IEEE CS Press.

[12] A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: A Middleware for Physical and Logical Mobility. In *Proc. 21st IEEE Int. Conf. on Distributed Computing Systems(ICDCS'01)*, pages 524–533, Arizona, USA, Apr. 2001. IEEE CS Press.

[13] O. Riva. Contory: A Middleware for the Provisioning of Context Information on Smart Phones. In *Proc. 7th ACM/IFIP/USENIX Int. Middleware Conference (Middleware'06)*, pages 219–239, Melbourne, Australia, Dec. 2006. Springer-Verlag.

[14] S. Voulgaris and M. van Steen. Epidemic-style Management of Semantic Overlays for Content-Based Searching. In *Proc. 11st European Conf. on Parallel and Distributed Computing (Euro-Par'05)*, Lisboa, Portugal, Aug. 2005. Springer-Verlag.

Semantic Middleware for Designing Collaborative Applications in Mobile Environment

Lamia Benmouffok
Université Pierre et Marie Curie
Laboratoire d'Informatique de Paris 6
104, avenue du Président Kennedy
75016 Paris - France
Email: lamia.benmouffok@lip6.fr

Jean-Michel Busca
INRIA - Rocquencourt
Laboratoire d'Informatique de Paris 6
104, avenue du Président Kennedy
75016 Paris - France
Email: jean-michel.busca@inria.fr

Marc Shapiro
INRIA - Rocquencourt
Laboratoire d'Informatique de Paris 6
104, avenue du Président Kennedy
75016 Paris - France
Email: marc.shapiro@acm.org

***Abstract*— The Telex middleware facilitates the design of collaborative applications in a mobile environment. It provides optimistic replication, tentative execution and disconnected work. It solves conflicts based on semantic information provided by applications. We study in particular a Shared Calendar (SC) application, whereby mobile users can create and manage meetings in a collection of shared calendars. The application provides Telex with objects representing (1) meeting creation and modification operations (actions), (2) dependence or conflict information between actions (constraints). When a conflict occurs, Telex proposes solutions to users.**

The advantage of this approach is a clean separation of concerns. The SC application writer concentrates on application logic, whereas Telex takes care of replication, consistency, conflicts, and commitment across all applications.

I. Introduction

Designing collaborative applications raises the key problem of ensuring the consistency of shared mutable data. This problem is even more difficult in a mobile environment due to its decentralized nature and to the volatility of participants.

The Telex middleware facilitates the design of collaborative applications by taking care of complex application-independent aspects, such as replication, conflict detection and repair, and ensuring eventual commitment. It supports optimistic replication [1], which decouples data access from network access. Telex allows an application to access a local replica without synchronizing with peer sites. The application makes progress, executing uncommitted operations, even while peers are disconnected. Telex propagates updates lazily and ensures consistency by a global *a posteriori* agreement on the set and order of operations. Local execution is tentative; due to conflicts, some operations may roll back later.

Unlike previous optimistic replication systems, Telex takes the semantic of the collaboration into account, building on the Action Constraint Formalism (ACF) [2]. A Telex application represents its shared data as a set of actions (representing application operations submitted by users), and a set of constraints between these actions, expressing their concurrency semantics. Telex uses this semantic information to accuraty detect conflicts and to propose solutions.

We designed a Shared Calendar (SC) application to demonstrate how Telex facilitates the design of collaborative applications in mobile environment. A SC application aims to help people organizing their agenda in a collaborative way. It allows people to create and manage private events as well as group meetings, scheduled on a collection of online calendars. The design of SC application illustrates the benefit of using Telex middleware; it also illustrates some limitations of Telex.

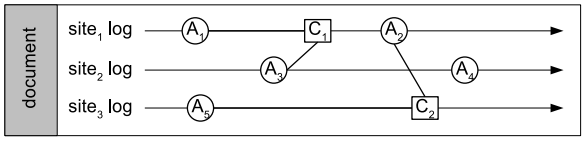


Fig. 1. Shared Document Representation

The remainder of this paper is organized as follows. Section II briefly describes the ACF. Section III presents the architecture of Telex and its operation. Section IV describes the Shared Calendar application built on Telex. Section V concludes.

II. Action Constraint Formalism

In ACF, an *action* represents an application operation and a *constraint* defines a scheduling invariant between two actions. The ACF defines three elementary constraints expressing commutativity, order and dependency relations. `(A non-commuting B)` states that executing A before B does not yield the same result as executing B before A. `(A not-after B)` indicates that A must not execute after B. `(A enables B)` means that B can execute if and only if A also executes.

Elementary constraints can be combined to express richer semantic relations, encompassing data semantics, application semantics and user intents. Thus, the cycle `((A not-after B) and (B not-after A))` states that A and B are *antagonistic*, i.e. an execution cannot contain both actions. `((A not-after B) and (A enables B))` expresses the fact that B *causally depends* on A. The cycle `((A enables B) and (B enables A))` indicates that A and B must be executed *atomically*.

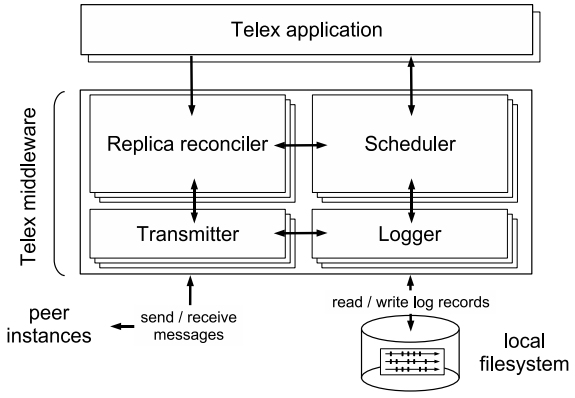


Fig. 2. Telex Architecture

III. Telex Middleware

A *document* is the basic sharing unit, and Telex represents its current state as a graph of submitted actions connected by constraints. As shown in Figure 1, it implements the history as a set of per-site logs, or *multi-log*. Furthermore, Telex allows applications to define *cross-document* constraints.

Figure 2 shows the overall architecture of a Telex instance running at some site. This instance supports several applications that together use its services. The Telex middleware is composed of two main modules — the scheduler and the replica reconciler — layered on top of two auxiliary modules — the transmitter and the logger. For each open document, Telex creates one instance of each module, which maintains the execution context of the document.

The transmitter and the logger are responsible for maintaining a replica of the document's multi-log at the local site. To this end, they implement an epidemic replication protocol which ensures that multi-log updates are eventually propagated to all *participating sites*, i.e. sites that collaboratively edits the document, either at the same time, earlier or later. The scheduler and the replica reconciler are described next.

A. Scheduler

The scheduler maintains an action-constraint graph that represents the state of the document known locally. Actions and constraints are added to the graph either by: the application, when local user updates the document, (ii) the logger, when it receives an update issued by a remote user, (iii) the replica reconciler, when it commits a schedule (see below).

Based on the action-constraint graph, the scheduler periodically computes *sound schedules*, i.e. sequences of actions that comply with constraints, and proposes them to the application for execution. In case some actions conflict, i.e. they do not commute or they are antagonistic, several schedules exist, each representing a particular solution to the conflict. The scheduler computes them one by one, upon application request, until one or more satisfaying schedules are found.

Actions submitted concurrently at different sites may turn out to be conflicting. Therefore, whenever a new action is addded to the graph, Telex checks whether constraints exist against concurent actions. Conceptually, Telex calls the application for every pair $\langle newaction, existingaction \rangle$ and adds to the graph the constraints that the application returns, if any. In order to optimize this CPU-intensive check, the application tags each action it submits with a set of numeric keys, one for each object that the action operates upon. Actions conflict only if their key sets intersect. Telex checks for this condition before calling the application, thus saving a significant number of unnecessary calls. False positives cause only a performance loss.

B. Replica Reconciler

Participating sites may generate different sound schedules from the same action-constraint graph. The role of the replica reconciler is to make sites agree on a common schedule to apply and thus achieve (eventual) mutual consistency. The agreed-upon schedule is said to be *committed*.

The replica reconciler implements a decentralized asynchronous commitment protocol based on voting. Periodically or on user request, each site proposes and votes for one or more schedules generated by the scheduler. Local user may specify the schedule(s) of his choice, if any. Votes are sent to participating sites, and a schedule that receives a majority or a plurality of votes is committed. The committed schedule is then materialized as a set of constraints added to the action-constraint graph.

The commitment protocol is fully asynchronous. It runs in the background and each instance determines locally when a schedule has won an election. Meanwhile, the scheduler keeps proposing (tentative) sound schedules to the application. In addition, the protocol may run only on a subset of participating sites that are know to be stable. The voting process is automated and does not require user intervention. The detailed protocol can be found in Sutra et al. [3].

IV. Shared Calendar Application

The Shared Calendar (SC) application design demonstrates how Telex facilitates the design of collaborative applications. The SC application provides users a way to manage their activities collaboratively. Each user has his own calendar, which he shares with the other authorised users.

To create a meeting with a group, a SC user creates a "meeting object" and shares it with the invitees. He notifies invitees by creating an action on their respective calendar.

When one receives an invitation he can accept it or decline it. If he accepts it, he can collaborate to hold the meeting: he can invite other users, and modify the meeting time, and location. For that purpose he creates actions on the corresponding meeting object, concurrently with other invitees. Consequently conflicts may appear. As we are in an optimistic replicated environment, those actions are tentative until committed. In case of antagonism, some of them are aborted.

A. Use case

Figure 3 shows an example concurrent execution of the calendar application. Users Jean-Michel, Lamia and Marc use

a Shared Calendar application to plan meetings between colleagues. Jean-Michel, Lamia and Marc are working separately and communicate only via the application.

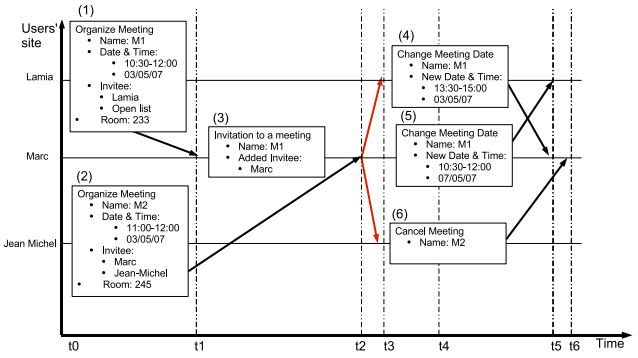


Fig. 3. Execution scenario the Shared Calendar application.

Jean-Michel organizes meeting *M2* on 3 May between 11:00 and 12:00. He allocates Room 245 for that purpose. He requires the presence of Marc and himself. This is illustrated by Action 2 in Figure 3.

Lamia organizes meeting M1 on the same day between 10:30 and 12:00. She allocates Room 233. She will attend the meeting, and allows other people to invite themselves, with actions 1 in Figure 3.

The application only needs to provide the above actions to Telex. Telex propagates them eventually to all interested sites (in this example, to Marc's site) even if some users are offline. Suppose that, at some point in time $t1$, Marc has received Lamia's actions, but not yet Jean-Michel's. This may happen, for instance, if Jean-Michel is working offline. Marc is interested in M1 and invites himself to that meeting (action 3 in the figure). Later, at $t2$ Marc knows Jean-Michel's actions.

As M1 overlaps, with M2 a conflict is detected at time $t3$. For this to happen, the SC application arranges that the corresponding actions' key sets overlap. Therefore, Telex upcalls SC, which returns an antagonism constraint, as explained elsewhere.

To avoid this conflict, Lamia shifts the start time of M1 to 13:30 (action 4). Concurrently ($t4$), Marc also sets the date of M1 to the 7 May (action 5), and Jean-Michel cancels M2 (action 6).

At $t5$ Lamia and Marc have received their concurrent modifications of meeting M1. Obviously M1 is scheduled at different times on Lamia's and Marc's calendar. SC provides the non-commute constraint, which causes Telex to order them the same way at all sites, after the commitment phase.

B. SC design using Telex

The application expresses its semantics by defining appropriate constraints between actions. The code for computing actions and constraints is part of the application; at run time SC outputs appropriate action and constraint instances to Telex, and Telex propagates them to the appropriate replicas. Thus application semantics is cleanly separated from the difficult systems task of ensuring consistency.

In more detail, the shared calendar application supports the following actions:

- *createEvent (meetingId)*: Create some event, for instance a meeting.
- *setInfo (time, meetingId)*: Modify the schedule of an event.
- *invite/addUser (userId, meetingId)*: Invite a person to an existing event.
- *allocate (roomId, meetingId)*: Allocate a room for an event.
- *cancelInvitation/cancelUser (userId, meetingId)*: Cancel an invitation.
- *cancelAllocation (roomId, meetingId)*: cancel a room allocation for a meeting.
- *Cancel (meetingId)*: Cancel a meeting.

Recall that this application supports optimistic replication. Each user of this application can generate one of the previous actions and execute it locally. However the execution remains tentative until an agreement phase reaches a consensus whether actions are committed, or aborted, or reordered.

In the use case scenario, Jean-Michel generates actions : A = *createEvent (M2)*, B = *allocates (245, M2)*, C = *addUser(Jean-Michel, M2)*, D = *addUser(Marc, M2)*. It groups them atomically with an enables cycle: `((A enables B) and (B enables C) and (C enables D) and (D enables A))`. He also generates E = *setInfo (11:00-12:00 03/05/07, M2)* and the constraint `A enables E`. Telex propagates these actions and constraints to Marc's site, as well as Lamia's actions concerning meeting M1.

Telex checks each pair of actions for their keys. Telex upcalls the application for possible constraints only two actions have a same key. SC computes its keys as follows. Each discrete 30-minute time slot has a unique identifier. For action *setInfo (description, time, meetingId)* where *time* is a set of slots $\{S_i, i = 1 \ldots n\}$ the generated keys are:

- The hash of *meetingId*,
- The hash of each slot identifier S_i.

Back to the use case scenario. At time $t2$ Marc receives actions F = *setInfo (" " ,10:30-12:00 03/05/07, M2)* and G = *setInfo (11:00-12:00 03/05/07, M2)*. Keys of the two setInfo match as both have a key that is a hash of the {11:00-12:00 03/05/07} slot. Therefore, Telex asks SC for the corresponding constraints. SC returns an antagonisism constraint `((F not-after G) and (G not-after F))`, which causes either F or G, or both, to eventually abort.

Telex suffers from some usability issues, because reifying application semantics into actions and constraints is not very intuitive. Furthermore, constraints are hard to validate.

To facilitate the use of Telex to develop collaborative applications, we propose a generic methodology for principled designs. A rule for using Telex is to make any shared mutable information a Telex-managed object. For instance, a meeting is an implicit information inherent to each invitees' calendar.

This information is shared between all invitees, and is mutable as any invitee can collaborate to modify the meeting information. The number of invitees is also dynamic. Thus the easiest way to manage a meeting is to make it an explicit Telex object, with corresponding actions and semantics. Figure 4 shows the state of Marc's site at time $t2$.

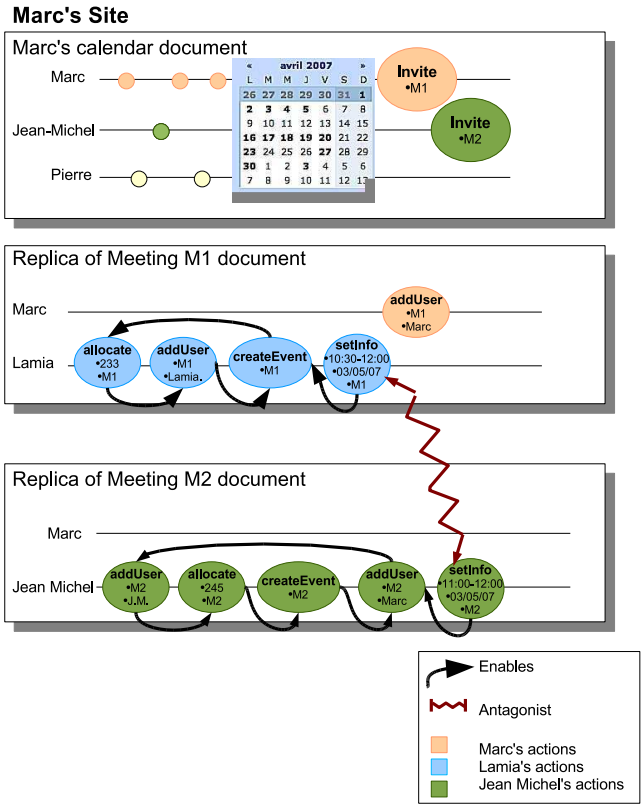


Fig. 4. Marc Site at $t2$.

The Telex scheduler insures that Marc will not be scheduled for $M1$ and $M2$ at the same time (Antagonistic actions). The reconciliation phase ensures eventual consistency.

V. Conclusion and Future Work

The Telex middleware facilitates the development of collaborative applications in mobile environment. The main contribution of Telex is that the reconciliation between replicas is application independent as Telex is semantically rich. We presented a design of a Shared Calendar application to demonstrate Telex benefits and highlight Telex usability limitations. To facilitate the use of Telex to develop collaborative applications, we propose a generic methodology for more principled designs. However this approach also suffers from limitations. Reifying application semantics into actions and constraints is not very intuitive. Furthermore, constraints are computed in advance, without knowledge of the actual state. We suggest two complementary approaches - A compiler should generate actions and constraints from a high-level specification - A checker should verify that all action-constraint combinations verify the application invariants.

References

[1] Y. Saito and M. Shapiro, "Optimistic replication," *Computing Surveys*, vol. 37, no. 1, pp. 42–81, Mar. 2005, http://doi.acm.org/10.1145/1057977.1057980.

[2] M. Shapiro, K. Bhargavan, and N. Krishna, "A constraint-based formalism for consistency in replicated systems," in *Proc. 8th Int. Conf. on Principles of Dist. Sys. (OPODIS)*, ser. lncs, no. 3544, Grenoble, France, dec 2004, pp. 331–345, http://www-sor.inria.fr/ shapiro/papers/opodis2004-final-2004-10-30.pdf.

[3] P. Sutra, J. Barreto, and M. Shapiro, "An asynchronous, decentralised commitment protocol for semantic optimistic replication," INRIA, Research Report 6069, 12 2006. [Online]. Available: https://hal.inria.fr/inria-00120734

Handling membership dynamicity in service composition for ubiquitous computing

Jeppe Brønsted
University of Aarhus
Computer Science Department
Åbogade 34, 8200, Århus N
Denmark
Email: jb@daimi.au.dk

***Abstract*—In ubiquitous computing, as more and more devices are introduced into the environment, new applications are made possible that exploit device capabilities in new ways. Currently, however, there is a mismatch between the effort involved in implementing these applications and the benefit they provide. Furthermore, there is a risk that the user loses the understanding of the system and although this is usually not a problem during normal use, it can be problematic if a breakdown occurs. A proposed solution that handles these problems is to use a service oriented architecture and implement applications as composite services and let information about the structure of the composites be available to the user at runtime.**

As long as the set of services that constitute the composite is static, traditional techniques can be used to specify the composite. But if the member set is dynamic it is problematic to specify which nodes partake in the composite and how they interact. In this paper we introduce mechanisms for handling membership dynamicity in service composition specifications. We demonstrate how an application scenario developed in cooperation with users can be implemented using the mechanisms and sketch how a decentralised interpretation can be realised.

I. Introduction

As more and more devices are introduced into our daily lives the vision of ubiquitous computing approaches realisation. Devices interact in new ways to provide services supporting users in home and work situations that were not possible until recently.

Currently, services incorporating multiple devices are typically implemented by device manufacturers to add increased value into their products and thus the possible applications are limited by what devices a particular company manufactures. Even when manufacturers cooperate to make their devices communicate using, e.g., open standards it is hard to predict which services the users want and which devices to combine. Furthermore, if a given service involves a particular set of devices, the service will only be available to users having that particular set of devices. Consider the following *geotagger* example mentioned in [1]:

> *A user has a GPS device, a digital camera, a mobile phone with GRPS, and a home server. When pictures are taken they are automatically tagged with positional information from the GPS and uploaded to the server using the mobile phone.*

In this application its not clear which manufacturer would implement the application since all the devices involved can be used for multiple purposes and should, e.g., the phone manufacturer decide to implement the service, it would only be of use to the subset of customers having the particular relevant device constellation.

Another issue with applications consisting of multiple devices interacting is that if a fault occurs, it can be hard to determine the cause. In the above example, the GPS device could run out of battery and unless the application have been designed with that particular contingency in mind, the user has to check each of the devices to resolve the situation. Another problem could be that the GPS device is configured to send messages in a format that is not understood by the application. In this case the user is probably even worse off because all devices will appear to be working correctly.

If device capabilities are exposed through service interfaces, anyone with access to the interfaces can, in principle, develop applications exploiting the devices and thus the manufacturers will not have to try to anticipate every combination a particular device might be involved in. To achieve maximum interoperability the manufacturers have to agree on a format for service specifications or at least make available the formats used. Whether the manufacturers are interested in this is not the topic of this paper but if a standard was agreed upon the utility of the devices would be enhanced.

Service composition have previously been used to implement applications for ubiquitous computing environments [2]. Service orientation alone will not solve the problem of creating services exploiting the particular device set a user has. If applications are implemented by developers using a service composition framework [3–5] it is hard for the user to control which services are used, how they are connected, and how they interact. Another option is to let the user try to specify the task he wants to solve in an abstract way and let the middleware determine how services should be composed [6–8]. This has the benefit that the user does not have to know which services are offered by the devices but only has to be able to specify the task to be solved. On the other hand, if a failure occurs, it can be hard for the user to find the error because the user's understanding of the system is rooted in the abstract task specification. A third option is to let the user experiment with devices and services and

manually compose the application [9–13]. This requires that the composition language or tool is sufficiently understandable and at the same time complex enough to make it possible to express useful applications. Previously, both centralised [9–11] and decentralised [12, 13] algorithms have been used to govern the flow of service invocations in the composite.

When users compose services into applications, it is problematic to handle composites with a varying number of members. An example could be a chat application that dynamically includes new devices as they arrive. If the application was expressed in source code, this would typically be handled by collections and specialised logic specifying how the composite evolves over time. Since source code is not understandable by end users in general, we do not consider this to be an option. To the best of our knowledge, none of the previous approaches deal with composites with varying member sets.

In this paper we investigate the problem of handling composites with dynamic membership. We present an extension of the PalCom assembly scripts that makes it possible to specify assemblies in which the member set varies over time and outline how a decentralised interpretation can be implemented. To evaluate the extensions we demonstrate how one of the PalCom prototype scenarios can be implemented using the mechanisms. The prototype scenario have been developed in cooperation with end users and represents realistic and relevant challenges.

The rest of the paper is structured as follows. In section II we briefly describe the PalCom architecture and the mechanism available for composing services into applications. In section III we present the prototype scenario and the problems involved in realising it using traditional techniques. In section IV we describe the extensions for handling membership dynamicity, describe how the prototype scenario can be realised using the extended assembly scripts, and outline how decentralised interpretation can be implemented. And, finally, in section V we conclude the paper and present future work.

II. The PalCom open architecture

In the PalCom project [14] an open architecture [15] have been developed that supports users and application developers in making more understandable applications. Using the architecture and the runtime system, applications can be built by composing services through scripts [1]. The scripts can be defined by application developers or by users interacting with a service browser [16]. Services can be composed at runtime and the internal structure of the composite can be opened up and inspected in case of a failure or a misconfiguration. Being able to inspect the running system supports users in understanding how the application works and determine in which part of the system the failure is located. The composite can be altered at runtime by replacing services with alternates to resolve failures or to adapt the application to changing network conditions.

The design of the architecture and the scripting language has been guided by requirements arising from a set of prototype scenarios developed in cooperation with users using participatory design techniques. The scenarios cover a broad range of application areas including support for major incidents, on site work by landscape architects, and rehabilitation of children with Downs syndrome. The scenarios have in common that they exploit combinations of digital devices to better support work situations and that they do not assume that a fixed network infrastructure is in place.

In PalCom, composite services called *assemblies* consists of simple event-based scripts that declare services and devices and describe the flow of events through the assembly. This is an easily understandable way of composing services and handles static situations with few devices well. However, the script language has currently no support for composites where the set of members varies over time. Furthermore, a URN [17] for each of the involved services must be known when the composite is created. This implies that it is, e.g., not possible to specify a composite that will dynamically include new devices of a particular type as they arrive.

A. PalCom assemblies

In the PalCom architecture device functionality is encapsulated in services that can be remotely discovered and invoked. Each service has a set of *commands* which can be either *in-going* or *out-going*. In-going commands are similar to asynchronous methods with an optional number of parameters. They can be invoked from other services or by the user. Out-going commands makes it possible for the services to provide output. The output can be used as input for in-going commands or can be presented directly to the user. The output has an optional number of parameters. An example could be a service acting as an interface for a lamp. The service would have two in-going commands `on` and `off` and an out-going command `state` that is invoked every time the state of the lamp is changed. Services and commands are composed in assemblies described by assembly scripts where services and devices are declared along a with description of which commands are connected. Variables that can hold state can also be declared.

The example given in the introduction can be implemented by the assembly script listed in figure 1. Lines 2–7 declares which devices take part in the assembly. Note that a unique name (URN) is given for each of the devices. Similarly, lines 8–13 declare the services in the assembly. In line 17–34 the flow of events through the assembly is described. E.g., lines 25–28 specify that when the out-going `photo_taken` command from the camera's `photo` service is invoked, the `tagger` service's `tag_photo` command on the mobile phone is invoked. In line 19 a variable is declared with a MIME-type [18] and in line 23 a value is assigned to it. The script in figure 1 can be created by using a text editor or by the user by interacting with a tool.

III. The Site Sticks scenarios

Since the users are involved in creating the assembly scripts, an important design goal is that the script language is as simple as possible. One could easily use a general programming language to implement the assemblies, but this would defeat the goals of simplicity and understandability. The assembly

```
1  assembly GeoTagger {
2    devices {
3      gps    = urn:palcom://gps;
4      camera = urn:palcom://camera
5      server = urn:palcom://server;
6      mobile = urn:palcom://mobile;
7    }
8    services {
9      gps on gps          = /gps;
10     photo on camera     = /photo;
11     tagger on mobile    = /tagger;
12     photo_db on server = /photo_db;
13   }
14   connections {
15    ...
16   }
17   script {
18     variables {
19       text/nmea-0183 gpsCoord;
20     }
21     eventhandler {
22       when position from gps on gps {
23         gpsCoord = thisevent.NMEA-0183;
24       }
25       when photo_taken from photo on camera {
26         send tag_photo(thisevent.Image, gpsCoord)
27           to tagger on mobile;
28       }
29       when photo_tagged from tagger on mobile {
30         send store_photo(thisevent.Image)
31           to photo_db on server;
32       }
33     }
34   }
35 }
```

Fig. 1. Geotagger script

script language has to be as simple as possible while at the same time powerful enough to support relevant scenarios. In this section we present the PalCom scenario Site Sticks [19] where a composite service with a dynamic member set is required for realisation. The scenario has been developed in cooperation with landscape architects from Edinburgh.

When landscape architects try to visualise how a project will blend into the landscape at a building site, a typical approach is to place marker sticks that represent the shape of buildings, roads, gardens, etc. as outlined in the digital building plans. Looking at a site with hundreds of sticks (see e.g. figure 2) it can be hard to figure out which sticks represent a particular building. The challenge is to visualise the digital design combined with the physical reality.

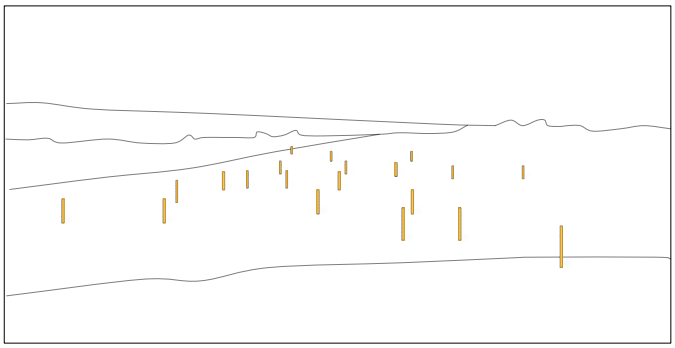


Fig. 2. Site Sticks

In the site sticks scenario, each stick is equipped with an embedded system with wireless communication. When the stick is placed in the ground, the position and role in the design is registered in the stick using a GPS device and a PDA. Later, when the architect wishes to visualise a particular part of the design, he can select the part on the PDA and the corresponding sticks will light up with a distinct colour. The initial placement and the registration will not be dealt with in this paper.

From a conceptual point of view its natural to express the application that makes a subset of the sticks light up as a composite service consisting of the sticks and a PDA. However, the scripting language presented in section II-A provides only limited support for the description of the assembly. One limitation is that the script will be big because all devices and services have to be declared explicitly. Another is, that the assembly cannot be expanded to include more sticks without changing the script. In the following section we propose extensions to the assembly scripts that will make it possible to express the application described more naturally.

IV. Handling membership dynamicity

The extensions of the assembly scripts we propose can be divided into two parts: *Selection* is about selecting which devices should participate in the assembly and *naming* deals with how to represent the services and devices in the specification of the flow of events.

A. Selection

Given a set of nodes we need a method for selecting those that should be part of the assembly. In the unextended version of the scripts this is done using URNs but for assemblies with dynamic membership this is, as mentioned previously, not enough. Instead, we propose to use a simple wildcard pattern on the device URN so that a single line in the device declaration part of the script (lines 3–8 in figure 1) can declare multiple devices. Lines not including a wildcard character ('*') will be interpreted as before. As an example, the line:

```
stick = urn:palcom://stick*
```

matches all devices with a URN with the prefix `urn:palcom://stick`. Hereby, all the sticks in the site sticks scenario can be declared in a single line.

While wildcards provides an easily understandable way of specifying a lot of devices, it has the drawback that semantic information has to be encoded in the device URN. In situations where devices with similar URNs have different service sets, it can be a problem only to include a subset of the devices in the assembly. Therefore, as an additional method for selection, we also use the information already present in the service declaration part of the script (lines 9–15 in figure 1) to exclude irrelevant devices. For example, line 13 in figure 1:

```
photo_db on server = /photo_db;
```

specifies that the service with the name `/photo_db` on the device `server` will be used under the name `photo_db` in the rest of the script. We propose that this statement will also imply that all devices matching the line that declares the device `server` must also have a service with the name `/photo_db` to be allowed into the assembly.

B. Naming

With the extension mentioned above, each line in the device and service declaration parts of the script potentially declares multiple devices/services and associates a name. This name is used in the eventhandler part of the script to specify the flow of events. We extend the semantics of the eventhandler part so that the line:

```
when photo_tagged from tagger on mobile {
```

is used when the out-going command `photo_tagged` is invoked on *any* `mobile` device. In the case that `mobile` only denotes a single device, the interpretation is unaltered. Similarly, the interpretation of the invoke part of the eventhandler is changed so that the line:

```
send store_photo(thisevent.Image)
  to photo_db on server;
```

will invoke the `store_photo` in-going command on *all* the devices denoted by `server`. Again, if `server` only denotes a single device, the interpretation is unaltered.

To allow for local flow of events on devices declared with a wildcard, the device name can be prepended with the keyword '`particular`' in the eventhandler. Assume for example that `my-device` is declared using a wildcard and the eventhandler includes the following lines:

```
when my-out-command from my-service-a
          on particular my-device {
  send my-in-command()
          to my-service-b on my-device;
}
```

Then, when the `my-out-command` is invoked on a particular device, the `my-in-command` will only be invoked on the same device.

The modifications above have the property that if no device is declared using a wildcard, then there is no modifications of the interpretation of the assembly scripts. This implies that the changes are backwards compatible.

C. Implementation of the Site Sticks scenario

We claim that the mechanisms for selection and naming described above can be used to implement the site sticks scenario in a simple and natural way.

Assume that each of the sticks is equipped with two services, `blink` and `building`. The `blink` service has a single in-going command `activate` that will make the stick light up. The `building` service has an in-going command `isPartOf` and an out-going command `isPartOfTrue`. If `isPartOf` is invoked with a building identifier that the stick represents a part of, the `isPartOfTrue` will be invoked.

The PDA has a `stick-gui` service where the user can specify which building he wants to see. The `stick-gui` service has a single out-going command `select` that will be invoked with a building identifier when the user selects a building.

Given the services and devices described above, the site sticks scenario can be implemented by the script in figure 3. Line 4 declares the sticks using a wildcard. Lines 17–20 forwards the user specified building ID to the sticks and if

```
1  assembly SiteSticks {
2    this = global-service-name;
3    devices {
4      stick   = urn:palcom://Stick*;
5      pda     = urn:palcom://PocketLoox;
6    }
7    services {
8      blink on stick     = /blink;
9      building on stick  = /building;
10     stick-gui on pda   = /stick-gui;
11   }
12   connections {
13     ...
14   }
15   script {
16     eventhandler {
17       when select from stick-gui on pda {
18         send isPartOf(thisevent.building)
19           to building on stick;
20       }
21       when isPartOfTrue from building
22         on particular stick {
23         send activate()
24           to blink on stick;
25       }
26     }
27   }
28 }
```

Fig. 3. SiteSticks script

the stick represents part of the building specified, the stick will be told to blink in lines 21–25.

D. Decentralised interpretation

One way to interpret the assembly scripts is to let a central node handle the eventhandler part (e.g. lines 16–26 in figure 3) of the scripts. Every time an out-going command is invoked, a message is sent to the central node which then determines which in-going commands should be invoked. As long as the assemblies only include a few nodes all connected to the same network, centralised interpretation is a viable option. However, for assemblies like the site sticks assembly it is not a good idea because the `building` service's invocation of the `blink` service (lines 21–25 in figure 3) would have to go through the central node. Therefore, we argue, it is necessary to interpret the scripts in a decentralised manner.

Decentralised interpretation can be implemented by distributing the script to all nodes in the assembly and let each node handle a part of the eventhandler script. When the member set of assemblies varies over time it is important that the distribution of the interpretation of the script is done in such a way that nodes leaving the network have as little influence as possible on the execution of the assembly. This means that if an assembly requires that a service S1 should invoke another service S2, the connection between the services should be handled by the nodes hosting S1 or S2. Hereby, that part of the assembly only fails if S1 or S2 fail.

We propose to divide the handling of the eventhandler part according to the following simple rule: An eventhandler clause is handled by the node originating the out-going command in the clause. As an example, the eventhandler clause in line 17–20 in figure 3 is handled by the `pda` node because the out-going command invocation comes from the `pda`. Similarly, lines 21–25 should be handled by each of the sticks.

Only nodes that participate in the assembly should receive the script. This requires that the distribution of the script itself depends upon the script. One way to accomplish this could be to flood the network in a n-hop radius with the script and let nodes not matching the device declarations ignore the script.

Unlike its centralised counterpart, decentralised interpretation requires that all nodes are able to interpret parts of the scripts and this excludes a class of devices with very limited resources. An alternative to completely decentralised interpretation is to relieve some nodes of the interpretation responsibility. This, however, requires a method for selecting the nodes taking over the responsibility and is left as future work.

V. Conclusion

In this paper we investigated the problem of handling composite services with dynamic membership. We proposed extensions of the PalCom assembly scripts that make it possible to specify composites with dynamic member sets and showed how a prototype scenario from the domain of landscape architects could be implemented using the proposed extensions. Finally, we outlined how the scripts can be interpreted in a decentralised manner.

We argue that the extended script language is simple and understandable. The idea behind the language is a basic case construct that specifies the flow of service invocations. A visual representation of the language might further support the user in understanding the scripts. Simplicity and understandability are relative to the user and therefore it is important also to make studies of end users working with the language. At present, none of the extensions have been implemented and therefore such studies are left as future work.

The suggested decentralised interpretation requires that all participating nodes are able to interpret the script and that can be a problem for very resource constrained devices. More powerful devices acting as proxies for the limited devices could be one way of handling this. In contrast to its centralised counterpart, the decentralised interpretation have the potential to scale better since no node have the sole responsibility for all communication in the assembly. Since no implementation exists at present, performance measurements are left as future work.

Acknowledgements

The author acknowledges the work done by the participants of the PalCom project to develop prototype scenarios and design and implement the PalCom open architecture and associated tools. The work presented in this paper has been supported by ISIS Katrinebjerg (http://www.isis.alexandra.dk).

References

[1] D. Svensson, G. Hedin, and B. Magnusson, "Pervasive applications through scripted assemblies of services," in *Proceedings of 1st International Workshop on Software Engineering of Pervasive Services*, 2006.

[2] J. Brønsted, K. M. Hansen, and M. Ingstrup, "A survey of service composition mechanisms in ubiquitous computing," to appear in 'Second Workshop on Requirements and Solutions for Pervasive Software Infrastructures (RSPSI)' at Ubicomp 2007. [Online]. Available: http://www.daimi.au.dk/~jb/papers/bronsted07b.pdf

[3] M. Román, B. Ziebart, and R. H. Campbell, "Dynamic application composition: Customizing the behavior of an active space," in *PERCOM '03: Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*. Washington, DC, USA: IEEE Computer Society, 2003, p. 169.

[4] 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, 2004.

[5] D. Chakraborty, A. Joshi, T. Finin, and Y. Yesha, "Service composition for mobile environment," *Mobile Networks and Applications*, vol. 4, no. 10, pp. 435–451, August 2005.

[6] E. Kıcıman and A. Fox, "Using dynamic mediation to integrate cots entities in a ubiquitous computing environment," in *Proceedings of HUC2000* , ser. LNCS, no. 1927, 2000, pp. 211–226.

[7] S. Maffioletti, M. Kouadri, and B. Hirsbrunner, "Automatic resource and service management for ubiquitous computing environments," *Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on*, pp. 219–223, 2004.

[8] M. Vallee, F. Ramparany, and L. Vercouter, "Flexible Composition of Smart Device Services," in *The 2005 International Conference on Pervasive Systems and Computing (PSC-05), Las Vegas, Nevada, USA., June*, 2005, pp. 27–30.

[9] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste, "Project Aura: toward distraction-free pervasive computing," *Pervasive Computing, IEEE*, vol. 1, no. 2, pp. 22–31, 2002.

[10] S. Ponnekanti, B. Lee, A. Fox, P. Hanrahan, and T. Winograd, "ICrafter: A Service Framework for Ubiquitous Computing Environments," *Proceedings of Ubicomp*, vol. 1, 2001.

[11] Y. Yang, F. Mahon, M. H. Williams, and T. Pfeifer, "Context-aware dynamic personalised service re-composition in a pervasive service environment," in *Proceedings of Ubiquitous Intelligence and Computing*, ser. LNCS, vol. 4159. Springer, 2006, pp. 724–735.

[12] W. K. Edwards, M. W. Newman, J. Z. Sedivy, and T. F. Smith, "Bringing network effects to pervasive spaces," *IEEE Pervasive Computing*, vol. 4, no. 3, pp. 15–17, 2005.

[13] W. K. Edwards, M. W. Newman, J. Sedivy, and S. Izadi, "Challenge:: recombinant computing and the speakeasy approach," in *MobiCom '02: Proceedings of the 8th annual international conference on Mobile computing and networking*. New York, NY, USA: ACM Press, 2002, pp. 279–286.

[14] PalCom - making computing palpable. [Online]. Available: http://www.ist-palcom.org

[15] PalCom, "PalCom External Report no 50: Deliverable 39 (2.2.2): PalCom Open Architecture," PalCom Project IST-002057, Tech. Rep., 2007. [Online]. Available: http://www.ist-palcom.org/publications/deliverables/Deliverable-39-[2.2.2]-Palcom-Open-Architecture.pdf

[16] ——, "PalCom External Report no 57: Deliverable 43 (2.6.2): End-User Composition: Software support for assemblies," PalCom Project IST-002057, Tech. Rep., 2007. [Online]. Available: http://www.ist-palcom.org/publications/deliverables/Deliverable-43-[2.6.2]-EndUserComposition.pdf

[17] R. Moats, "URN Syntax," RFC 2141 (Proposed Standard), Internet Engineering Task Force, Tech. Rep. 2141, May 1997. [Online]. Available: http://www.ietf.org/rfc/rfc2141.txt

[18] MIME Media Types. [Online]. Available: http://www.iana.org/assignments/media-types/

[19] PalCom: On Site. [Online]. Available: http://www.ist-palcom.org/application-areas/on-site/

End-to-end Middleware for Distributed Sensor Applications

Nelson Matthys, Sam Michiels, Wouter Joosen and Pierre Verbaeten
IBBT-DistriNet, Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{nelson.matthys, sam.michiels}@cs.kuleuven.be

***Abstract*—Many industrial applications, such as supply chain logistics, may considerably benefit from the use of wireless sensor networks (WSNs), as their involved logistics services (product tracking, localization, or asset monitoring) require real-time sensor information. However, developing such distributed sensor applications is a complex process, as these applications are distributed in an end-to-end fashion across a huge amount of heterogeneous platforms, and deployed in dynamic operational environments. Therefore, a middleware platform is needed which offers on the one hand generic services and enables the integration of application-specific plugins, while on the other hand offers support for the composition of those services under various scenarios. In this paper, we present the key research challenges in developing and managing distributed sensor applications. We describe the main components of the middleware architecture and show their relation to the identified key challenges. This analysis and design of a middleware architecture is an important step towards using WSNs in a realistic business context.**

***Index Terms*—Sensor middleware, distributed software architecture, multi-tiered architectures**

I. Introduction

The environments in which distributed software applications execute become more and more dynamic, heterogeneous and integrated with the physical world. It is generally accepted that wireless sensor networks form a key and emerging technology that enables this evolution. Supply chain logistics represents a typical example of such a distributed application that can benefit from the use of WSNs to track, localize or monitor products while they are being transported using different means such as trucks, ships and trains.

Applications for distributed logistics control, for instance, may use a large number of sensor nodes and RFID-tags that are deployed close to - or even inside - the object being observed. For instance, fruit and vegetables logistics providers might equip each crate with a sensor node to track their equipment and monitor the conditions of the products being placed inside. The asset management system of the logistics provider could then monitor the conditions of the transported products. Whenever certain conditions are violated (e.g. temperature exceeds a given threshold), the system can trigger an alert and inform the responsible employees.

Many of these applications are implemented as cooperating software components residing on different infrastructural components, such as programmable sensor nodes, (mobile) gateways or access points, end-user devices, and back-end systems. For instance, the enterprise asset management system consists of services, deployed on sensor nodes, (e.g. temperature sensing or localization), on gateways or access points (e.g. aggregation or access control), and applications on an employee's PDA (e.g. a web-browser) or back-end servers (e.g. the management system).

However, as WSNs arise as a promising solution for these distributed sensor applications, their development and management is complex and thus comes at a certain cost. Besides the typical characteristics of wireless sensor networks, such as resource-scarcity, energy limitations, and unreliable wireless communication, this complexity is mainly due to three reasons.

First, applications are distributed in an end-to-end fashion across (i) programmable sensor nodes attached to products or packings, (ii) (mobile) gateways and wireless access points, placed in warehouses or storage sites, and (iii) a network of end-user devices and enterprise back-end systems. This means that applications must be able to deal with various kinds of *heterogeneity*, both in platform and network type (resource-constrained wireless sensor nodes, gateways and fixed WiFi access points, application servers), as well as in environment (deployed in indoor, outdoor, or even mobile environments).

Second, as logistics providers ship millions of tons of goods annually, distributed sensor applications typically involve huge amounts of devices. This extreme *scale of deployment* makes manual management of all devices infeasible.

Finally, the execution environment of distributed applications is subject to *dynamism*. Changes in end-user application requirements, user mobility or other scenarios requires the distributed application to take specific actions. For instance, when products enter a warehouse, the inventory management system could automatically require a more fine-grained localization service to be deployed on each product.

These three challenges indicate that the provision of an end-to-end sensor middleware could support the distributed sensor application's developer and manager. We define such an end-to-end sensor middleware as follows: a middleware layer that is distributed over, and build upon a heterogeneous set of devices, ranging from small programmable sensor nodes, over gateway devices and network infrastructure, towards back-end systems. This middleware layer can be tailored to fit on each device platform and it can be customized to provide support according to the applications' different needs.

The main contributions of this paper are that (i) it identifies the key challenges that make the development and man-

agement of distributed sensor applications hard, and (ii) it proposes an end-to-end sensor middleware architecture that tries to tackle these challenges.

The paper is organized as follows: Section II gives a short overview of the domain of distributed sensor application development and management, while Section III elaborates on the key research challenges for building such systems. Section IV sketches our proposed middleware architecture and illustrates how this middleware handles these challenges. Section V illustrates related work. Finally, Section VI summarizes our contributions and identifies future work.

II. Analysis

This section provides an in-depth analysis of the problem domain of distributed sensor applications. It describes the involved stakeholders and their role on the development and management of the different installed applications and services.

A. Involved stakeholders

There are many different stakeholders involved in the development and management of distributed sensor applications. We identified four main categories of stakeholders: end-users, technology providers, platform providers and integrators.

In the context of supply chain logistics, *end-users* could be, for instance, logistics providers, manufacturers, or wholesalers. These end-users typically pose several application and information requirements, that might change over time.

The required hardware technology and sophisticated algorithms or protocols for distributed applications are delivered and implemented by *technology providers* or *specialists*. In the previously sketched context, programmable sensor nodes, RFID-tags and readers form the underlying technology, while algorithms like localization, protocols like timesync, or platform-specific implementations like a temperature sensor driver are typically provided by technology specialists.

Platform providers are responsible for the coupling of the different hardware platforms into a common communication platform. This communication platform allows software components, residing on different devices, to exchange any type of information with each other.

Finally, *integrators* are responsible for the integration of application requirements from end-users with the platform and technology from platform and technology providers. This integration is translated by the development and deployment of several application components on the underlying communication platform. These application components may use the platform's offered services to achieve their goals.

B. Involved services and their management

Distributed software systems consist of several cooperating services and applications. Services are implemented as components, offering specific functionality, and exposing their API towards applications or other services. Applications are software components that may then use these APIs in combination with application logic to achieve their goals.

Services can be classified into two categories. First of all, there exist services that are needed to perform application-specific functionality, such as localization, tracking, temperature sensing, etc. Second, there exist services that are related to non-functional requirements, such as authentication or authorization, aggregation, group management, etc. The first category is in general used by integrators to develop applications, while the second category is in general used by platform providers, responsible for network management.

Several stakeholders are responsible for the management of the distributed software system. Integrators are responsible for the management of business applications, and thus they need to maintain the different installed applications and services on the distributed communication platform. Platform providers are responsible to keep the underlying communication platform up and running, which includes the management of data (e.g. by providing caching in the infrastructure) and general network management. Finally, technology providers are only responsible for the maintenance of system-specific components residing on each device.

III. Research challenges

Several key challenges arise when building such complex distributed systems based on sensor networks. In this section, we identify three key challenges that must be handled when developing and managing realistic sensor applications: (i) heterogeneity, (ii) scale of deployment, and (iii) dynamism.

A. Heterogeneity

Heterogeneity is a complex key challenge to deal with when building distributed software systems, especially in the context of sensor networks as it mainly complicates uniform management. Heterogeneity exists at multiple levels: (i) resource and network heterogeneity, (ii) heterogeneity in posed requirements, and (iii) platform heterogeneity. From a software perspective, modularization is an excellent approach to deal with these different kinds of heterogeneity, as it allows one to customize the software being installed on the different infrastructural components and under different application scenarios.

In the context of supply chain logistics, resource heterogeneity is very common as there are many different types of devices involved, ranging from simple wireless sensor nodes, gateway nodes, employees' PDAs towards powerful application servers. Heterogeneity in network types is also common since sensor nodes communicate with each other using low bandwidth radio communication (ZigBee), while PDAs and gateways or access points may use WiFi, and application servers in general use high-speed network connections. Requirements can be heterogeneous as well, as they may be related to quality-of-service (QoS) or quality-of-data (QoD), security, performance, etc. Finally, besides sensor networks, other kinds of distributed systems, such as P2P networks or grid infrastructures for data processing, may as well be involved in the context of distributed industrial applications, resulting in a wide heterogeneity in computing platforms.

B. Scale of deployment

Realistic distributed sensor applications typically consist of a huge amount of devices, crossing system administration and enterprise boundaries. As a result of this large-scale deployment, manual management of all devices is considered infeasible as it is cost-inefficient and often error-prone. High-level policy interpretation and distributed coordination mechanisms can provide a solution to perform safe and efficient automated management.

C. Dynamism

The operational environment of large-scale distributed systems is subject to changes in time. To deal with this dynamism, distributed applications should be developed using reusable components. This approach allows one to easily replace individual components. Together with this component-oriented approach and by implementing coordination mechanisms, one can achieve consistent system and platform-wide adaptations.

In realistic distributed applications, dynamic changes happen due to, for instance, the changing of end-users' requirements, network or device failures. This dynamism may have influence on the installed applications and services. Not only existing functionality can therefore alter or be removed, also new functionality could be required to be installed. This adaptation of functionality and behavior of distributed sensor applications is not without any risk. First, one must take care that existing applications can be broken if an installed component is modified, disabled or completely removed. Second, safe distributed reconfigurations are difficult to achieve in sensor networks mainly through resource limitations, unreliable communication, and the huge amount of involved devices.

IV. Architecture

Based on the key research challenges, discussed in Section III, we propose in this section an architecture for an end-to-end sensor middleware that handles these challenges. The architecture is an integration layer on top of an underlying multi-tiered system. This layer allows one to deploy applications on it, as well as provide management support for the underlying distributed system.

A. Underlying system

As explained in Section I, distributed applications using sensor networks traditionally consist of different levels, namely programmable sensor nodes, gateways, and back-end systems. Therefore, the underlying system to deploy distributed sensor applications on will be based upon multiple tiers, where each tier houses a device class.

In the context of supply chain logistics, we consider at least four different classes of devices, which are illustrated in Figure 1. Basic sensor nodes, attached to products, reside in the basic or lower tier, while more advanced sensor nodes, attached to logistic infrastructure, reside in the aggregate tier. Gateways, access points and employee's portable devices reside in the communication tier, while enterprise systems running back-end application software reside in the highest or enterprise tier.

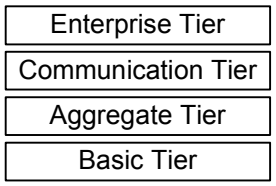


Fig. 1. Multi-tiered architecture for the supply chain platform

B. End-to-end middleware as integration layer

There is the need to integrate the technology and software components residing on each tier in order to form a common end-to-end platform where integrators and service providers can easily deploy applications and services on. The middleware therefore provides a uniform layer on each of the four tiers. This integration layer provides support for customization and management of component compositions, tailored to the capabilities and responsibilities of the underlying device platform.

C. Key middleware components

The proposed middleware architecture is illustrated in Figure 2, and is based on a component-oriented programming model. The architecture is structured around three main subsystems. Each subsystem is responsible for a category of management: (i) management of installed functionality, (ii) platform and network management, and (iii) device specific-management. The component-oriented programming model allows each subsystem to be customized to the hardware capabilities and responsibilities of the underlying devices.

1) Subsystems: The *core runtime subsystem* is needed to assure the correct execution of the installed components. To handle resource, network and platform heterogeneity (see Section III), the core runtime subsystem is implemented on top of a uniform layer: the *core runtime API*. The API's functionality is implemented in a platform-specific manner. The API includes operations for instantiation, linking and removal of components, as well as for keeping track of the interactions between the different components in the middleware framework. The subsystem itself provides an API that can be used by the different responsible stakeholders to gain access to and information from the subsystem's components.

The *management subsystem* is used by network managers to control the behavior of the underlying communication platform. For instance, one can install caching services, perform automated management and grouping control to deal with the scale of deployment and the inherent dynamism of the infrastructure due to network failures and topology changes.

Finally, the *functional subsystem* exposes a well defined API towards integrators or service providers to allow them to deploy services (e.g. a localization service) and applications (e.g. a container monitoring application, combining measurements from different types of sensors) on the platform. All services have a service descriptor *SD*, which is a high-level language description used to describe the behavior of the service (e.g. to specify its required services or interaction paradigms).

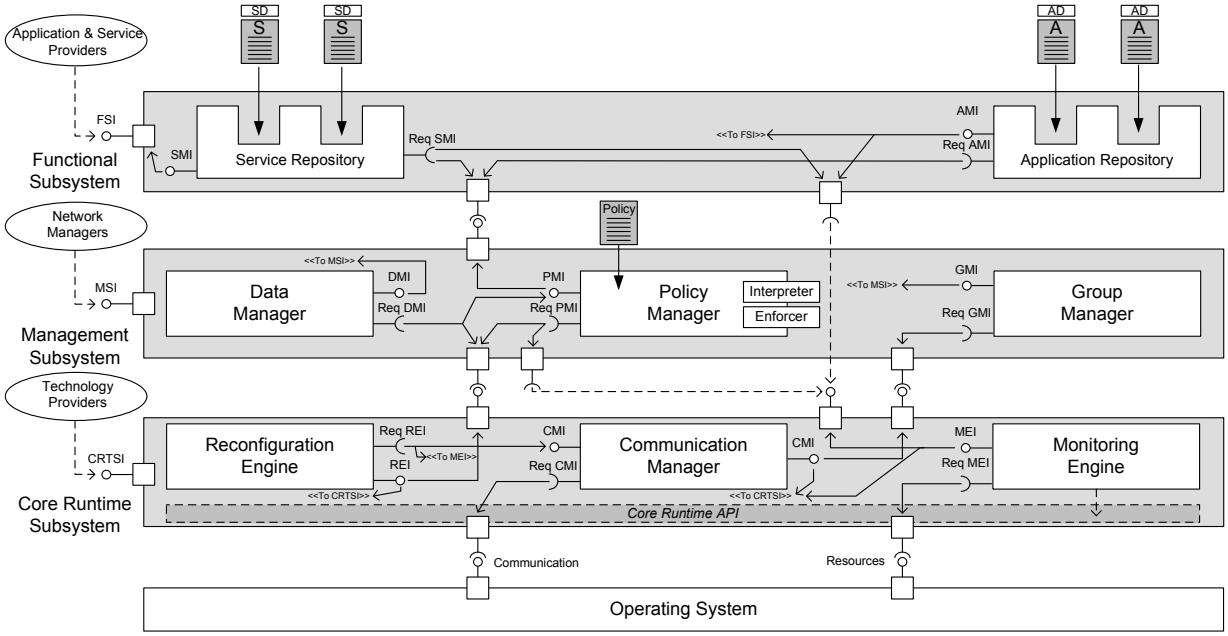


Fig. 2. The proposed middleware architecture, as seen from a local perspective

A similar descriptor (*AD*) is included with each end-user application component. This AD can for instance specify that the container monitoring application requires three different sensor services (e.g. temperature, humidity, door monitoring).

2) Components residing in each subsystem: The *monitoring engine* is responsible for gathering information about the different interactions and connections between the installed components. This information can then be passed to the *reconfiguration engine* to assist with dynamic reconfigurations (e.g replacing a service that is required by other services). Communication between different devices across adjacent tiers is possible through the *communication manager*, which provides a uniform abstraction for inter-component communication.

The management subsystem consists of network and resource management components, such as a *group manager* (e.g. to gather temperature readings from a group of nodes) and a *data manager* (e.g. to provide data caching of temperature readings). To deal with the management of large-scale sensor applications, we provide a *policy manager*. This component is connected to the monitoring engine to gather details about the current middleware configuration. It deals with the translation of business policies into lower level configuration descriptions. The policy manager can for instance instruct the reconfiguration engine to execute a dynamic reconfiguration if the operational environment is subject to change (e.g. when products are placed in a cooled container, the middleware can decrease the temperature sampling rate or turn off the service).

Finally, the functional subsystem provides a *service repository* and an *application repository*, which allows one to keep track of the installed services (resp. applications) and deploy new services (e.g. localization or temperature sensing), resp. applications (e.g. alert signalization). Both repositories depend on the policy manager, as they take application or service descriptors and pass them to the manager. Based on the installed policies, the policy manager can then pass this information to other components (e.g. the caching data, coming from the temperature service, in the data manager, or reconfiguring the middleware to install the required services).

D. End-to-end distribution

Figure 3 gives an example of a possible configuration of the sensor middleware across multiple tiers in the context of supply chain logistics. Each tier consists of a customizable subset of interacting components, associated with the tier's responsibilities and capabilities.

As the figure illustrates, the basic tier is only offering simple services to applications residing on its upper tiers (e.g. temperature sensing), whereas the aggregate tier is responsible for the managing, tasking and aggregating data of its associated nodes in the basic tier. The communication tier is mainly responsible for data and network management through, for instance, data caching and access control to network resources. Finally, the enterprise tier in general does not take care of network management. It only executes end-user applications (e.g. the enterprise asset management system) and requires data coming from the network. The enterprise tier is however mainly responsible for the triggering of dynamic reconfigurations, which then need to be performed inside the network. Finally, to deal with the extreme scale of deployment, each tier has the policy interpreter and reconfiguration engine enabled, to allow uniform and flexible management of the entire distributed infrastructure.

V. Related work

Tenet [3] leverages on a two-tiered architecture for sensor networks that provides a fixed tasking library to program

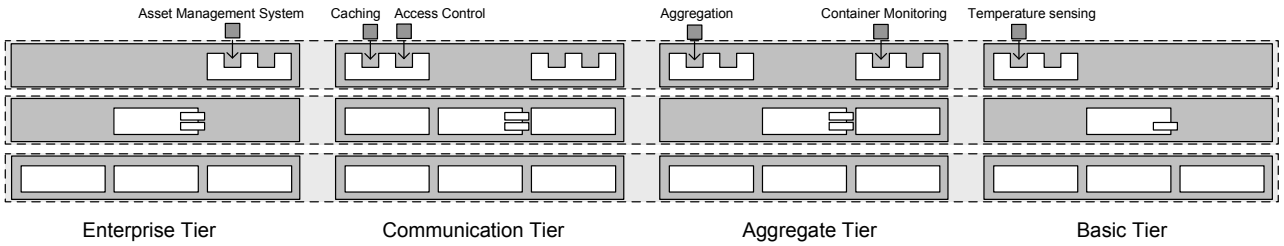


Fig. 3. Example configuration of the middleware in the context of supply chain logistics

applications. Nodes in the lower tier are managed and tasked by the upper tier nodes. In this context, the tenet approach fits in between the aggregate and basic tiers. Dynamic reconfigurations are however not possible, as the tasking library cannot be extended dynamically.

DAViM [6] leverages on state-of-the-art virtual machine technology for sensor networks. It treats the sensor network as a lightweight service platform, by allowing multiple applications to be run concurrently. Additional services and applications can be dynamically installed, changed and removed in the network. DAViM can serve as an representative middleware platform as it implements similar service and application repositories, as well as a concurrency manager. We argue that an extended version of DAViM may serve as a platform for the two lower tiers, however it is currently bounded to the devices residing in the aggregate tier due to its memory requirements.

To support the dynamic deployment of distributed applications on heterogeneous platforms, the RUNES middleware [2] leverages on the GridKit [4] middleware. RUNES implements a middleware kernel with a fixed API and provides reconfiguration support to deal with environmental dynamism. However it does not support large-scale application management.

The OSGi framework [1] consists of a component model, which is suitable for mobile devices residing in the communication tier. OSGi allows components to be adapted at runtime, however it does not provide support distributed reconfigurations, component monitoring, and policy-based management.

Finally, policy-based management approaches, such as Ponder2 [5], exist for sensor networks. Ponder2 implements a Tiny Policy Interpreter that provides support for specification and enforcement of dynamically loadable policies. The Ponder2 approach is comparable with our proposed policy interpreter, however it is focused on policy enforcement and not on the translation into lower level configuration descriptions.

VI. Conclusion and future work

We described the key research challenges (heterogeneity, scale of deployment, and dynamism) that must be handled when building large distributed sensor applications. Based on these challenges and from a representative example in the domain of supply chain logistics, we proposed a customizable end-to-end middleware architecture on an underlying multi-tiered platform. The middleware integrates the different tiers, and provides, through specific components, support for the identified key challenges.

Although many research projects are investigating the development of sensor middleware, we are not aware of other in-depth requirements studies within a particular application domain, especially when focusing on the end-to-end spectrum of sensor application development. Therefore, we would like to receive support from the community whether the identified subsystems and components (especially the monitoring engine, policy and data manager) are relevant. In the near future, we will also consider security and interoperability between different platforms and (remote) services as other important key challenges for building distributed applications.

We are currently implementing a proof-of-concept of the architecture in the context of a project with industry in which applications for temperature control and tracking will be deployed. The prototype will combine heterogeneous sensor nodes running various operating systems in different tiers. We are especially investigating the policy-based approach to manage these large-scale distributed systems.

Acknowledgment

The authors are grateful to Wouter Horré for his valuable comments on the paper and for proof reading the text. Research for this paper was sponsored by IBBT, the Interdisciplinary institute for BroadBand Technology, and conducted in the context of the MultiTr@ns project [7].

References

[1] OSGi Alliance. About the OSGi Service Platform, technical whitepaper, revision 4.1, June 2007.

[2] Paolo Costa, Geoff Coulson, Cecilia Mascolo, Luca Mottola, Gian Pietro Picco, and Stefanos Zachariadis. Reconfigurable component-based middleware for networked embedded systems. *International Journal of Wireless Information Networks*, 2006.

[3] Omprakash Gnawali, Ben Greenstein, Ki-Young Jang, August Joki, Jeongyeup Paek, Marcos Vieira, Deborah Estrin, Ramesh Govindan, and Eddie Kohler. The tenet architecture for tiered sensor networks. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (Sensys)*, Boulder, Colorado, November 2006.

[4] Paul Grace, Geoff Coulson, Gordon S. Blair, and Barry Porter. Deep middleware for the divergent grid. In *Proceedings of the ACM/IFIP/USENIX 6th International Middleware Conference*, pages 334–353. Springer Berlin / Heidelberg, 2005.

[5] S Keoh, K Twidle, N Pryce, A. Schaeffer-Filho, E. Lupu, N. Dulay, M. Sloman, S. Heeps, S. Strowes, J. Sventek, and E. Katsiri. Policy-based management of body-sensor networks. In *Proceedings of the International Workshop on Wearable and Implantable Body Sensor Networks*, March 2007.

[6] Sam Michiels, Wouter Horré, Wouter Joosen, and Pierre Verbaeten. Davim: a dynamically adaptable virtual machine for sensor networks. In *Proceedings of the First International Workshop on Middleware for Sensor Networks (MidSens)*, November 2006.

[7] Multitrans project. http://projects.ibbt.be/multitrans.

Using COSMIC – A real world case study combining virtual and real sensors

Michael Schulze and Sebastian Zug
Otto-von-Guericke University
Faculty of Computer Science
Department of Distributed Systems (IVS)
Universitätsplatz 2
D-39106 Magdeburg
{mschulze, zug}@ivs.cs.uni-magdeburg.de

Abstract

The cooperation of distributed nodes in sensor networks forms a dynamic structure of information providers and information consumers termed as sources and sinks. Often, the used nodes differ by the available performance, network capabilities, operating system, applications etc. although, all of them have to be integrated in an appropriate network structure. Hence, a middleware is necessary to provide a common communication interface for the network in the whole system to cover the heterogeneity. To enable the integration on different platforms and into different systems the COSMIC middleware itself is designed flexibly and adaptively.

In this paper we present a cross platform case study, which shows the information exchange via COSMIC between micro-controllers and PCs on different network types by C or C++ applications and Matlab/Simulink. The case study illustrates, apart from other features, the possibility for an experimental setup combining virtual and real sensors/actuators in the sense of hardware in the loop scenarios.

1 Introduction

Complex mechatronic systems like cars, mobile platforms etc. join a large number of embedded sensor/actuator modules which individually or combined make information available, support calculations and data exchange or influence their environment actively. These systems are currently limited in their implementations and behaviour by a predefined structure of functionalities, interfaces and information sources. Additional interaction with other dynamically appearing components of an intelligent environment are neither intended nor possible. However, only if an application can use all available possibilities of its environment, its tasks will be optimally completed. Hence, for a distributed approach of data aggregation, analysis and the resulting interactions a middleware is necessary.

The role of the middleware in an embedded network is manifold: Firstly, as argued above, it has to hide the different addressing and routing mechanisms of the various physical sub-networks. All applications should use a common communication interface. Secondly, as the underlying networks often have different quality properties, it must provide means to handle this. Additionally, the middleware should support dynamic network configuration issues like adding or omitting components without reconfiguring all subnets completely. Therefore the chair of Embedded Systems and Operating Systems developed an event-based Publish/Subscribe middleware termed COSMIC (**CO**operating **SM**art dev**IC**es).

In this paper, we describe a case study based on an experimental setup using COSMIC to illustrate the potential of a common communication interface for the development of embedded systems. The paper is structured as follows: In section 2 we briefly introduce the concept of COSMIC. Section 3 describes the experimental platform, the application structure and analysis options and challenges. The conclusion and future remarks are summarised in section 4.

2 COSMIC

The COSMIC middleware described in [3, 4] offers an event-based communication model according to the publisher/subscriber concept. COSMIC is especially designed to allow cooperation between smart sensors and actuators on different hardware platforms ranging from 8-bit micro-controller up to 32-bit PC/Workstations and interacting over a broad variety of communication media like Controller Area Network (CAN) [8],ZigBee [9], TCP/IP to name a

few.

In COSMIC an event is a programming abstraction and the carrier of the exchanged information. A COSMIC event consists of three different parts:

- a subject, represented by a unique identifier (UID) that describes the content,
- the content or payload itself for instance the value of a distance measurement and
- additional attributes (e.g. sensor position, context, quality) which are optional.

Events may arise in two different ways . Firstly, an event is spontaneously generated by the hardware because of a detection on a sensor interface. This means the physical environment is the stimulus of an event. Secondly, an event is periodically initiated by a clock to sense a change of a variable or of a state within the system.

Beside events COSMIC uses event channels as abstraction for event transfers. An event channel has the same subject as the corresponding event. The event is published by pushing it to the according event channel. In case of subscription, the occurrence of an event is notified to the application by the event channel. The programming abstraction event channel is introduced mainly to map the UID of an event to specific network addresses and therefore it hides the heterogeneity of the different network architectures by providing a global addressing scheme. Furthermore the event channel is used for network resource allocation - for instance a part of the bandwidth. Depending on temporal constraints or the importance of the event, the event is classified into three different quality levels which are hard real-time (HRT), soft real-time (SRT) and none real-time (NRT).

In COSMIC all events are handled by the event channel handler (ECH) which is part of the event layer (EL). The EL- marked by a filled circle in figure 1- is the interface to application level. Consequently, the publisher uses the EL to send events to event channels and on the other hand the EL provides the subscriber with notification and supports it by reading of events from event channels.

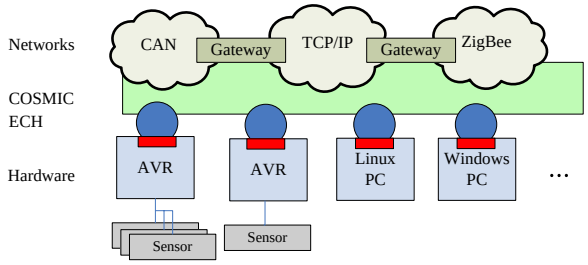


Figure 1. Current network structure

A possible implementation of the network structure with various hardware platforms as well as communication media is shown in figure 1. To allow information exchange between publisher and subscriber over network boundaries, the networks are connected by gateways. Currently, implementations of COSMIC for Atmel AVR micro-controllers, Motorola HC08, Siemens C167, Linux and Windows were realized which support the communication via CAN and TCP/IP.

The COSMIC middleware is implemented as a family [6] to achieve flexibility, adaptability and dependability. The implementation for the AVR microcontroller is an example of such a member of that family. The common functionality of the family is an intrinsic part of each member regardless of the platform where COSMIC is instantiated. Then the platform-specific functionality is only part of the relevant family member. Moreover, the design as a family allows a fine-grain selection of the required functionality necessary to spare the constrained resources on the embedded system. For example to deal with the limited memory (e.g. few kilobytes in sensor nodes), the software should only provide the functions actually needed by the application. In order to reach this goal, the functionality of the middleware is a collection of configurable components and functions. Design decisions about the required properties are deferred as long as possible and often determined by application needs.

Apart from the functional properties, the encapsulation of non-functional requirements - like dependability issues and real-time properties - need a separate treatment. These requirements - termed crosscutting concerns - are often fundamental system policies and refer to issues as robustness, fault-tolerance and real-time. Since crosscutting, quality features may apply to multiple functions and it is impossible to implement them as independent encapsulated entities. However, this would restrict the above-mentioned freedom of selection and adaptation. Aspect-oriented programming (AOP) seems to be a suitable possibility to deal with crosscutting concerns [5]. AOP allows separating the functional middleware components and non-functional components called aspects. Aspects are woven into the middleware during build time. Thus there is no extra runtime overhead to dynamically introduce these aspects.

3 Case Study: Interoperable network

3.1 Hardware description

As test environment for our software we use a development platform consisting of 4 nodes connected via a Controller Area Network (CAN). Each controller can be equipped with additional interfaces (sensor connection, ZigBee board, serial communication adapter) and is programmed separately or jointly

using CAN very comfortably. One of the nodes includes a LED array for visualization. For simulation of failures, each controller can be switched off individually. Therefore, tests for a dynamically changeable network structure are possible. As depicted in figure 2, two sensors are integrated, i.e. a temperature and a distance sensor as representation of real sensor data. PCs can be integrated in the CAN network as well as being connected with each other via TCP/IP.

The micro-controllers run our PURE operating system and the PC works under Linux (in near future under Xenomai - areal-time Linux extension [1]) or Windows. The communication is handled by COSMIC.

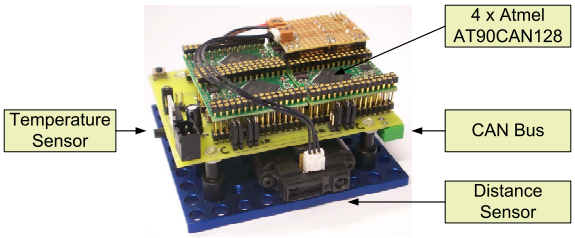


Figure 2. Development platform

As actuators, several mobile robots can be integrated in the network.

3.2 Software structure

The test environment demonstrates the flexibility of the distributed approach in two scenarios

1. "Real" sensors produce information which is consumed by other participants for data logging, visualization and data fusion.

2. Virtual sensors - like a PC - publish logged or calculated data instead of measured values.

The subscriber cannot differentiate between these two channel sources. Hence, the seamless interchangeability provides modern development methods like Hardware and Software in the Loop.

For the first case, our distance sensor periodically captures a voltage value as representation of the distance between sensor and an obstacle . This information is published on the CAN, and node 3 in figure 3 reflects the events by an LED array. The Linux PC works in two ways. Firstly, it acts as a gateway for the connection of CAN and TCP/IP. Secondly, a small C application runs, which is subscribed for the distance channel. It calculates a distance in $[cm]$ by the voltage value and logs all values at the same time. The use of Xenomai will offer real-time functionalities in this context. In Matlab/Simulink a median filter and a graphical user interface is used for a comfortable visualization of the sensor data. Further developments for data analysis, fusion and visualization will exploit the manifold number of toolboxes for Rapid Prototyping. In order to provide real-time applications such implemented models can be transferred and used by code generation tools for different platforms.

In the second scenario, sensor node 2 is switched off. Logged or calculated data are propagated by a Linux application or from Matlab/Simulink. Node 3 depicts the value as in the first scenario. This means if a mobile robot logs all information about its environment once, the replayed data can be used for instance in reproducible examinations of navigation or control algorithms .

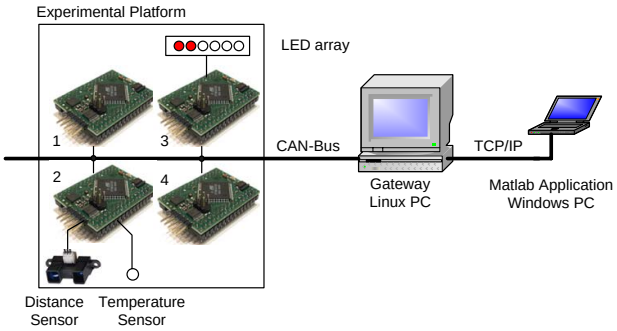


Figure 3. Scenario software scheme

4 Conclusion and Future Remarks

The simple demonstration setup illustrates the practical possibilities of a distributed system with a transparent communication via middleware.

1. A transparent and defined uniform interface supports application developers considerably. The design and implementation of distributed embedded systems can be accelerated by concentration on the application.

2. The dynamic adaptability of the system on runtime reacts to the appearance and disappearance of components caused by communication problems.

3. Exchangeable data sources and sinks simplify experiments combining real components and virtual modules. Hence, the test scenarios can use predefined reproducible information generated by a simulation first and validate the results by real measurement data.

4. The monitoring of the data transfer by established engineering programs supports Rapid Prototyping development for data aggregation, fusion and interpretation.

Point 1 in this enumeration is based on manifold versions of COSMIC. Hence, a family of COSMIC

and appropriate operating systems are necessary for different hardware environments.

The current COSMIC implementation offers non-real-time communication only. For further extensions a time synchronisation will be included to offer real-time event channels. Therefore first drafts of the synchronisation algorithm presented in [2] reach average time deviations of $6\mu s$. This synchronisation is fundamental for more complex dynamic data fusion algorithms.

For an adaptability of the network mentioned in point 2 and 3 two aspects have to be considered. Firstly, each channel should provide information about its events like sensor type, measuring ranges, noise performance etc. Therefore [7] integrates functionalities for electronic data sheets and establishes appropriate service discovery functions in COSMIC. Secondly, intelligence for the information selection has to be designed for data sinks.

Point 4 is aimed at an advanced integration of Matlab/Simulink toolboxes for a flexible and dynamic data fusion.

As a more interactive test platform, we will use "Q" - a quad drive robot 4 - for further steps. Q represents the distributed approach very consequently. It consists of four independently steerable driving units, each monitored by a micro-controller and a number of infrared sensors. Additional gyroscopes and compass modules can be integrated. The network is connected by a CAN.

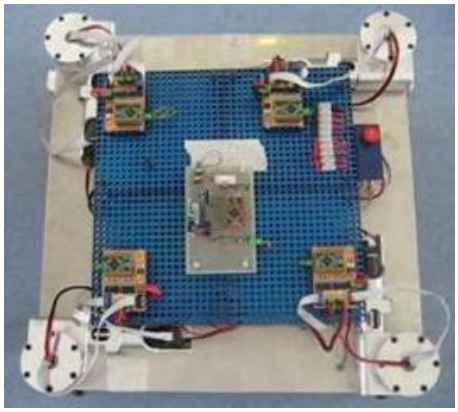


Figure 4. Mobile robot Q

The sophisticated mechanical drive system of Q offers movements with many degrees of freedom (e.g. moving and rotating simultaneously). Only with extensive communication, distributed controlling and flexible interaction of all modules tasks - "Driving through a door without collision" - can be completed. Using this in experimental setups, virtual sensors (smart bumpers, ultrasonic etc.) can simulate obstacles or disturbances for testing control algorithms. So the real environment with a robot system can be superimposed by virtual elements. After testing, the subscribers of the virtual environment are switched off and the robot works correctly without any changes in software. Complex moving patterns, path planning and so on can therefore be tested without the danger of physical damage or increased mechanical stress.

References

[1] Xenomai: Real-time framework for linux.

[2] M. Gergeleit and H. Streich. Implementing a distributed high-resolution real-time clock using the can-bus. In *Proceedings of the 1st International CAN-Conference. Mainz, Germany*, 1994.

[3] J. Kaiser and C. Brudna. A publisher/subscriber architecture supporting interoperability of the can-bus and the internet. In *2002 IEEE International Workshop on Factory Communication Systems*, Väesteras, Schweden, August 28–30 2002.

[4] J. Kaiser, C. Brudna, C. Mitidieri, and C. Pereira. COSMIC: A middleware for event-based interaction on CAN. In *9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2003)*, Lisbon, Portugal, September 2003.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[6] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[7] H. Piontek and J. Kaiser. Self-describing devices in cosmic. 2004.

[8] Robert Bosch GmbH. *CAN Specification Version 2.0*. 1991.

[9] ZigBee Alliance. *ZigBee Specification - IEEE 802.15.4*. 2003.

Index of authors

Allani, Mouna, 14

Benmouffok, Lamia, 58
Blair, Gordon, 26
Brønsted, Jeppe, 62
Busca, Jean-Michel, 58

Costa, Paolo, 48
Côrte-Real, José, 44

Felber, Pascal, 8

Garbinato, Benoît, 14, 38

Holzer, Adrian, 38

Joosen, Wouter, 68

Kristensen, Mads, 32
Kropf, Peter, 8
Kummer, Raphaël, 8

Leggio, Simone, 20
Lin, Shen, 26
Luo, Jun, 38

Mascolo, Cecilia, 48
Matthys, Nelson, 68
Michiels, Sam, 68
Miranda, Hugo, 20
Mocito, José, 44
Musolesi, Mirco, 48

Nguyen, Tuan Dung, 54

Patrick, Eugster, 38
Pedone, Fernando, 14
Picco, Gian Pietro, 48

Raatikainen, Kimmo, 20
Rodrigues, Luís, 20, 44
Rouvrais, Siegfried, 54

Schulze, Michael, 74
Shapiro, Marc, 58
Stamenković, Marija, 14

Taïani, François, 26

Verbaeten, Pierre, 68

Zug, Sebastian, 74