A Self-Organizing Lookup Service for Dynamic Ambient Services

Klaus Herrmann*, Gero Mühl*, Michael Jaeger†
Berlin University of Technology
Institute for Telecommunication Systems
EN6, Einsteinufer 17, D-10587 Berlin, Germany
{klaus.herrmann, g_muehl, michael.jaeger}@acm.org

Abstract

The provisioning of Ambient Services *is gaining importance as users become more and more embedded in environments that are saturated with electronic devices. In our previous work, we have proposed the* Ad hoc Service Grid *(ASG) approach as a means for deliberately setting up an infrastructure for providing Ambient Services at medium-sized locations like shopping malls. In this paper, we introduce a self-repairing Lookup Service architecture that is able to handle dynamic services in an AGS. These services can autonomously replicate and migrate within an ASG to optimize resource usage and message latency. Our Lookup Service is able to cope with the inconsistencies caused by service migrations efficiently by applying a lazy, request-driven update protocol. We discuss the architecture and the protocols employed by the Lookup Service. Moreover, we provide experimental results that show the efficiency and effectiveness of our approach.*

1 Introduction

Mobile computing technologies pervade our every-day life with increasing speed. The mainstream in the area of mobile service provisioning is represented by cellular phone networks and WLAN hot spots. Both technologies currently enjoy wide-spread commercial usage. While cellular networks are used to cover wide areas with services that are meaningful on a global or regional scale, WLAN provides Internet access on a small scale (restaurants, airport lounges etc.). However, a third technology builds on ad hoc networking (MANET) principles [1] to seamlessly provide services that are hidden in our immediate environment. These are called *Ambient Services*. Typically, Ambient Services enable the user to interact with resources in his vicinity. One example is a user using the nearest printer without prior configuration. Other scenarios foresee more complex services that support the user's actions at his current location in an unobtrusive way.

We have proposed a specific infrastructure for providing arbitrary Ambient Services at medium-sized locations like shopping malls, hospitals, and construction sites [2]. This infrastructure together with the supporting software platform for enabling service provisioning is called *Ad Hoc Service Grid* (ASG). An ASG uses MANET technology to connect individual PC-class devices (called Service Cubes) dispersed over the location. Together, the Service Cubes provide the basic infrastructure and the resources for service provisioning. The communication within this infrastructure is free of charge and at the same time the installation is quick and easy. Moreover, through the addition or removal of Service Cubes during runtime, the ASG can be scaled up or down with minimal effort to fit the current requirements of the facility. However, the highly distributed nature of this approach, requires that an ASG has to self-organize many aspects of its operation. In particular, an ASG service must be able to find the best Service Cubes for its execution autonomously. Additionally, it needs to be able to replicate dynamically to meet the current demand encountered in the ASG. Therefore, ASG services are mobile and replicated.

In this paper, we present the architecture and the protocols of the ASG Lookup Service. This service is distributed in nature and employs *lazy propagation* to disseminate lookup information among the distributed lookup service instances. The result is a system that can deal with service location information which is inconsistent and outdated due to service migration and replication. We show that our update protocol is self-repairing with respect to temporarily inconsistent lookup information. Moreover, our lazy, usage-driven update propagation minimizes network load by updating only those parts of the network that are currently used to access services. Other regions remain outdated until they actively take part in interactions between clients and services. We argue that this *soft definition* [3]

*Sponsored by Deutsche Telekom
†Sponsored by Deutsche Telekom-Stiftung

of what a *correct* or *acceptable state* of the overall lookup system is, makes the system flexible and adaptable while clients still have hard guarantees to rely upon.

The rest of this paper is organized as follows. In Section 2 we discuss other lookup systems and explain their shortcomings with respect to service dynamics. Section 3 introduces the Ad hoc Service Grid infrastructure. The Lookup Service architecture, its components, and its protocols are detailed in Section 4. Section 5 covers several possible failure situations and shows how the lookup protocol recovers from them to preserve its self-repair feature. Experimental results, that show the effectiveness and efficiency of the lookup system, are presented in Section 6. In Section 7, we discuss open issues and possible drawbacks of the current system. Finally, Section 8 concludes the paper.

2 Related Work

Any service infrastructure that involves a certain degree of dynamics has to provide some means for discovering and looking up services. Therefore, a wide variety of different lookup architectures have been proposed and implemented by research and industry. The most prominent industrial standards are Jini [4], UPnP [5], and Salutation [6]. Jini was designed to support nomadic, Java-enabled environments where mobile devices join an existing network and use services in an ad hoc manner. A Jini lookup service is discovered using multicast. It may offer Java-based service interfaces that are downloaded to a mobile device in order to interact with a given service. Jini lookup services can form a hierarchy and requests may be passed up this hierarchy for resolution. Universal Plug and Play (UPnP) is a framework defined at a much lower level than Jini. It offers IP address allocation and DNS name assignment for mobile devices and builds, for example, on DHCP. UPnP's Simple Service Discovery Protocol (SSDP) supports registration and discovery of devices. This may involve dedicated directory services but does not rely on them. In Salutation, devices use a Salutation Manager (SLM) for the lookup process. SLMs may exchange registration information and support clients by mediating data transport that covers different transport protocols. A client queries a near-by SLM in a similar way as is done in the ASG Lookup Service. However, none of these industry standards support service replication and mobility explicitly. The dynamics covered by these approaches is related to *physical mobility* (device mobility) rather than *logical mobility* (software mobility).

In mobile agent research, several lookup mechanisms have been devised that explicitly support logical mobility [7, 8, 9]. They either use brute force mechanisms to find a mobile agent, log current positions at a more or less centralized server, or they set up forwarding chains to follow the route taken by agents [10]. However, maintaining replicated agents and mediating adequate replicas upon client requests is not an issue in this area.

3 Ad hoc Service Grid Architecture

WLAN access points and cellular phone networks present two wide-spread technologies for providing mobile services to customers. While cellular phone networks are ideal for covering wide areas with services that may also be location-based, WLAN is popular for setting up local hot spots with a very limited geographical extension. Thus, large-scale as well as small-scale coverage are relatively straight forward. However, there are locations that range between these two extremes in terms of scale. Such *medium-scale locations* include, for example, shopping malls, hospitals, and construction sites. These locations may benefit from a complete coverage using wireless access technology in order to provide services to users. Typically, such services are provided by local providers (e.g. shop owners) and accessed by local users. Thus, the bulk of communication between clients and services is inherently local. Cellular phone networks introduce unnecessarily expensive, global communication via some network provider with a relatively low bandwidth. On the other hand, a coverage based on access points requires these devices to be wired to some existing infrastructure. This introduces enormous costs that often exceed the costs for the wireless equipment by far.

To fill the gap that exists between large-scale and small-scale coverage, we have introduced the *Ad Hoc Service Grid* (ASG) [2]. In an ASG, inexpensive, PC-class devices are dispersed over a medium-scale location (in the order of a few thousand square meters). These devices (called *Service Cubes*) have a power plug and a wireless network interface that is capable of setting up an ad hoc network connection to the devices in its vicinity. The Service Cubes (also called *Cubes* or *nodes* in the remainder of this paper) represent the modules of the ASG infrastructure providing both the access network and the computing resources for mobile location-specific services. Service Cubes may be dynamically added or removed to scale the system according to the demand. The control and management of this infrastructure is difficult due to its dynamics and distributed nature. Therefore, a software platform (called *Serviceware*) is required that enables the ASG to self-organize different aspects of its operation. In the rest of this section, we explain the key aspects of the ASG infrastructure and its Serviceware before we turn to the ASG Lookup Service.

3.1 Dynamic Service Positioning

A number of uncertainties in the ASG environment prevent services from being placed in fixed, predefined positions. Even if a Service Cube is chosen that seems to be

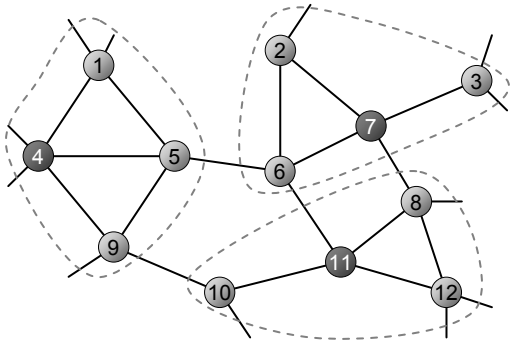

Figure 1. Clustered network topology: Three clusters with cluster heads 4, 7, and 11.

ideal for a given service replica, the addition of new nodes, for example, may render it suboptimal over time. Moreover, the request patterns generated by users may change. Thus, a single service placed statically on one node may lead to network congestion and high latencies perceived by users, if their requests have to be routed through the whole network to reach the service.

Therefore, we introduced different dynamic service positioning algorithms that are able to migrate and replicate services during runtime to react to changes in user demand [2]. These algorithms essentially reduce the distances between requesting clients and services. Thus, the network load is reduced and the processing load is shared between the individual services through replication. This also reduces the latency perceived by users.

The fact that services may replicate and migrate at runtime poses special requirements on the ASG Lookup Service. It has to be able to deal with this dynamics and still mediate valid information on how to access services.

3.2 Clustered Network Topology

In order to enforce a basic structure on the otherwise unstructured ASG network, we adapted a clustering algorithm proposed by Basagni [11]. This distributed algorithm elects cluster heads and partitions the network into small clusters (see Figure 1). As a result, each ordinary node (none-cluster-head) is directly connected to its cluster head. Cluster heads play a special role in the ASG Serviceware since they provide infrastructure services like the Lookup Service.

The clustered topology is also used to simplify the routing of messages. Each node has a unique ID and a node's address consists of its node ID and the ID of its cluster head. For example, node 1 in Figure 1 has the address "1.4". The routing tables hold a next hop node for each cluster head ID and the node ID in the destination address is used by the destination's cluster head to find the node within its cluster.

3.3 Middleware Model

We employ the basic middleware mechanisms of `MESHMdl` [12]. This middleware for mobile ad hoc networks builds on the abstraction of mobile agents. A `MESHMdl` application consists of a group of interacting agents that may use their mobility to adapt to dynamic changes in their environment. Moreover, `MESHMdl` offers an asynchronous communication mechanism that is based on tuple spaces [13] to decouple applications and render them more autonomous and less susceptible to the destructive effects of device mobility.

3.4 Service Properties

The services running in an ASG are not limited in their complexity. It should be noted that it is not mandatory for a service to be able to replicate or migrate at all. The decision to enable these features is up to the service programmer. The Serviceware provides a mechanism and protocols for achieving data consistency among the replicas of stateful services. It implements an eventual consistency model that is inspired by the one found in Bayou [14]. An ASG service consists of an arbitrary number of replicas that have the same *service type*. A *Customer Navigation Service* and a *Product Information Service* in a shopping mall scenario would be two different service types. Several replicas may exist for each of them.

3.5 Client Access

Client applications (simply called *clients*) are running on the mobile devices of users. These clients may be preinstalled on the user's devices or they may be downloaded on-demand from the Lookup Service in a similar way as it is done in Jini [4]. We assume that, as the user moves through the ASG, his device can always connect to at least one Service Cube. Through one of these Cubes, the client may access any services running in the ASG irrespective of its location.

4 Lookup Service Architecture

One instance of the distributed Lookup Service is running on each cluster head. Consequently, each ordinary node has a Lookup Service running on one of its neighbor nodes. Wherever a client connects to the ASG network, it can quickly discover services, lookup service replicas, and download client applications. However, this also

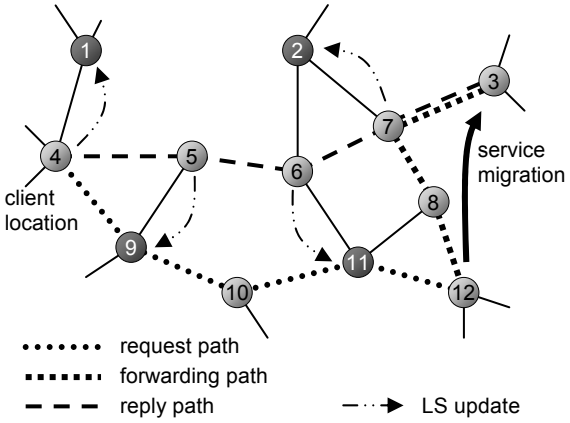

Figure 2. Request-driven update process.

means that information about the positions of service replicas needs to be disseminated across the Lookup Service instances to make it available locally when needed. Since the ASG supports service replication and migration, this poses special problems not present in other lookup systems like Jini, UPnP, or Salutation.

4.1 Update Process Overview

Before we go into the details of the architecture and its elements, we give a high-level overview of the basic mechanisms employed. In the ASG, there is a fundamental trade-off between the network load caused by Lookup Service updates and the accuracy of the information provided by the Lookup Service. Keeping all Lookup Service instances up to date all the time causes a large message overhead. Not doing so may cause clients to fail because they rely on outdated location information. To solve this problem, we employ a lazy, request-driven update strategy.

Figure 2 depicts the basic idea of the update scheme. We assume that, initially, a service replica S is running on node 12 and a requesting client C is connected to node 4. At that point, the Lookup Services at the cluster heads 1, 2, 9, and 11 have the correct location of S. This information gets outdated as S moves from node 12 to node 3. A reason for this migration could be a shift of request load that can be compensated by moving towards this direction. However, for the update protocol, this is not relevant. Immediately after the service migration, C still has the old service location information and continues sending its requests to node 12. A so-called *Proxy Agent* on node 12 has been informed by S about its migration before S left. This Proxy forwards the requests to the new location of S (node 3). The Proxy stores meta information in the forwarded requests, marking them as *forwards* and containing the new location of S. S gets a forwarded request and generates a reply. The meta information from the request message is automatically copied into the reply. After that, S sends the reply back to C. As depicted in Figure 2 the reply message may take a different route than that traveled by the request. As the reply is routed through the network, the so-called *Lookup Snooper Agents* on that route inspect the message and extract its meta information. One Lookup Snooper Agent is running on each node. When such an agent detects a service reply message whose meta information marks it as belonging to a forwarded request, it sends a *service location update message* to its cluster head (depicted as arrows in Figure 2). This message is read by the cluster head's Lookup Service, which in turn updates its location information base with the new location of S. Thus, triggered by the reply, all Lookup Services along the way, including the one used by the requesting client C on node 1, will eventually be updated. We assume that C regularly requests the current location of S at the Lookup Service. As a consequence, after some requests have been forwarded by the Proxy Agent on node 12, C finally gets the updated information and sends its requests directly to node 3. At that point, normal operation is restored and the Lookup Services at nodes 1, 2, 9, and 11 are up-to-date again.

Note how only those parts of the lookup infrastructure are updated that are close to the reply path. Lookup Services that are further away, are not involved. They will only be updated when they are directly involved with requests sent to S. Moreover, the update information is carried by reply messages. Only a small number of additional messages are necessary ("LS update" arrows in Figure 2, also called *snooping messages* hereafter). In this way, the network load caused by update messages is minimized, but at the same time, the active parts of the network are kept up-to-date. This lookup architecture is self-repairing since the lazy updating process is active throughout the network at any time, and it is automatically triggered only if location changes occur. The Lookup Services themselves are only passively involved in this process.

An active propagation phase precedes this process of continuous self-repair. This active propagation is triggered when a service replica is initially started. After its start, S registers with its local Lookup Service, that in turn propagates this new registration once to all other Lookup Services to establish this information throughout the system. This is necessary since a Lookup Service needs at least the information about which replicas are running to be able to return the most adequate one upon a client query. This initial propagation uses a *cluster head broadcast* mechanism offered by the routing system. A similar propagation is done to remove information from the system when a service replica dissolves (removes itself).

It should be noted that even in remote regions of the network, where no update arrives over extended periods of

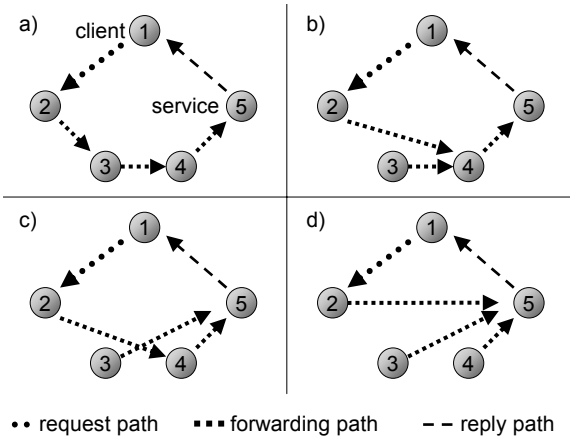

Figure 3. Establishing forwarding shortcuts.

time, the initial, outdated location information is sufficient to find a service replica (via forwarding) and update its location. This replica may be much further away then initially estimated by the Lookup Service. This, in turn, is likely to cause a different replica to be chosen if the client poses the next query. After a few of these cycles, the local Lookup Service has sufficiently up-to-date information to return an appropriate replica to the client.

4.2 The Proxy Agents

Even though the Proxy Agents are not merely dedicated to the lookup update process, they play an important role. At each node one Proxy Agent is responsible for relaying requests to services, analyzing request patterns, triggering migrations or replications, and forwarding requests to services that migrated away. The forwarding process may involve more than one Proxy if the service has gone through several migrations. Each Proxy only has the information about the service's next location. When it forwards a request, the Proxy at that next location may also have to forward it because the service has already left for the next node. Thus, a chain of forwards may occur. Note that forwarding loops can be avoided by removing any existing forward for a service S if S revisits a node.

An simple additional mechanism reduces the length of forwarding chains. This can save transmissions and reduce latency. Figure 3 depicts how shortcuts can be made in forwarding chains that eventually eliminate any intermediate Proxies between the one receiving the original request and the one currently hosting the service. This can be achieved by introducing one additional message type. If a Proxy receives a forwarded message from the previous Proxy in the chain, it sends a *redirect forward message* back to the sending Proxy to set that Proxy's next hop to its own next hop. For example, node 3 in Figure 3a sends a *redirect forward message* back to node 2 when it receives the forwarded request. The redirect contains the next forward hop of node 3 (which is node 4). Now, node 2 uses node 4 as its next hop and bypasses node 3 (Figure 3b). Next, node 4 redirects node 3 directly to node 5. Figure 3c shows the situation after the first request has been forwarded to the service's new location (node 5). When the second request is forwarded, node 4 redirects node 2 to use node 5 as its next hop. At that point, all forwards point directly to the service's new location and the chain is resolved (Figure 3d). This approach does not require any additional state in the forwarded messages. Moreover, it has the nice feature of reducing chains based on their usage. If a chain is not really used, no resources will be wasted to make it shorter. The more messages pass through it, the shorter it gets. An alternative would be to let the Proxy at the new service location send messages to all forwarding nodes and redirect them in one step. However, this requires the complete forwarding path to be put into the forwarded messages.

4.3 The Lookup Snooper Agent

A Lookup Snooper Agent (LSA) is running on each node and uses the snooping API provided by the middleware to inspect all messages that originate at or pass through the node. Its snooping filter matches all messages whose meta data marks them as *forwards* and as *service replies*. Every Proxy that forwarded the respective request, put the last known *service record*[1] of the requested service into the requests meta data. Thus, when the request eventually arrives at the service, it contains the correct service record. Upon snooping a reply message, a LSA extracts the *service record* and send it to its own cluster head where it is received by the local Lookup Service. The Lookup Service, in turn, replaces its outdated service record with the new one. To avoid multiple redundant messages, the LSA uses two mechanisms:

1. It inspects the route taken by the reply message and discards it, if it has already passed through one or more nodes in the same cluster. Since the first of these nodes has already sent the update to the Lookup Service, there is no need to resend it. If we assume that nodes 6 and 7 are both in the cluster of cluster head 2 in Figure 2, then node 6 does not send the update message to node 2 since it recognizes that node 7 must have already done so.

2. Each LSA keeps a limited history of service record uuids sent to the Lookup Service. If it finds a newly received record in this list, it does not resend it. For a network of 100 nodes, it turns out that a history of 10

[1] A service record is the data structure used by the Lookup Service to store all data pertaining to a registered service replica.

records suffices to avoid unnecessary retransmissions almost completely.

4.4 The Lookup Service

The actual Lookup Service is rather simple. A client may query it in one of two ways:

1. It may request a specific service replica via its unique identifier or
2. it may request a service type.

In the latter case, the Lookup Service will choose the *most appropriate* replica and return its service record to the client. In the current implementation, the "most appropriate" replica is always the one closest to the requesting client. However, it is also conceivable that other criteria are applied and that clients specify these criteria in their queries. When a service registers at a Lookup Service, a service record is created that holds all the information needed for accessing the service. A copy of this record is returned to a client as a result of a successful query.

4.5 Maintaining other Dynamic Data

The ASG Lookup Service is designed to handle dynamic data (mainly service replica locations) efficiently and effectively. As a by-product, it is straightforward to update and maintain additional dynamically changing data and to provide it to clients. For example, we use this to disseminate the current load experienced by the service replicas throughout the ASG network. Each service stores its load (e.g. number of requests processed during a certain time period) in the meta data of every n-th reply messages. The Lookup Snooper Agents listen for messages carrying load information and propagate it to their Lookup Services. This enables clients to choose services not only based on their distance, but also on their load.

5 Self-Repair Properties

We argue that the described system is self-repairing. That is, it can compensate the inconsistencies caused by service mobility and preserve a *legal state* without manual intervention. For the system's normal mode of operation, discussed in the preceding sections, this is obvious. However, in a system like the ASG, there are also several external influences that may lead to erroneous states beyond those caused by mobility. In this section we will investigate possible disturbances and explain how the system autonomously returns to a legal state. We start by defining the term "legal state".

Definition 1 *The state of the lookup system is* legal *if and only if any client query*

1. *yields location information that eventually leads to the proper delivery of requests to an adequate service replica, or*
2. *results in a respective notification if and only if no adequate service replica is present in the system.*

Any configuration that does not satisfy this definition is considered an *illegal state*. Note that this is a very weak definition. However, it offers a sufficient guarantee to clients by stating that their requests will arrive. For the client, it is indeed irrelevant whether the result of its query is outdated. A stronger definition is not required. For the lookup system, however, this weak definition does not enforce rigid and brittle mechanisms, and it leaves room for flexible solutions. The idea of *softening definitions* in software systems to avoid brittleness was first brought forward by Shaw [3].

5.1 Topology Changes

The ASG topology changes if a new node is added or a node is removed or crashes. Note that the clustering and routing algorithms are able to compensate such events at the networking layer. The mechanisms used to achieve this are well-known and will not be explained for spatial restrictions. The question that we discuss here is, how does such a change effect the lookup information in the system and how can it recover from illegal states caused by these changes?

If a new node is added and has a cluster head within transmission range, it simply joins this cluster head. This has no effect on lookup information. If the new node cannot join any existing cluster head, it has to become a cluster head itself and starts its own Lookup Service L. To get valid lookup information, it queries the local routing table for the closest other cluster head until it finds one that has an operational Lookup Service L'. Then, L requests the entire service table of L' and uses it. Even if this service table is not up-to-date, this information is sufficient for the request-driven update scheme to work properly. Thus, the system is in a legal state again.

We assume that there is a regular way of removing a node which gives the system the opportunity to prepare (e.g. migrate services to other nodes). A harder problem is occurring if a node crashes. If the crashed node v was a cluster head, then one of two situation may occur:

1. All the ordinary nodes that were in v's cluster before have other cluster heads as neighbors and can join these clusters. In this case, their addresses change. While the network routes will be fixed by the routing algorithm, the outdated addresses stored in Lookup

Services must be fixed by the lookup system. Therefore, the Lookup Service on each cluster head that integrates new nodes broadcasts the address changes to all other Lookup Services.

2. At least one ordinary node u that was in v's cluster before has no other cluster head as a direct neighbor. In this case, a new cluster head is selected that starts its own Lookup Service and updates its empty service table in the same way explained above for newly added cluster heads.

If the crashed node was an ordinary node, the cluster structure is not damaged. Thus, no such repair processes have to be started. However, irrespective of whether the crashed node was a cluster head or an ordinary node, there are basically two problems that may arise for the lookup system.

1. The node may have been part of a forward chain, which is now broken, or

2. the node may have hosted service replicas.

We will explain how to cope with both of these problems in the next two subsections.

5.2 Repairing Broken Forwarding Chains

A forwarding chain is broken if one Proxy in the chain does not have a valid next forward hop. Either this information is missing or the next forwarding hop is not reachable (has crashed). In addition to missing next hops, loops may occur in the chain if a service fails to remove old forwarding pointers. Missing forward pointers, loops and crashed nodes can be detected by the last valid Proxy in the chain. This Proxy sends a notification about the error back to the client that sent the request. This client, in turn, notifies its Lookup Service L about the problem[2]. Note that, a broken forwarding chain means that at least one service replica is not reachable using the information provided by L. Depending on the nature of the service, this may violate definition 1 of a legal state, if no other replica exists or if other replicas are not adequate. Therefore, L tries to fix the problem. It removes its invalid lookup entry for the unreachable service and broadcasts a *reset* message containing the replica uuid to all other Lookup Services (cluster head broadcast). Any Lookup Service that receives this message and does not know the whereabouts of the replica in question, also removes its entry for it. Only the Lookup Service that has the replica in its cluster reacts by propagating the correct information to all other Lookup Services. To avoid race conditions (propagation messages overtaking reset messages), each reset message contains a unique id. A propagation message that is sent as a response to a reset messages is tagged with the same id. Thus, if a replica receives the propagation message first, it is able to recognize that the respective reset message is missing, and it may ignore the message if it arrives later. Thus, the service's location is updated throughout the ASG. Note that the remaining forward pointers of the broken chain do not present a problem. Since all Lookup Services in the system have the correct location after the repair, none of these pointers will be used again. Moreover, if the service returns to one of these nodes, it will remove the old pointer anyway.

[2]This can also be made transparent for the client by introducing a client-side proxy that takes care of such faults. For the sake of simplicity, this is not the case in the current implementation.

5.3 Dealing with Service Crashes

If a service replica crashes due to a software error and its Proxy remains operational, then the Proxy can unregister the replica properly[3]. If, however, a replica vanishes, for example, due to a node crash, different measures have to take effect. To purge the system from location information pertaining to replicas that have crashed, each replica has to periodically send a *registration refresh* message to its local Lookup Service. If a Lookup Service does not receive this message for an extended period of time, it assumes that the replica has crashed and was unable to unregister properly. It reacts by broadcasting a *reset* message (see Section 5.2) to its fellow Lookup Services which will remove the replica completely from the lookup system if it has really crashed. However, if it resides in a different location (failed to unregister properly before leaving), the correct location information will eventually be propagated. If a replica is unable to send its *registration refresh* message (e.g. due to high load) and the Lookup Service accidentally assumes a crash, then the replica will eventually be able to send messages again, which results in a new system-wide registration.

5.4 Avoiding False Negative Query Results

If, for some reason, a Lookup Service failed to gather the information about a new service type being introduced in the system, it would falsely answers client queries negatively, stating that nothing is known about any replica of that type. This would clearly violate our definition of legal states. Thus, before a Lookup Service returns a negative query result, it starts an incremental search to verify its correctness. It queries the routing table to find all existing cluster heads and their hop distances from its own node. Then it starts sending queries for the service type in question, gradually increasing the radius of its requests. The assumption here is that in most cases a near-by Lookup Service will have the desired information. If, for example, the service

[3]Note that the replicas of a service may crash independently.

registration propagation for some reason failed at its source, this protocol will traverse the entire network. In this case, a normal cluster head broadcast would be much more efficient. Note that in both cases, malicious clients may use this feature to run denial of service attacks by repeatedly querying for unknown services to overload the network. Resolving these problems is subject to our current research.

6 Experimental Results

To evaluate the ASG Serviceware and in particular the distributed Lookup Service, we have conducted a simulation of an ASG network with 100 Service Cubes and three distinct service types. The clustering algorithm yields 23 clusters on average for a network of 100 nodes. Different numbers of clients have been simulated (see Figure 4). On average, each client sends one request every 4 ticks. The number of mobile clients remains constant throughout the simulation, and they are started at 250 ticks. The simulation is discrete, and time is measured in ticks. Every 5000 ticks, an adaptation algorithm is run that tries to select better service locations to reduce the overall network load.

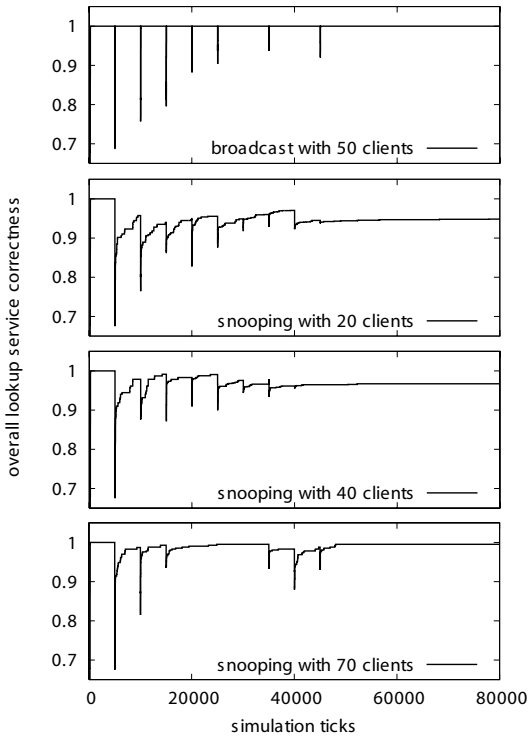

Figure 4. Lookup correctness measure for the overall lookup system with broadcast update propagation (top chart) and snooping update propagation with different numbers of clients (bottom three charts).

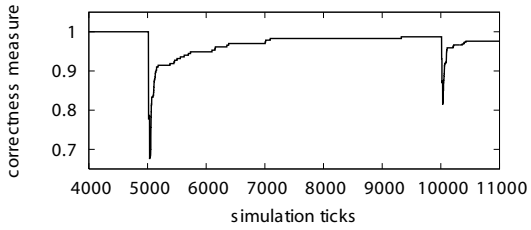

Figure 5. Single recovery phase of the snooping protocol.

The first result is depicted in Figure 4. It presents a comparison of a simple broadcast update protocol (first chart) with our lazy update protocol that is based on snooping (latter three charts). The lookup correctness measure (LC) introduced in this comparison is simply given as

$$LC = \frac{\sum_C}{\sum_A}$$

where $\sum_C$ is the sum of correct service entries in all Lookup Services and $\sum_A$ is the sum of all service entries (correct and incorrect) in all Lookup Services.

The reference broadcast protocol propagates location changes immediately to all Lookup Services by performing a cluster head broadcast of the new information. Therefore, after a small drop in LC whenever a migration takes place, the system recovers completely in a very short time. The other three charts in Figure 4 show the LC measure of three simulation runs with the same initial setup but different numbers of clients. Due to the random initial service replica placement (one per service type), the system performs suboptimal at the start in terms of network load. Therefore, many adaptations take place in the early stages on the simulation. After a few adaptation steps, the system manages to find a good service placement. Thus, the number of adaptations decreases due to a lack of better placements. Moreover, a single migration has less of an effect on the correctness as more replicas are added. Therefore, the drops in the LC measure (at multiples of 5000 ticks) become less significant over time.

For a small number of clients, the snooping protocol recovers relatively slowly from every drop in lookup correctness and only reaches up to about a 95% correct overall state. This is due to the fact that only the active areas in the network participate in the update process. In all four runs, no adaptations where allowed after 50000 ticks. The smaller the number of clients in the system is, the lower the final correctness level. This is due to the fact that more clients produce more requests per time unit, and they penetrate the system more completely. Thus, more Lookup Services get correct information.

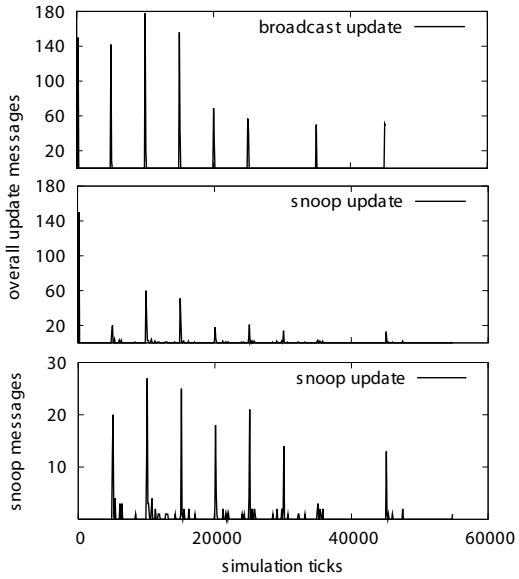

Figure 6. Lookup update messages.

A single recovery phase of the snooping update protocol is depicted in Figure 5. It shows the typical behavior of the protocol. After a sharp drop in correctness due to service mobility (at 5000 ticks), the protocol very quickly recovers to a correctness level of about 0.9. This is due to the initial dissemination of updates along the currently used reply paths. After that, the curve asymptotically approaches the theoretical maximum correctness of 1. Thus, the system is very quick in repairing those regions of the lookup system that are currently in use and as clients move to different regions, the correctness slowly increases further.

In Figure 6, the upper two charts present a comparison of the overall update protocol messages transmitted. This includes broadcast messages and snooping messages. Remember that our lazy snooping protocol also uses broadcasts for registration of new replicas and unregistrations. The significant reduction in update traffic becomes evident from the first two charts. The broadcast protocol produces 1050 messages over the 60000 ticks depicted in the chart, while the snooping protocol produces only 507 messages. This is a significant improvement of over 50%. The third chart shows only the snooping messages sent. Note how the sharp peaks in snoop messages corresponds with the high initial repair rate after each adaptation step (every 5000 ticks) in Figures 4 and 5. The volume of messages quickly decreases as less Lookup Services need to be updated.

7 Discussion

While the proposed lookup system in its current form achieves the desired goals in terms of adaptability, effectiveness, and efficiency, it also has some drawbacks and limitations. In this section we shall briefly discuss some of the advantages and disadvantages not presented thus far.

Services Replies The general concept of the ASG lookup system only works for services that generate replies. This does not present a general problem as any service can send a reply even if it was only for the sake of keeping the lookup system up-to-date. Such *non-functional replies* could be produced and consumed by the Proxy Agents transparently.

Snooping Overhead Message snooping and the general concept of implementing routing and transport mechanisms at the middleware layer may introduce additional overhead. We still need to quantify this overhead. However, resolving the strictly layered structures of classical distributed systems is necessary when it comes to achieving self-organization at the higher layers. Thus, this problem is of a more general nature and not solely a consequence of our approach.

Overhead through Forwarding Chains The lazy update scheme, trades off efficiency of updating with the increased latency of initial updates passing through long forwarding chains. However, if a system is heavily used, forwarding chains rarely get longer than one or two hops. Thus, the longer route has only a minimal effect. On the other hand, if the system is lightly loaded (few clients), then forwarding chains may get longer since updates happen more infrequently. However, in these situations the additional networking resources needed are also available since few requests have to be transported through the system. This presents a nice feature as the network adapts to the request load, minimizing forwarding chains automatically as the load increases.

Security Self-organizing software systems in general require new paradigms for achieving security, privacy, and integrity. Security is, in most cases, associated with explicit manual configuration and control. This is in stark contrast with the idea of letting a software system structure and control itself autonomously. The nature of adequate security paradigms is currently an open issue and a tough problem. Therefore, we consciously avoided this topic in the design of our system thus far. It seems obvious that some self-repair mechanisms like sending a *reset* message when a query for an unknown service type is received, open the gates for denial of service attacks. Such problems will have to be resolved before platforms like the ASG become commercially exploitable.

8 Conclusions

As the trend of rendering our every-day environment more intelligent progresses, Ambient Services that seamlessly offer new ways of interacting with the world around us become more important. To handle the dynamics exhibited by such environments, software platforms need to be adaptable and able to cope with frequent structural changes in the supporting infrastructures. In this paper, we have proposed a new request-driven Lookup Service that is capable of dealing with service mobility and replication. It is self-repairing and relies on a request-driven lazy update protocol to update only those regions of the network that actively participate in service interactions. Our experimental results show that the overall lookup system manages to recover from inconsistent states caused by service mobility. Moreover, it does so very efficiently by transporting meta information in regular service replies, and by employing message snooping to disseminate update information.

This lookup system is a vital ingredient for rendering service provisioning for mobile users more adaptable. Together with intelligent, dynamic service placement algorithms [2], it enables an ambient service infrastructure like the ASG to adapt to current client demand to achieve better resource usage and improved quality of service.

However, our work in the ASG area are only initial steps towards truly self-organizing software systems for seamless Ambient Services. As we discussed in Section 7, there are still some tough questions to be answered and some basic research to be done. The basic idea of rendering systems reactive instead of proactive, however, seems to be a promising general model for self-organizing systems. Moreover, our approach for building a lookup system, is in line with the call for soft and homeostatic systems [3]. In our view, this principle is fundamental for future self-organizing software platforms as it broadens the range of acceptable system states and enables gradual degradation and autonomous recovery instead of sudden failure.

References

[1] S. Giordano, *Handbook of Wireless Networks and Mobile Computing*, ch. Mobile Ad-Hoc Networks. Wiley, John & Sons, 2002.

[2] K. Herrmann, K. Geihs, and G. Mhl, "Ad hoc Service Grid - A Self-Organizing Infrastructure for Mobile Commerce," in *Proceedings of the IFIP TC8 Working Conference on Mobile Information Systems (MOBIS 2004)*, IFIP – International Federation for Information Processing, Springer, September 15-17 2004.

[3] M. Shaw, "Self-healing: Softening precision to avoid brittleness," in *Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS '02)*, pp. 111–113, November 2002.

[4] Sun Microsystems, "Jini(tm) architecture specification." http://www.jini.org/standards/, Feb 2004.

[5] UPnP Forum, "UPnP(TM) Devic Architecture 1.0," December 2003.

[6] The Salutation Consortium, "Salutation architecture specification – version 2.0c." http://www.salutation.org, June 1999.

[7] G. Kastidou, E. Pitoura, and G. Samaras, "A scalable mobile agent location mechanism," in *Proceedings of the 1st International Workshop on Mobile Distributed Computing (MDC'03)*, May 2003.

[8] G. Samaras, C. Spyrou, E. Pitoura, and M. Dikaiakos, "Tracker: A universal location management system for mobile agents," in *Proceedings fo the European Wireless 2002 Conference. Next Generation Wireless Networks: Technologies, Protocols, Services and Applications*, February 2002.

[9] T.-Y. Li and K.-Y. Lam, "An optimal location update and searching algorithm for tracking mobile agent," in *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pp. 639–646, ACM Press, 2002.

[10] Y. Aridor and M. Oshima, "Infrastructure for mobile agents: Requirements and design," in *Proceedings of the 2nd International Workshop on Mobile Agents* (K. Rothermel and F. Hohl, eds.), vol. 1477 of *Lecture Notes in Computer Science*, pp. 38–49, Springer, 1998.

[11] S. Basagni, "Distributed Clustering for Ad Hoc Networks," in *Proceedings of the IEEE International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN)*, pp. 310–315, June 1999.

[12] K. Herrmann, "MESHMdl - A Middleware for Self-Organization in Ad hoc Networks," in *Proceedings of the 1st International Workshop on Mobile Distributed Computing (MDC'03)*, May 2003.

[13] D. Gelernter, "Generative Communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, 1985.

[14] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pp. 172–182, ACM Press, 1995.