DAEDALUS – A Peer-to-Peer Shared Memory System for Ubiquitous Computing

Peter Ibach[1], Vladimir Stantchev[2], and Christian Keller[1]

[1] Lehrstuhl Rechnerorganisation und Kommunikation, Institut für Informatik, Humboldt-Universität zu Berlin
Unter den Linden 6, 10099 Berlin, ibach—keller@informatik.hu-berlin.de
[2] Net Business Center, Fachgebiet Systemanalyse und EDV, Technische Universität Berlin, Franklinstrasse 28/29, 10587 Berlin, Vladimir.Stantchev@sysedv.tu-berlin.de

Abstract. Data sharing in a large scale and for high volatility tolerance requires peer-to-peer solutions where traditional multiprocessor shared memory systems are not applicable. Efficiency of those P2P shared memory systems depends, in particular, on scale, dynamics, and concurrent write accesses. We have developed a P2P shared memory solution, DAEDALUS, based on SUN's JXTA framework, and integrated an efficient stochastic locking protocol, proper resource clustering, and semi-hierarchical grouping of nodes. We evaluated the applicability under heavy load, scale, and node mobility. Here, DAEDALUS outperformed a client/server system and solved its inherent scalability problem.

1 Introduction

Shared memory systems provide the foundation for efficient development of distributed applications. A lot of mature shared memory solutions for multiprocessor systems exist. However, data sharing in a large scale and for high volatility tolerance – typically occurring in ubiquitous computing scenarios – is still unaccomplished and has become an active field of research. Under such conditions, peer-to-peer architectures provide advantages over traditional distributed architectures with classical shared memory approaches. On the other hand there are numerous shared memory systems that are well designed for large amount of write accesses, but those are usually intended for supercomputers or cluster computing. However, none of these systems fully cover issues arising in ubiquitous computing scenarios where network topology and quality of service parameters are subject to frequent changes. Providing a synchronized and consistent view on the shared data for all participants with reasonable communication overhead, accordingly, is challenging and requires proper utilization of caching, routing, grouping, data compression, cryptography, forwarding, and consensus.

Therefore, we have developed DAEDALUS, a platform-independent and lightweight framework for peer-to-peer communication. It enables mobile/embedded devices to easily and efficiently share their data. Data may be distributed among thousands of peers and subjected to permanent updates, while further dynamics induced by mobility or environmental changes remain transparent to users or application developers. Devices may join or leave groups in an ad-hoc manner and can be members of an arbitrary number of groups at the same time, while our framework keeps the data *stochastically* in sync and consistent among all members of any group even in case of numerous concurrent write accesses. Our approach therefore uses *stochastic locking* and *semi-hierarchical grouping*. The implementation is based on SUN's JXTA Java classes for peer-to-peer communication. As a case study, we have integrated it in MagicMap, a cooperative WLAN positioning system. The client/server communication here did not scale well and required reliable connectivity. Both problems could be successfully solved using DAEDALUS, which achieved significant improvements regarding dependability, scalability and performance.

2 The MagicMap Application Scenario

MagicMap is a cooperative context aware computing application we introduced in [4–6]. Every node senses its environment and uses the observed data to calculate its location and situation. From that, location/situation specific actions can be triggered. The system works cooperatively, i.e., nodes exchange their measurements among each other. Calculations can be done redundantly on multiple nodes to improve fault tolerance, in particular, to prevent a minority of malicious nodes to affect system stability. In our current implementation we use WLAN equipped Laptops, PDAs, and Smartphones that exploit WLAN signal strength to sense the environment and calculate their positions (see Fig. 1). Nodes sense the WLAN received signal strength (RSSI) of neighboring nodes (access points, other clients, or previously measured reference points) and estimate the physical distance. A *spring layout* algorithm moves the nodes with unknown positions such that length of edges best match the calculated physical distance. Thus, the graph converges to a "magic map", where nodes are located approximately at their true physical position.

Since different nodes may calculate devices positions, the calculating nodes need access to signal strength measurements. Consider the following scenario shown in Fig. 2. Node C wants to know the position of node B. Node C, as well as node B, has low processing capabilities. Node A has

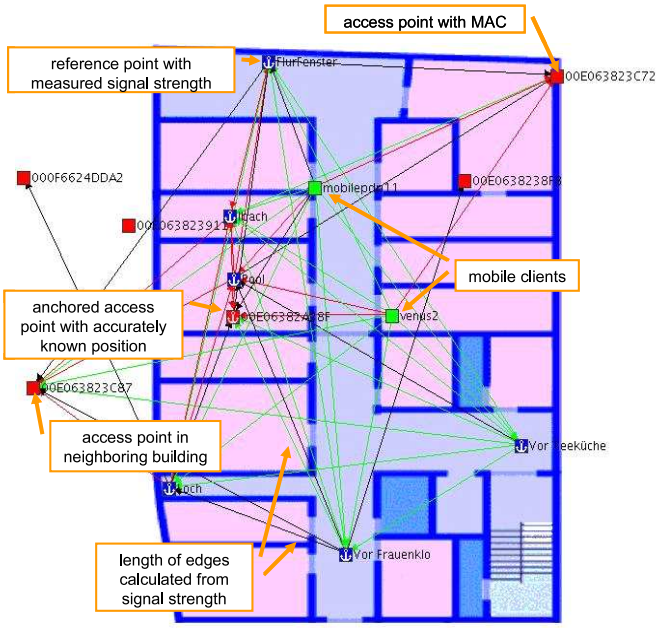

Fig. 1. MagicMap screenshot

high processing capabilities and therefore calculates the position of node B. Node E, being sufficiently capable as well, does that calculation too for redundancy reasons. All WLAN-aware nodes sense signal strength (1) and forward it to the nodes where calculation is done (3).

Note, that in this scenario we assume the mobile clients A, B, and C to sense the signal in a symmetric manner, i.e., A senses signal strength from B and symmetrically, B can sense signal strength from A. Some nodes may not sense the signal, in our case node D, which might be an access point or a peer node without MagicMap installed. However, given D is using its WLAN interface, its radio signals can be sensed by other nodes (2).

Finally, the calculated positions are sent to node C (4) who then can use, for example, the mean value of both calculations as best position estimation. In case C receives three or more independent position estimations, it could employ elaborated voting algorithms for improved fault tolerance and resilience against malicious behavior. To provide a real-time picture, signal strength measurement and position recalculation is done periodically at least every 10 seconds. Obviously, this scenario implies

significant performance and real-time demands: all measured values need to be on time at the nodes calculating the positions, and finally, all calculated positions need to be on time at those nodes, that have interest in this information.

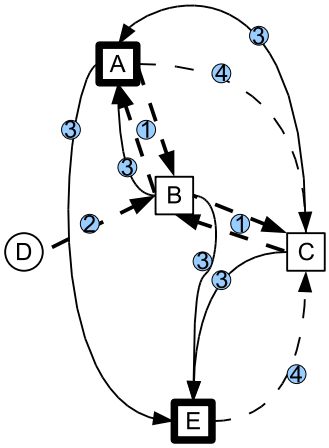

Fig. 2. Example scenario with high-performance nodes A and E and low performance nodes B and C

3 Peer-to-Peer Data Sharing Concepts

Several research projects emerged in the last years investigating efficient data updates in peer-to-peer systems. However, they impose limitations that reduce their usefulness in ubiquitous computing scenarios. Systems like Freenet [3], OceanStore [8], or P-Grid [1] assume no conflicting writes, going as far as limiting updates to the original author of a data item in Freenet. Ivy [7] requires application-level programming to cope with conflicting manipulations of data objects and only provides some tools to detect those conflicts. These systems do not provide any locking mechanisms or other concurrency protocols since their main purpose is to provide high scalability – at the costs of sacrificed consistency. Systems such as JuxMem [2] take the opposite approach: they provide locking mechanisms while limiting the size of the network.

3.1 Concurrency Control – Pessimistic and Optimistic Approaches

There are two opposed approaches for concurrency control, the pessimistic and the optimistic one. The first assumes that conflicting write accesses to a data item might cause intolerable inconsistency and thus have to be avoided anyway (conflict prevention). To guarantee that no other node is performing a concurrent write access to any replica of a data item, a node has to lock that item to prevent it from other concurrent manipulations. In a distributed scenario, this requires two-phase locking, i.e., the node has to wait for all item replicas to confirm the lock request. After the write has been performed, all replicas have to be updated accordingly to obtain a consistent state. Meanwhile, since the data might be temporarily inconsistent, additional write or read accesses to it are not allowed.

Pessimistic locking, hence, is not applicable in highly dynamic networks where typically presumed latencies cannot be guaranteed.

Therefore, an optimistic approach, in contrast, assumes that temporary inconsistency resulting from concurrent writes to a data item is tolerable. It employs conflict resolution instead of the above conflict prevention. Optimism is accounted for the assumption that the number of actual conflicts and resolving them will be manageable and temporary inconsistencies will be rare. If, however, a conflicting update occurs, nodes have to use roll back or roll forward mechanisms to resolve inconsistency and recover a consistent system state. An example is Ivy, which stores the history of operations that have been performed on the items. It does not resolve conflicting updates, but it detects them and provides application-level means to resolve them.

Instead of pessimistic or optimistic conflict handling, its also possible to create a disjoint global storage space, such that conflicts cannot occur. Freenet, for example, combines keys for files with a private key, specific to a user, and thereby creates a global name space with private subspaces. This however, would result in unmanageable network traffic and does not fit the MagicMap scenario where every device can publish estimates of other devices' positions.

Since none of the above approaches seem appropriate for our purpose, we have employed a hybrid approach (see Section 4.2).

3.2 Considering Different Node Capabilities

All above systems assume the peers to possess comparable capabilities. This assumption, although it might be acceptable in workstation environ-

ments, is unrealistic in heterogeneous networks of ubiquitous computing. Therefore, caching a snapshot of the overall storage system as required by Ivy is only feasible for very small distributed file systems. OceanStore does allow multiple nodes to change a single data item. To prevent faulty nodes from publishing wrong version information, a Byzantine agreement is formed between all primary replicas. OceanStore however, as well as P-Grid, does not offer means to prevent or resolve conflicting write accesses to the same data item. Since MagicMap updates position information rather frequently, such peer-to-peer systems are likewise not appropriate.

4 The DAEDALUS Peer-to-Peer Shared Memory System

The system architecture is divided into platform dependent and platform independent components, see Fig. 3. Measurements of signal strength and collection of other sensor data is highly dependent on particular hardware, operating system, and drivers. The platform independent components – in particular the DAEDALUS shared memory and the normalization and calculation of position estimations – are written in Java. All components are freely available via our website `www.magicmap.org`.

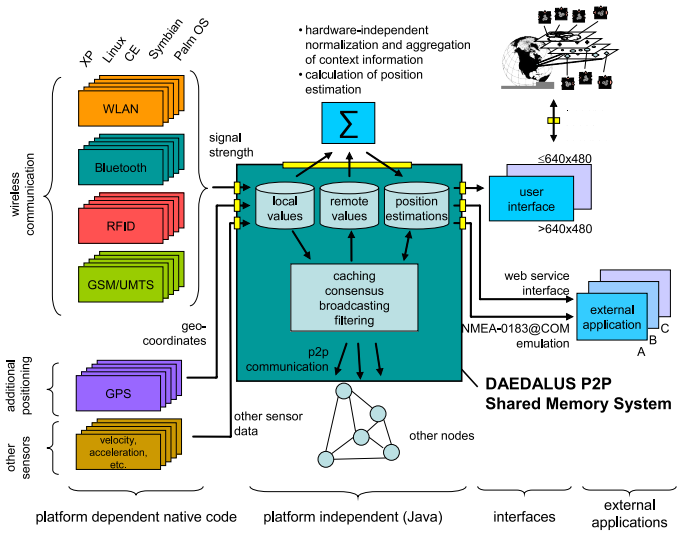

Fig. 3. System architecture with platform dependent and platform independent components communicating via the DAEDALUS peer-to-peer shared memory

4.1 Peer Groups

The basic idea of our shared memory system is to assign every data item a specific peer group. Peers that have interest in this data item join the related group and serve as a replica. The advantage of this approach is scalability. Thereby, the amount of messages send does not depend on the number of nodes participating in the entire system, instead it depends on the number of peers interested in this data item. While the number of nodes in a network could become rather big in real world scenarios, the number of peers interested in a specific data item is limited. The idea, however, has a downside: once no peer is interested in a data item, it will be lost. To prevent this, nodes having enough resources to join multiple groups in parallel will be asked to join this group, in case the number of member nodes is decreasing below certain threshold. As these groups still can grow rather big, a further differentiation is needed. A percentage of all nodes in this group acts as a manager. Managers act as replicas, vote on locking requests, and keep track of the group size.

4.2 Stochastic Locking – a Hybridization of Pessimistic and Optimistic Concurrency Control

Since both, pessimistic and optimistic approaches are not feasible in our scenario, we pursue a hybrid approach. We use a locking mechanism but we do not require all nodes to answer a lock request. Instead, only a relatively small number of nodes has to answer and broadcast their decision to all managing nodes in a group as shown in Fig. 4. The requesting node has successfully locked a data item, if a majority of those answers is positive. This approach is optimistic, as it assumes that enough nodes receive the lock request messages and there are only a few faulty nodes that give an insane answer regarding a request. It is as well pessimistic to a certain degree, as it reduces the number of conflicts by locking a resource before updating it. While this *stochastic locking* cannot *guarantee* that no conflict occurs, it does provides a high probability of conflict prevention.

4.3 Scalability Considerations

As only a fixed number of managers is required to answer a client request, the expected traffic for each update process is limited and known. However, in order to ensure that the number of managers answering a request does not exceed the threshold, the managers have to keep track how much of them are in a group. Therefore, every peer joining a group broadcasts

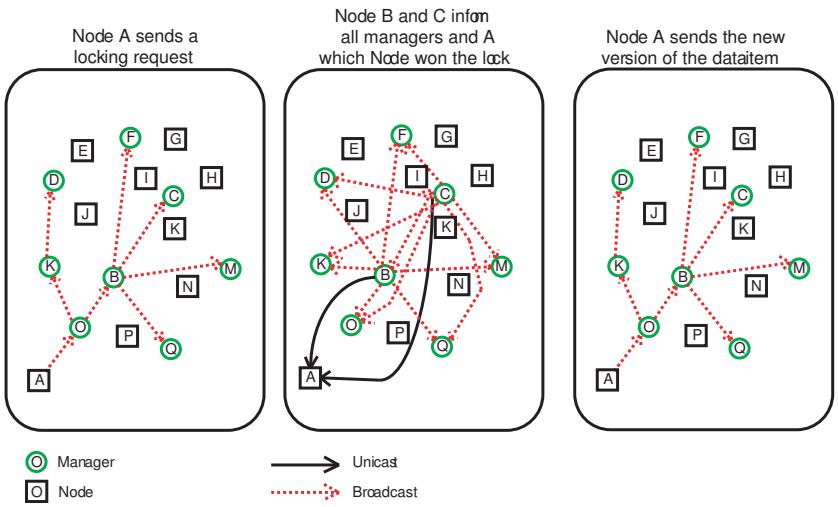

Fig. 4. The locking process. Node A sends a lock request to all manager nodes. Of those manager nodes B and C respond and broadcast their decision to all other managers.

a hello packet to all managers. A fixed number of managers will provide the new member with all necessary information, such as group size and a list of managers. Every peer node joining the group starts as a manager. If the peer later discovers that there are already enough managers it can alter its status and become a regular client. Additionally, managers check whether there are still enough managers in the group, and ask clients of the group to become managers, if the number falls below the threshold. On the other hand, if a manager detects that there are not enough members in its group, they call other nodes that still have enough resource capacity available to enter the group.

While the load of a client is independent from the number of nodes in the group, the load of managers does grow with the size of the group. For a single process the load is constant. However, as only the request for data items can be balanced over all managers, the load for writing, locking and counting is not. Therefore, the number of messages a manager has to process increases linear with the number of nodes in the group. This however does not compromise the original goal of low load for small computing devices. As the chance for such a device to be a manager decreases with group size, the load for small devices will not grow beyond a point which depends on the ratio of small and large nodes within it.

4.4 Integration into MagicMap

We implemented the peer-to-peer shared memory system as a Java application which communicates with any local application via UDP datagrams. It supports calls to read the data item of a given name, to store a new version and to lock and unlock the data item. Additionally, we included calls to search for groups and peers. As group names are the same as their data item's name, a search for all groups will result in a list of all available data items. By applying a name scheme an application can easily search for all data items it needs. We have developed one call specifically for MagicMap: joining a specific group. We use this call to create a hierarchical tree that stores all nodes and their positions.

4.5 Data Clustering

To keep management overhead reasonable, the data items have to be clustered appropriately. One clustering option is to subsume all external measurements according to each node and store it in a single data item. This would allow the position calculating nodes to easily discover the relevant data. However, it would increase the number of groups that each node has to join and would cause frequent locks and updates to data items.

This made the alternative option – aggregating all values measured by the same node – most promising to us. As only a single node will change the data item, no locking is required. However, now the calculating node has to find all other nodes that have measured the signal strength of the node to be located. To make the discovery process feasible, we decided to add a data item for each node to store a list of the nodes having measured its signal strength. Thus, the calculating node can scan the list and find all the data items required to calculate the node's position. As this node list has to be updated by different nodes, locking is required. Fortunately, the number of updates to the list typically remain in a manageable amount.

The position values for each node are stored in a single data item. As there are typically less than five nodes actually updating this data item, this does not cause heavy load. We end up with three data items for every MagicMap node. For a node A there are *A-Measurements* where this node stores all signal strengths it sensed, *A-See* stores all nodes that sense signals from A and *A-Position* contains the calculated position of this node. A node that wants to know the position of node A accesses *A-Position*. If no other node has yet calculated the position and the data item is empty, this node may want to calculate the position itself. To do

so the node first reads *A-See* and then accesses all measurement data items of the nodes in this list.

5 System Evaluation

We conducted our tests using the MagicMap application as a case study and compared the delay of data item updates in the client-server setup to the DADALUS peer-to-peer setup at different numbers of participating nodes (see Fig. 5).

#Nodes	C/S Avg. Delay	C/S Standard Deviation	P2P Avg. Delay	P2P Standard Deviation
10	2.5 s	11 s	1.0 s	461 s
20	3.5 s	12 s	1.6 s	810 s
40	7.5 s	13 s	2.9 s	1,103 s
80	12.5 s	14.5 s	4.1 s	1,221 s
120	-	-	4.2 s	1,069 s

Fig. 5. Comparing the data update delay of the standard client-server and the DAEDALUS peer-to-peer setup

In the client-server setup, updates were done via a centralized server using Web Service communication. The peer-to-peer setup utilized JXTA broadcast/unicast and comprises locking the data item, updating it, and finally releasing the lock.

For both setups we employed 8 Dell PDAs as "low capable" nodes and 8 desktop computers as "high capable" nodes and simulated further nodes. The ratio of low to high capable nodes was kept at constantly 1:1. We tested each setup for a period of 6 hours and repeated the measurement three times at different days. While we consider the obtained result quite realistic, true real world measurement with heterogeneous devices in a magnitude of hundreds or even thousands of nodes have to remain for future work.

6 Conclusion and Outlook

We have proposed a peer-to-peer shared memory system designed for ubiquitous computing scenarios. It provides stochastic locking and data

clustering to arrive at reasonable performance even at high scale and dynamics. In our WLAN positioning case study implementation we used relatively well equipped Dell PDAs and measured performance parameters. Using these measurements, we further investigated scalability and other quality of service issues by simulation. The results indicate that, regardless of group size, 95% of all data updates will not take longer than 6 seconds, provided that no conflicting writes occure.

Future work may integrate a way to preserve multiple versions of a single data item. Also a privacy scheme has to be developed to protect data and improve system acceptance – since user locations are definitely very sensitive information.

References

1. Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. P-grid: a self-organizing structured p2p system. *SIGMOD Record*, 32(3):29–33, 2003.
2. Gabriel Antoniu, Luc Boug, and Mathieu Jan. Juxmem: An adaptive supportive platform for data sharing on the grid. *Scalable Computing: Practice and Experience*, 6(3):45–55, November 2005.
3. Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009, 2001.
4. P.K. Ibach, T. Hübner, and M. Schweigert. MagicMap - Kooperative Positionsbestimmung über WLAN. In *Chaos Communication Congress*, Berlin, Germany, December 2004.
5. P.K. Ibach, F. Schreiner, V. Stantchev, and H. Ziemek. Ortung drahtlos kommunizierender Endgerte mit GRIPS/MagicMap. In *35. Jahrestagung der Gesellschaft für Informatik*, Bonn, Germany, September 2005.
6. P.K. Ibach, V. Stantchev, F. Lederer, A. Wei, Th. Herbst, and T. Kunze. WLAN-based Asset Tracking for Warehouse Management. In *IADIS International Conference e-Commerce*, Porto, Portugal, December 2005.
7. Athicha Muthitacharoen, Robert Morris, Thomer Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.
8. Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: The OceanStore prototype. In *Proceedings of the Conference on File and Storage Technologies*, pages 1–14, March 2003.