Report Number: WUCSE-2004-34

2004-06-18

Context Aware Session Management for Services in Ad Hoc Networks

Radu Handorean, Rohan Sen, Gregory Hackmann, and Gruia-Catalin Roman

The increasing ubiquity of wireless mobile devices is promoting unprecedented levels of electronic collaboration among devices interoperating to achieve a common goal. Issues related to host interoperability are addressed partially by the service-oriented computing paradigm. However, certain technical concerns relating to reliable interactions among hosts in ad hoc networks have not yet received much attention. We introduce ”follow-me sessions”, where interaction occur between a client and a service, rather than a specific provider or server. We allow the client to switch service providers if needed. The redundancy offers scope for reliable communication in the presence of mobility induced disconnections.... **Read complete abstract on page 2.**

Context Aware Session Management for Services in Ad Hoc Networks

Radu Handorean, Rohan Sen, Gregory Hackmann, and Gruia-Catalin Roman

Complete Abstract:

The increasing ubiquity of wireless mobile devices is promoting unprecedented levels of electronic collaboration among devices interoperating to achieve a common goal. Issues related to host interoperability are addressed partially by the service-oriented computing paradigm. However, certain technical concerns relating to reliable interactions among hosts in ad hoc networks have not yet received much attention. We introduce ”follow-me sessions”, where interaction occur between a client and a service, rather than a specific provider or server. We allow the client to switch service providers if needed. The redundancy offers scope for reliable communication in the presence of mobility induced disconnections. We exploit strategies involving the use of contextual information, strong process migration, context-sensitive binding, and location-agnostic communication protocols. We show how follow-me sessions mitigate issues related to proxy-based service-oriented architectures in ad hoc networks, making them more reliable.

2004-34

Context Aware Session Management for Services in Ad Hoc Networks

Authors: Radu Handorean, Rohan Sen, Gregory Hackmann, Gruia-Catalin Roman

June 18, 2004

Abstract: The increasing ubiquity of wireless mobile devices is promoting unprecedented levels of electronic collaboration among devices interoperating to achieve a common goal. Issues related to host interoperability are addressed partially by the service-oriented computing paradigm. However, certain technical concerns relating to reliable interactions among hosts in ad hoc networks have not yet received much attention. We introduce "follow-me sessions", where interaction occur between a client and a \textit{service}, rather than a specific provider or server. We allow the client to switch service providers if needed. The redundancy offers scope for reliable communication in the presence of mobility induced disconnections. We exploit strategies involving the use of contextual information, strong process migration, context-sensitive binding, and location-agnostic communication protocols. We show how follow-me sessions mitigate issues related to proxy-based service-oriented architectures in ad

Context Aware Session Management for Services in Ad Hoc Networks

Radu Handorean, Rohan Sen, Gregory Hackmann, Gruia-Catalin Roman
Department of Computer Science and Engineering
Washington University in St. Louis
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130-4899, USA
{radu.handorean, rohan.sen, ghackmann, roman}@wustl.edu

ABSTRACT
The increasing ubiquity of wireless mobile devices is promoting unprecedented levels of electronic collaboration among devices interoperating to achieve a common goal. Issues related to host interoperability are addressed partially by the service-oriented computing paradigm. However, certain technical concerns relating to reliable interactions among hosts in ad hoc networks have not yet received much attention. We introduce "follow-me sessions", where interaction occur between a client and a *service*, rather than a specific provider or server. We allow the client to switch service providers if needed. The redundancy offers scope for reliable communication in the presence of mobility induced disconnections. We exploit strategies involving the use of contextual information, strong process migration, context-sensitive binding, and location-agnostic communication protocols. We show how follow-me sessions mitigate issues related to proxy-based service-oriented architectures in ad hoc networks, making them more reliable.

1. INTRODUCTION
The increasing ubiquity of computers and portable computing devices has made pervasive computing feasible. Many hosts in such environments are small mobile devices with limited storage space and low computational power but with wireless communication capabilities. These portable devices may not carry all the programs their users may need (e.g., they do not have enough storage space for all the programs the user needs) or may not have enough computational power to run some of the code they carry (e.g., running a public/private key encryption algorithm takes many resources, such as memory space and CPU cycles, but the code footprint is not large). One way to expand their capabilities is to search other hosts for the functionality they need or to find hosts willing to accept code to be executed on their behalf (i.e., the code will be pushed onto these helper hosts), in case they have it but cannot afford to run their own code. Independent of how the interaction between the client and the server begins (the functionality is discovered or pushed), the code another host (service provider) runs for the client's benefit is called a *service*. The client controls the service remotely, while it runs on a different host.

As mobile devices move in space, they are able to communicate for limited periods of time when within wireless communication range of each other. An interaction begun between two hosts may need more time to complete than the interval of connectivity between the two hosts. The client could, therefore, partially complete the task with the help of some host, pause its work, and resume it on another host when the first host becomes unreachable. We can thus stretch the processing of a task over multiple hosts as they fall within the client host's communication range and can contribute pieces of computation toward finishing of the entire task.

For reliable service provision in ad hoc networks, it is imperative that the interaction between the client and the service provider completes. In ad hoc networks, the movement of hosts can bring them out of wireless communication range, so we cannot guarantee uninterrupted connectivity between a client and a server application. Most of the time, host movement is independent of the application behavior. For this reason, we add a layer of abstraction to mask this unwanted added complexity in the client-service provider interaction by introducing the notion of a *"follow-me"* session.

Definition: A *"follow-me"* session is the aura of interaction between a client application and a service that provides external functionality needed by the application, masking the disconnections between intervals of connectivity.

Intuitively, a session is the interaction between the client application and the service until the client finishes the task at hand. We emphasize that the interaction is between the client and the *service*, not a specific service provider or process. We allow the client to switch service providers if needed, given that the new provider offers the same functionality (i.e., the provider changes but not the service itself). This can be achieved in two ways. First, the client can automatically connect and exploit the same functionality from a new provider as the current provider moves out of communication range and a new one comes within communication range (assuming the new provider offers the same functionality). Second, the process offering a service migrates from host to host, following the client's host as it moves in space. The follow-me session offers, within limits, the *continuity* of service provision; all the challenges associated with having the service follow the client host will be handled in a manner transparent to the client application, using strong process migration, a new concept called context-sensitive binding, and location agnostic communication protocols.

The remainder of the paper is organized as follows: Section 2 presents background material for this paper. Section 3 describes the server migration mechanism. In Section 4 we discuss the context sensitive binding mechanism. The implementation of the system is detailed in Section 5. Section 6discusses the contributions of our work, together with future work remarks. We conclude the paper in Section 7.

2. BACKGROUND AND CONTEXT AWARE SESSION MANAGEMENT

One of the key features of service oriented computing (SOC)is the decoupling between the interface and the implementation of some functionality, which can be advertised by service providers and discovered by service users at run-time. There are two main approaches to SOC: proxy-based and web services.

2.1 Service oriented computing architectures

Proxy-based SOC entails a server process running on some host (e.g., service provider's host) and a proxy object that acts as a remote handle for that server. The proxy object is retrieved by the clients and used as the local representative of the service implementation. The client interacts with this proxy object as it would with any other locally available object. In some cases, this proxy object can deliver on its own the entire functionality requested by the client. Most of the times, the proxy object connects to the server running on the provider's host and tunnels the client's method calls to be executed by this server, while giving the client the impression that its calls are handled locally. The interaction between the proxy and the server is not of interest to the client application but of interest to us because of the implications of mobility on the interaction between the two, which we handle at middleware level.

In web services, lower level languages such as the Web Services Description Language (WSDL) [1] describe how the data is sent across the wires. This layer handles all application layer protocol issues. Higher level languages describe what is being sent across and why. High level description languages are required to be clear and concise, and to support easy matching between two entities. Ontologies are high level languages that capture the semantics of an entity and its relation to other entities. An ontology is often itself structured into layers. DAML+OIL [2] is a combination of a markup language to construct ontologies (DAML) and an ontology inference layer (OIL) to interpret the semantics of the description. It is currently the language of choice for formulating ontologies for web services. The lowest layer of the system provides the syntax and is encoded in XML [3], since it is a W3C standard and hence has maximum acceptance in the market. Above the syntax layer a framework provides a basic set of constructs to describe entities and relations. In DAML+OIL, the Resource Description Framework (RDF) [4], also a W3C standard, fulfils this functionality. However, RDF is not powerful enough to describe entire ontologies. DAML+OIL fills this gap by extending the RDF concept to provide more constructs and relations.

The major difference between the web services and proxy-based models is that in web services $\mathsf{service} = \mathsf{server}$ while the rest is assumed to be known by the client while in proxy-based systems, $\mathsf{service} = \mathsf{server} + \mathsf{proxy}$, where the client only needs to be able to interact with the proxy.

Both approaches (proxy-based and web services) to SOC have been developed under the assumption of a wired infrastructure. They are characterized by centralized architectures, easy to deploy in wired networks. The transition to ad hoc networks entails more than just porting the software infrastructure onto mobile hosts. In [5] we have motivated the use of proxy-based architectures over web services in a mobile ad hoc networking environment. In our research, we adopted the proxy-based model because we found it to be more flexible and easier to adapt to the challenges raised by the ad hoc environment. Briefly, proxy-based SOC offers the advantage of allowing for thin client applications. When the application needs external help, it uses the proxy of the needed service and then discards it. This allows for a modular design, built around a thin core client application.

A well-known implementation of this model is Jini [6] [7]. That implementation, however, was targeted towards a wired, fairly reliable, networking environment, made evident by the centralized design and implementation of service repositories where the providers advertise their offerings and where clients go to search for needed functionality. Jini employs a leasing mechanism, which is a garbage collection mechanism that cleans up orphan advertisements which remain in the service repository after the service provider is no longer available (e.g., disconnected because of a crash or because its host went out of communication range, in the case of a mobile host). While a leasing mechanism prevents overcrowding the service repository with obsolete advertisements by periodically wiping out the orphan advertisements, it does not solve the consistency problem, i.e., clients can still discover advertisements before they get collected. This scenario is depicted in Figure 1.

In ad hoc networks the infrastructure for communication is made entirely of mobile hosts that interact in a peer-to-peer manner. There is no fixed infrastructure on which the mobile hosts can rely for message relay. Centralized architectures fail because the availability of a service repository running on a certain host cannot be guaranteed when hosts move (they depend too much on a single host that can go away). Consider the scenario in Figure 2. The client could use the service offered by the server running on the nearby host but it cannot discover the service because the advertisement is stored on a third host, where the lookup service is running (assuming the server was in contact with the lookup service and had a chance to advertise itself; otherwise, the service provider is non-existent to the outside world since it did not have a chance to publish its advertisement).

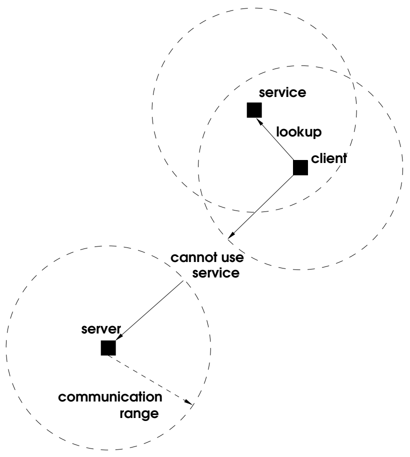

Figure 1: The service discovered is unusable because it is not reachable.

As long as two devices are connected (direct contact or multi hop), they can exchange information and interact as if they were wired. In ad hoc networks, a stable multi-hop connection has severe temporal limitations due to the routing algorithms that have to be carried out by hosts themselves. In [8] we addressed issues related to service discovery in ad hoc networks and security of interactions in this open environment. We continue our work by taking the client-service provider interaction to a new level, in a disconnected working environment. We provide mechanisms which improve the client-service interaction despite the challenges raised by the mobility of hosts and the inherent disconnections.

While using a proxy isolates the client from the network communication issues and remote invocation challenges, thus allowing for thin clients, disconnections continue to be a challenge for the design of the proxy-server interaction. In ad hoc networks this proves to be a difficult task, a common challenge for all service providers in this changing environment. Having identified disconnections as a common challenge, we provide a layer of abstraction which shifts part of the challenge from the (service) application developer down into a middleware layer.

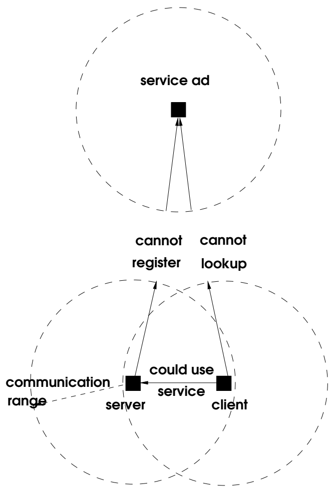

Figure 2: The service registry is unreachable.

2.2 "Follow-me" session

We introduce the concept of a *"follow-me" session* for SOC in ad hoc networks. If we compare the definition from Section 1 against the definition of a working session as it is currently used: "in computing, in particular networking, a session is either a lasting connection using the session layer of a network protocol or a lasting connection between a user (or user agent) and a peer, typically a server, usually involving the exchange of many packets between the user's computer and the server"[1], we observe a few fundamental differences. In our case, the session spans beyond a single interval of direct connectivity. Since the hosts can repeatedly move in and out communication range, the working session can span multiple such connectivity intervals. Throughout an entire working session, the client (in actuality the service proxy used by the client) interacts with its server sporadically (e.g., when invoking a method and returning the results) and therefore the lower layers in the stack of protocols need not enforce a live connection if it is not being used (i.e., if a disconnection occurs while nobody is talking, the applications should not crash or even notice). We have thus eliminated the "lasting connection" from the classic definition, allowing a session to be interrupted by disconnections. In our case, the session can involve multiple servers, as the same proxy could connect to another server that comes within communication range after a previous interaction with another server delivering the same functionality was interrupted because of host mobility. A session can span multiple hosts as well, since the server implementing the service may be mobile code relocating to follow the client as it moves in physical space.

We can identify several important elements that will help us manage a "follow-me" session in the ad hoc networking environment. One important issue is for the middleware to be able to transparently reconnect to a new service provider which offers the same functionality as the one currently being used, if the new provider will be available for a longer period of time. We mention that there can be multiple reasons why the proxy needs to reconnect to a new server but in this paper we only consider the duration of the interaction (direct connectivity) between the two hosts as a measure for what "better" service means. Another important issue is to be able to move the implementation of the server to which a proxy connects from one host to another, to follow the client's host as it moves in space. We also need a communication protocol that supports resuming in case of temporary disconnection, and supports transparent relocation of the source/destination processes from one host to another. We call this a *location agnostic communication protocol.* Since the transfer of pause/transfer/resume cycle of an ongoing computation as well as source data deployment and (partial) result retrieval take a toll on the computation and transfer time, we need to evaluate the time it takes for these actions to be performed and factor them along with

[1]As defined by the Free On Line Dictionary of Computing at http://wombat.doc.ic.ac.uk/foldoc/.

connectivity data into the session management algorithms. We include all these into a middleware layer to provide the "follow-me" session in a manner transparent to the client.

A complete overview of the challenges is presented in the simplified but comprehensive scenario depicted in Figure 3. The circle represents the client application's host. The horizontal dashed line represents the client hosts's trajectory. As the client approaches host $H1$, it pushes the implementation of a service it needs onto $H1$, while holding on to the proxy object which will be used to manipulate this server remotely. The dashed arrows indicate how the server moves (a variation of the scenario might allow the client to simply discover the server which is already running on $H1$). The solid arrows denote the client's interactions with the server dedicated to carrying out the task and are not related to session management. When the client is in position c, the server migrates from $H1$ to $H2$ since the client will soon lose connectivity to $H1$ but will remain connected to $H2$. The transfer from $H1$ directly to $H2$ is possible because $H1$ and $H2$ are within communication range of each other. When the client reaches location e, there is no host where the server could jump and therefore the client will "take back" the server (the dashed arrow from $H2$ to the client). The client cannot run the server on its own host because the resources available on that host do not allow it and therefore the client will only transport the server until a new host is found. At location f the client pushes the server onto $H3$ where the client will manipulate it remotely until the client reaches location h. The disconnection from $H3$ is imminent but $H4$ advertises the same service. The client will continue its job using the service advertised by $H4$, since it is much cheaper to migrate the computation state than the entire server process, and have the new server "resume" from a predefined intermediary progress point. While the client interacts with $H4$ the task is completed.

2.3 Mechanisms for delivery

The session management middleware we are describing in this paper provides transparent support for the interaction between the proxy manipulated by the client application and the server process the proxy connects. To deliver "follow-me" sessions we develop a server thread migration mechanism (for situations when the service is not offered on a certain host) and context-sensitive binding (for situations when the service is offered by multiple providers and the client can choose which one to use). To smooth the transition from one provider to another (via migration or rebinding) we provide a supporting layer that implements a location-agnostic communication protocol.

3. SERVER THREAD MIGRATION

An essential contributor to our "follow-me" session is the process migration. Process migration is composed of code migration, object state migration and execution state migration. The code migration downloads the binary code of the process on the target machine, where the loader can find it and load it into memory. Object state migration, also known as object serialization, transfers the state of an object's instance variables of the object when the code is already available on the destination machine. Execution state migration entails transferring the program counter and the call stack content.

3.1 Types of migration

Depending on whether the migration addresses the execution state migration or not, process migration can be split in two types: weak migration and strong migration.

Weak code migration requires that the process can run, but not *resume*, on the destination host. This involves making the code available on the destination machine, loading it and *restarting* the process from the beginning, losing any progress the process may have made before migration. The execution state is not transferred during weak migration. Examples of weak migration are [9, 10]. In some cases, some initialization data can be transferred along with the process but that does not account for execution state transfer. The process is still started from the very beginning, except that the memory is initialized to contain potential partial results. An example of such behavior can be observed in μCode [11].

Strong code migration entails the migration of the execution state as well. This allows for processes to be stopped, transferred and resumed at a new destination. To deliver the desired semantics, our design entails a strong migration mechanism for Java threads (as described in Section 5, our system is implemented in the Java programming language). While capturing and transferring the execution state, the program counter of the Java virtual machine (which indicates where the execution should be resumed from) and the Java call stack (which is equivalent to conventional languages call stack) are captured and transferred in a serializable format to the new destination. Obviously, strong mobility is more powerful but it is also more expensive to deliver. Systems that support strong migration are [12, 13].

Ideally, the migration happens completely transparently to the process being transferred. This is, however, extremely dangerous. For example, such a process could be transferred at a moment when it holds locks on resources. Without support from the operating system, these locks would never be released, since the owner process does not operate under the supervision of the operating system on the current host. A system that achieves transparent process migration, in cooperation with the operating system is [14]. In our case, the JVM on each host plays the role of the operating system and it is one of our goals not to tamper with the JVM. Hence, we cannot migrate a server in a manner completely transparent to the process itself, and therefore to the human developer who writes the code of the server. We give the programmer control over the places where the process is paused and transferred by manually marking such locations with checkpoints. This does not guarantee that the developer does not use the checkpoints in wrong places, e.g., while having a resource locked for exclusive access.

3.2 Migration

We deliver our strong migration in two forms: *lightweight* and *heavyweight* migration. The lightweight migration entails only the transfer of state information. In Figure 3, when the client is in position e or h, between $H2$ and the client (in position e) or the $H3$ and $H4$ (when the client is in position h) the migration is lightweight. The client cannot run the executable code, so it only needs the state information from the server process on $H2$. Similarly, host $H4$ only needs the state information from $H3$ since it already has the

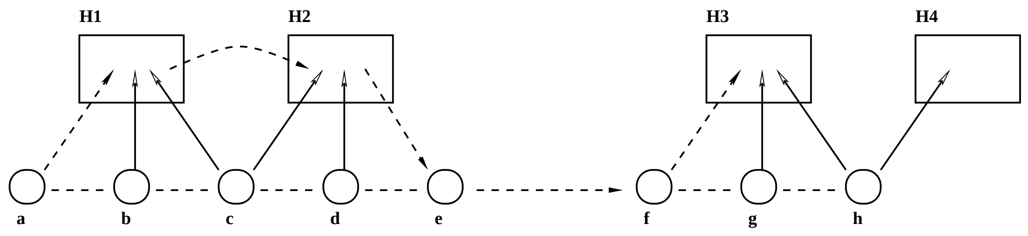

Figure 3: Follow-me session.

code (it provides the same kind of service) and therefore a lightweight migration would suffice. In contrast, when the client encounters $H1$ (which does not offer the service but offers to run the client's code) at position a, a heavyweight migration is needed to upload all the bytecode onto $H1$, as well as in position c and the server migrates from $H1$ to $H2$, which does not offer the service needed by the client either.

During migration, the serialization process wraps only the content of an object (values of member variables) and not the bytecode from which the object was created. This includes all objects inside the initial object. Therefore, we need a separate mechanism to transfer the bytecode for each object along with its dependencies to the destination host. We use a combination of reflection and exceptions to build the closure of the server classes that need to be migrated. We developed a custom class loader [15] that is able to capture on the destination host all exceptions caused by missing bytecode. After catching such an exception, we use reflection to build the list of dependencies (instances of other classes the object that triggered the exception creates) and send it to the the source host, demanding the missing bytecode. On the source host, we inspect the proxy we advertise using reflection when we publish the service advertisement. We publish the code of all classes the proxy instantiates and which are not part of the standard JDK or of the middleware (i.e., they may not be available on a client machine). This code is downloaded, installed and loaded by our custom class loader on the destination host. We offer more details about this procedure in Section 5 of this paper.

A process should not migrate when holding reserved resources. Therefore, the developer should only place checkpoints in places where the process does not hold any locks on shared resources. Since locks on shared resources are usually not held for a long time (to avoid hoarding), we consider this to be an acceptable constraint on the developer.

Regardless of whether the migration is lightweight or heavyweight, the server continues to execute on the source host. The migration entails a *copy* of the process onto the destination host (heavyweight migration) or a transfer of the state information (lightweight migration) and does not *move* the process from the original host. The "old" server continues to run until it reaches the next checkpoint. As we will see more in detail in the implementation section, a Java thread cannot reliably be stopped from the outside. (Java has a mechanism to do so but it is deprecated for stability reasons.) Instead, the middleware sets a flag which the thread verifies every time it enters a checkpoint. Thus, the termination initiative appears to belong to the server itself.

3.3 Checkpointing and state saving

Checkpointing is a mechanism introduced to improve the fault tolerance of software. It entails saving the current state of a program and its data, including intermediate results to non-volatile storage, so that if interrupted the program can be restarted at the last checkpoint. If a program run fails because of some event beyond the program's control (e.g. hardware or operating system failure) then the processor time invested before the checkpoint will not have been wasted. We address failures triggered by hosts' disconnection while a task is processed in a distributed manner.

If a proxy-server interaction does not run to completion during a window of connectivity, the task has to be completed during a subsequent window of connectivity between the client's host and another host running an instance of the same server. In this section we consider the case when the server migrates from the now-disconnected host with a longer window of connectivity. Using checkpoints, the interaction can resume from an intermediary point (i.e., from the last checkpoint the execution flow went through), and does not have to be restarted from the beginning.

At each checkpoint we record the state of the thread, including its program counter and call stack, which are known as the *execution state*. The standard JVM does not make available any information about the program counter. We had to introduce an artificial program counter, which is updated at each checkpoint. The value of the program counter is transferred to the destination host and is used to resume the execution of the server process. As in any conventional programming language, every time a method is invoked, a new frame (or entry) is created and pushed to the top of the call stack. The frame, which contains the parameters of the method call and local (temporary) variables, is destroyed when the method returns.

The *data state* is composed of the values of instance variables (live objects). These are easier to transfer as serializable objects (we do request that these objects are all serializable).

When the server migrates from one host to another, the state recorded at the last checkpoint visited is transferred. Any further computing is lost as the session and state information resumes from that last checkpoint, e.g., if the checkpoint is placed just before a `for` loop, the loop will be started from the beginning when the execution is resumed at the destination host. If the checkpoint is added immediately inside the loop, the execution resumes with the last iteration of the loop executed on the initial host of the server process.

4. CONTEXT SENSITIVE BINDING

In the previous section we showed how we can deliver the functionality of a follow-me session by migrating a process from one host to another. In this section, we present a different strategy for achieving this functionality through context-sensitive binding, a novel mechanism that decouples the interface of a service from its realization. The realization of the interface is provided by a changing set of servers such that the best server provides the realization at any given time, where best is defined by user-specified criteria. For example, best may be defined as "the server with the highest available bandwidth" or "the server with highest remaining battery power" or "the server that is physically closest to the client". The switching between servers occurs dynamically and transparently in a context-sensitive manner. Using Figure 3 as an example, if we consider time to disconnection as the criteria for switching servers, once $H4$ comes within the client's host communication range the context-sensitive binding mechanism will disconnect the proxy from the server on $H2$ and connect to the server running on $H4$.

4.1 Novel features of context-sensitive binding

Policy based selection - The set of qualifying providers is chosen based on programmer specified policies. An example of a policy is "I want to be connected to the server which is closest to me, with the remaining battery power of the server being the tiebreaker". When choosing a service provider, the client specified policy is the first filter that is applied after choosing a set of service providers that already offer the needed functionality (i.e., this step executes after the service query/discovery step) or that offer to run code provided by the client. Other parameters for the policy may include security constraints (e.g., the client cannot print secret documents on any of the available printers), geographical restrictions imposed on the provider's location (e.g., the client will not print on a printer on a different floor), etc.

Metric based evaluation - Once the set of candidates is determined, the best provider is chosen according to client-specified metrics (e.g., the client prints on the fastest printer that passed the previous filters).

Transparent binding maintenance - Context-sensitive binding provides for dynamic switching between servers to provide the best available service at any given time. Except for a small interval of time when the mechanism is switching between servers, continuous binding between the client and the server is maintained. However, the switching of the server is masked from the client by the middleware, so from the client's perspective, the binding appears continuous for the time interval it uses the service.

4.2 Mobility and context-sensitive binding

In the previous section, we mentioned that the behavior of the context-sensitive binding mechanism was driven by client-specified policies and metrics. While one can specify any kind of policy, in the interest of a targeted and complete treatment, we focus on spatiotemporal policies, as these are among the most relevant in ad hoc networks. More specifically, we deal with the policy that emphasizes the connectivity of hosts over all else. The metric for evaluating hosts is the duration of connectivity between the client host and the server host(a longer interval is better).

Context-sensitive binding (CSB) is responsible for changing the server such that it is the best available given a context. In ad hoc networks, the primary reason for context changes is physical mobility. As hosts move in space, they encounter a changing set of hosts over a period of time. Since other hosts too move in space, the set of hosts that the client host is connected to changes rapidly and unpredictably. Given these changes, the context-sensitive binding mechanism must constantly evaluate which of the servers is most likely to be connected the longest and switch the server if necessary. In addition, it must have the capability for some rudimentary decision making, e.g., it must decide whether the cost of switching the server is worth it or not.

Context Gathering: To be able to make decisions about which server to choose, CSB must gather contextual information. In our implementation, which is based on the LIME middleware, all the required contextual information (connected host list, location of hosts, etc.) is provided as a resource by the middleware. We can simply obtain this information via programmatic calls to the middleware, eliminating the need for additional communication or processing.

Server Choice: Once the context information has been gathered, the next step is to choose the best option. This is done by ranking each choice according to the metric specified. Currently we support only simple metrics which specify a parameter, and to minimize or maximize it. Support for more complex metrics is currently in the pipeline.

Switching Servers: Once a server is chosen, the client begins its interactions with the server. Periodically, a snapshot of the interaction is taken (as described in Section 3) and propagated back to the client. The client stores this snapshot locally. In parallel, the context gathering process periodically evaluates the context to check if the current server is still the best server to use. The periodicity of this check can be customized according to the environment. For example, for fast moving cars, a periodicity of 1 second would be appropriate, whereas for ships, a periodicity of 10 seconds would not be detrimental and save computational resources. By default, we set the periodicity to 4 seconds.

If it is determined that the server needs to be switched, the middleware preempts the interaction between the client and the server and sends a termination message. The client's calls to the server are then temporarily held locally while the latest snapshot is propagated from the client to the server. Once this snapshot has been installed, again using the mechanism in Section 3, the calls are directed to the new server. This is achieved via the use of location agnostic protocols, which are described later in this section. It should be noted that the rebinding to a new host can be influenced by other parameters than those related to the physical movement of hosts. For example, if a certain host does not renew its security credentials, it may become unqualified to provide the same service it has been providing, even if it still has the server code. The clients will "avoid" that server because the context-sensitive binding mechanism will break the connection with that host, even if another option is not available. Another example could be a printer running low on paper. If the client has multiple documents to print, at some point, the context-sensitive binding mechanism will connect to another printer that has more paper in the tray.

The key difference between CSB and strong migration is that only the state is transferred without any thread state migration. The usefulness of this is when proprietary software is involved, or there are licensing issues. Rather than move the process, we simply switch to another instance of it, which allows the client to continue its work without raising issues of security and ownership of code.

4.3 Location agnostic protocols

A key element in the delivery of the mobile session management is the communication protocol between the client(proxy) and the server applications. All these interactions logically belong to the same "follow-me" session. Therefore, the client side should not be impacted by the disconnection from server, server migration or context-sensitive rebinding to a new server as long as the session is in progress. While the disconnection cannot be completely hidden from the client application (when the client expects an immediate answer from a disconnected server, there is nothing the middleware can do to help), it can be masked as a delayed response until the two hosts reconnect or the answer is obtained from another party. When the server migrates or the proxy is rebound to another server, the change has to be completely transparent to the client application.

To deliver this, we employ location-agnostic communication protocols. The client obtains a unique session identifier which will be used to stamp all messages exchanged in a working session. The client only knows that at the other end there is a server handling the requests belonging to this session. Similarly, the server knows to pick up and serve only messages marked with the appropriate session ID. This leads to a communication protocol based on the content of the messages rather than the explicit destination stamped on each message. This idea resonates with modern implementations of the Linda [16] coordination model.

5. IMPLEMENTATION

The framework described in the previous section has been implemented in Java, using LIME [17] as a middleware to handle the implications of an ad hoc wireless network, i.e., physical mobility of hosts. In this section we present a brief overview of LIME, followed by a description of the implementation. We also show a proof of concept via a set of demo proxies running on our client.

5.1 LIME overview

LIME is a Java implementation of the Linda [16] coordination model, designed for ad hoc networks, which masks details associated with coordination and communication from the application programmer. A host running LIME runs a `LimeServer` upon which run one or more LIME agents, which are analogous to applications. Coordination in LIME occurs via transiently shared *tuple spaces*. Every tuple space in LIME is identified by a name. Tuple spaces having the same name can be merged to form a federated tuple space when their hosts are within communication range.

Tuple spaces are containers for tuples. Tuples are ordered sequences of Java objects which have a type and a value. An agent places a tuple in the tuple space, making it available to all other agents that are sharing the same tuple space. To read a tuple from the tuple space, an agent needs to provide a template, which is description of the tuple that the agent is interested in. A template is a sequence of fields, each of which can contain a formal representing the required type for that field or an actual value that identifies the type and value of the corresponding field. A template is said to match a tuple if all the corresponding fields match pairwise.

An agent can access the tuple space via standard Linda operations (`rd` (read a tuple), `in` (remove a tuple), `out`(write a tuple)). The `in` and `rd` operations take a template as a parameter and return a tuple as the result or block until a match is found (the operations are synchronous). LIME offers probe variants of the traditional blocking operations (e.g., `inp`, `rdp`), and group operations (e.g., `outg`, `ing`, `rdg`, `rdgp`, and `ingp`). While the original calls return a matching tuple (if available) or null otherwise (if nonblocking), the group operations return all matching tuples.

To provide asynchronous interactions, LIME extends the basic Linda tuple space operations with a reaction mechanism $\mathcal{R}(s, p)$, defined by a code fragment s that specifies the actions to be executed when a tuple matching the pattern p is found in the tuple space. Blocking operations are not allowed in s, to ensure programs reach fixed point.

LIME protects applications from the complexity associated with sudden disconnection by using location information. The concept of safe distance [18] helps preserve the consistency of the system by predicting disconnections. When a host approaches a group, it *engages* with the group only after it comes within safe distance of some member of the group. Once the safe distance is exceeded, an automatic *disengagement* protocol is triggered and the group is split, ensuring that no messages between group members are lost.

5.2 Checkpoints

The implementation of strong code migration presents several interesting technical problems. First, the standard JVM does not allow programs to save or restore the program counter. Second, once an application has been migrated, it should stop running on the original host; but arbitrarily stopping threads in the middle of execution is inherently unsafe. Finally, saving the complete state of an application involves saving its local variables, which cannot be accessed at runtime by an external library.

We approached these problems by choosing to rewrite the bytecode of applications rather than trying to manipulate them at runtime. This rewriting process adds bytecode to applications to add support for strong code migration, including code to work around these technical limitations.

As noted earlier, the state of the server application is saved at specific checkpoints. The application programmer creates mobile applications by extending the `MobileThread` class, which adds several methods and fields to Java's standard `Thread` class as described below. The programmer defines checkpoints by calling the `addCheckpoint()` method. Though this appears to the programmer to be an ordinary method call, it simply serves as a placeholder in the bytecode to indicate when the partial progress and state information are recorded. After compiling the Java source code, the resulting bytecode is passed into the bytecode rewriter. The rewriter loops through all the methods in the class and modifies them to allow strong code migration.

To do this, the rewriter first collects a list of all the local variables in the current method. It then adds a field for each of these local variables; these fields will be used later to store the state of the local variables. The rewriter also inserts a field to store an artificial program counter. The rewriter then searches for all calls to `addCheckpoint()`. At each checkpoint, the rewriter inserts code to check the `do_pause` field, which indicates whether or not the application thread is being paused so it can be migrated. If this field is set, then the method immediately returns. If it is not, then the method copies all of the in-scope local variables to the fields described above and then sets the artificial program counter to some unique value. Finally, the rewriter removes the call to `addCheckpoint()`, since it only serves to mark the bytecode. The rewriter also appends code at the end of the method to copy these fields back into the corresponding local variables and jump to the checkpoint; these "restoration points" provide a place for the thread to restore its state and return to the last checkpoint it passed before being migrated. The bytecode rewriter then adds code to the beginning of the method to see if the `paused` field is set. If it is, then the application jumps to the appropriate restoration point based on the contents of the artificial program counter field. This has the indirect effect of restoring the thread's local variables and the JVM program counter.

The `MobileThread` class adds two important methods to the standard `Thread` class: `pause()` and `unpause()`. The `pause()` method sets the `do_pause` and `paused` fields to `true`; the former tells the thread that it should stop execution as soon as it reaches the next checkpoint, and the latter tells the thread that it should restore its state when it is restarted. The `unpause()` method simply resets the `do_pause` field to `false` and restarts the thread; since the `pause()` call set the `paused` flag, the thread will jump to the appropriate restoration point and return to the last checkpoint passed before pausing. This way, rewritten applications can be migrated across hosts by pausing the application thread, serializing it on the original host, deserializing it on the new host, and unpausing it.

5.3 Migration

When migrating applications across hosts, it is very likely that one or more will not have the bytecode for the migrated application or one of its dependencies. The solution to this problem is a custom class loader that can locate missing bytecode from shared tuple spaces.

When a host on the network creates an instance of an application, it passes the application's class file to our middleware, which searches the class's bytecode to find all the other classes it refers to. It then recursively analyzes these new dependencies until the set of classes converges; this is the application's full class closure. (Any classes provided by the standard JRE or any parts of our middleware are excluded from this set, since it is assumed that they are always available on any host.) The middleware then packages the bytecode for all these classes together in a JAR file and creates a new tuple of the form <Names:class names, BinaryCodeFile:bytecode>. It `outs` this tuple into a shared `CodeRepository` tuple space.

When an application is migrated to a remote host, our middleware attempts to deserialize it. If this fails due to missing bytecode, the deserialization mechanism throws a `ClassNotFoundException`, which out middleware intercepts and uses to fetch the needed bytecode. This is done by using a custom classloader and a custom `ObjectInputStream` that refers to this new class loader. Our custom `ObjectInputStream` intercepts any failed attempts to resolve classes locally and invokes our custom `LWClassLoader`, which attempts a `rdp` operation on the code repository using the pattern <Names:class name, BinaryCodeFile.class> to retrieve the byte code for the required class from the code repository. If this `rdp` operation succeeds, the class loader loads the JAR package contained in this tuple into memory and presents each the bytecode of each class inside to the class loader.

5.4 Protocols and context sensitive binding

The choice of tuple space-based communication was a natural fit for communication protocols that are immune to disconnections, support resuming, and do not use explicit location information for message delivery. The tuple space communication is similar to exchanging messages on a board: the source puts the message out and the recipients come and look for the messages they need. The level of granularity we assume allows to transfer a tuple atomically from one host to the other. If a disconnection occurs during transfer, the tuple will remain available in the source agent's local tuple space. The protocol supports resuming only at the layers above tuple space coordination, and therefore if the transfer of a tuple is interrupted it will have to be restarted from zero. The tuple-based communication (which is analogous to message passing), handled at middleware level, protects the application from crashes in face of disconnections.

The location-agnostic character of interactions is also a benefit of using tuple spaces. The recipient of a message listens for messages using a template that describes the messages it should read (using the reaction mechanism described in the LIME overview section). In our implementation, each proxy-server pair stamp their messages with a shared session ID. There may be more servers providing the same service, but each server will only pick up messages labeled with the session ID it is currently serving. This session ID is created by the proxy when the proxy is instantiated on the client's machine by obtaining a hash based on the object's memory address, which is unique on the host. We combine this value with a host ID and stamp it with the time when the session begins, which makes it globally unique. The proxy will communicate the session ID to its server, which learns to pick up messages with this particular stamp. Looking at Figure 3, when the client is in position h it can talk to two servers on $H3$ and $H4$. The server running on $H3$ will pick up the tuples generated by the proxy running on the client because it can match the session ID.

The context sensitive binding handles redirecting the traffic to a new server. That is, it makes a new server pick up messages and service requests in a manner transparent to the client. If the new server is the result of migrating the old server onto a new host, the state information the server carries with it will include the session ID. If a new server picks up the task and continues the work, the light migration that ships the state of the computation and partial results will also transfer the session ID, and therefore know to pick up and service requests that come in the corresponding tuples. In the current implementation, if there are multiple servers available, the context-sensitive binding mechanism chooses the one with which the connection is guaranteed for a longer period of time (based on an analysis of the motion profiles of the carrier hosts and wireless communication range).

5.5 Demo application

We developed an application as proof of concept. Figure 4 shows a map and 5 mobile hosts on a street. The experiment follows the scenario described in Figure 3. The client drives in a car and meets other cars on the way. The client wants to record a radio show which it cannot receive in its car. The client deploys the software, which records the show in MP3 format, onto $H1$, the first car that the client encounters that is capable and willing to run the service on client's behalf. As $H1$ is about to go out of client's range, the service is migrated onto $H2$ where it continues to record the show. The client disconnects from $H2$ and before meeting $H3$ misses a significant part of the broadcast but resumes recording the show with $H3$'s help. When $H4$ comes into range, the middleware discovers that $H4$ already runs a recording service and transfers the computation state to $H4$. The show ends while the client and $H4$ are in contact, when the client obtains from $H4$ a MP3 file containing the radio show, less the intervals spent during migration.

Figure 4 captures the client in a position equivalent to e in Figure 3. The positions and the motion profiles of the 5 hosts emulate the scenario above such that we can test the types of migration, context-sensitive binding and location agnostic communication protocols described in the paper. The progress bars in the four dialog boxes indicate partial progress on each host. We simulated the application indoors but we used 5 hosts and forced all interactions to be as if the hosts were in real motion. The reason is the ratio between the wireless communication range of the IEEE 802.11 cards we used (about 100m) and the errors of the GPS readings (the advertised error is less than 10m but we found it to be around 25m). We developed a virtual space simulator (VSS) for location information. This is a server that runs on a machine and feeds location information to the clients that connect to it. We configured the motion profile of each host into VSS, which periodically tells each host where it is and how it moves at each moment. The middleware running on each host makes the decisions about migration and binding based on this location information received from the VSS server, which makes our indoors simulation be realistic. The servers encoded a radio show from an Internet live feed, which does not affect the validity of the simulation.

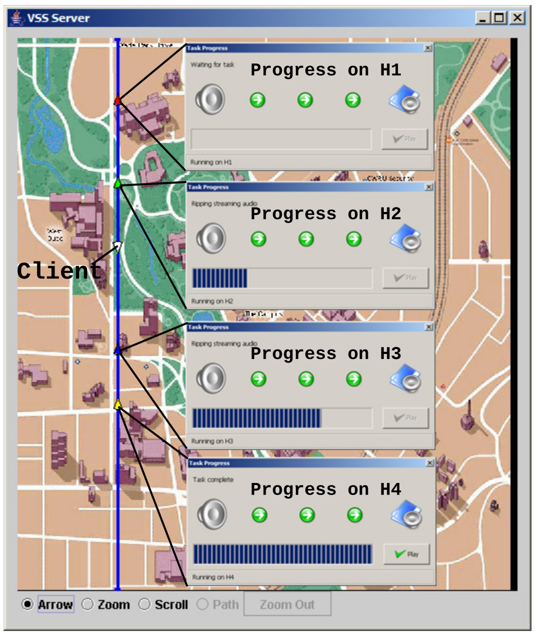

Figure 4: Mobile client and servers.

6. DISCUSSION AND FUTURE WORK

An issue we did not address in this paper is related to the security of inter-host collaboration. There are concerns about accepting code from another host to be executed locally. Certain restrictions have to be applied such as the downloaded code should not have access to the local file system or to certain parts of the system's memory, similar to the restrictions imposed on Java applets. The code uploaded by a client onto a server can also be the target of the server's curiosity and the client may want to protect the service's code from a malicious server's introspection. A solution for this is presented in [19] where the authors describe a method to compute using encrypted functions (i.e., the code uploaded is encrypted). The tensions between the servers' concerns about accepting code for execution and the clients' concerns about not giving away code need to reach a balance where both parties are satisfied with the level of security.

Another important issue is to figure the time needed to complete a migration so we can start it early enough. Factors influencing this time interval are the speed of the source and destination hosts, the size of the data being transferred (code, input data, partial results), the computation power of the two machines, the bandwidth of the wireless link, etc. The motion profiles of the two hosts and most of the other parameters can be obtained at runtime (e.g., the bandwidth can be obtained from the wireless cards driver, the size of the data to be transferred can also be determined relatively easy, etc). Another challenges is evaluating how much time it takes for a piece of code to execute on a certain machine. This cannot be statically determined as the load of the processor varies at run time and there are no processor drivers we could interact with (like we can read the bandwidth of a link from the network card driver). A way to tackle the issue is to have a sample piece of code used as unit of measure and report the execution of the rest of the code to this unit.

Thus, after many runs, a certain procedure can be labelled as taking x units to complete for a certain input (recursive calls, different sizes of the input, or waiting for input or in synchronized calls can affect the evaluation). The sample piece of code could be run in the background from time to time and by measuring the time it takes to execute we can get a sense of the system's load and the execution speed of the application's code in real time.

Related to process migration, we mentioned that a process should not migrate while holding locks on shared resources. On the new host the locks are invalid while on the old host the locks continue to block the resource until the operating system or Java Virtual Machine releases these locks by force. It is part of our future work to develop a mechanism that releases the locks automatically before migration and resumes execution of the process on the the host by redeclaring the interest for those locks. The process would thus compete for the locks like any other process that has always been running on that host. To do this, we need to develop our custom locks which will be recognized by our middleware such that they can be manipulated automatically.

A "follow-me" session accompanies the client until the task is finished. It resembles a cloud spanning the client host and one or more other hosts where services of interest to the client may be running. An approach similar to ours could be developed based mobile IP [20]. MobileIP allows for a host to "drag" it's IP address along as it moves in space. The work was developed for wired networks where the same host would connect to the network from different places or would cross network boundaries. The mobile node uses two IP addresses: a fixed home address and a care-of address that changes at each new point of attachment. From each new point of attachment the host communicates "home" its new address. The traffic destined for this host will always be directed towards the home address. At this location, a server relays the packets to the IP address the mobile host has registered. The approach is interesting because it exhibits the location-agnostic characteristic. It supports mobile computing but does not entail software migration and also assumes Internet (or at least LAN) connectivity for the long distance routes the packets may have to follow.

7. CONCLUSIONS

In conclusion, we presented a middleware architecture that supports SOC in mobile ad hoc networks. We defined the "follow-me" session which follows the client while the task at hand is under processing. To cope with challenges of ad hoc networking and transparently maintain the session with the entities involved, we employ strong process migration, context-sensitive binding and location-agnostic protocols. We delivered stop/transfer/resume computation semantics and disconnection and migration-proof coordination among participants.

Acknowledgements: This research was supported by the Office of Naval Research under MURI research contract N00014-02-1-0715. Any opinions, findings, and conclusions expressed in this paper are those of the authors and do not necessarily represent the views of the research sponsors.

8. REFERENCES

[1] W3C-XML-Activity-On-XML-Protocols: W3c recommendation: Web services description language 1.1. http://www.w3.org/TR/wsdl (2003)

[2] Horrocks, I.: DAML+OIL: A description logic for the semantic web. IEEE Bulletin of the Technical Committee on Data Engineering (2002)

[3] XML-Core-Working-Group: W3c recommendation: XML version 1.0 second edition. http://www.w3.org /TR/2000/REC-xml-20001006 (2000)

[4] W3C-Semantic-Web-Activity: Worldwide web consortium page on resource description framework. http://www.w3.org/RDF/ (2003)

[5] Sen, R., Handorean, R., Roman, G.C., Gill, C.: SOC Imperatives in Ad Hoc Wireless Settings. In: Service-Oriented Software System Engineering: Challenges and Practices (to appear). (Idea Group)

[6] Waldo, J.: The Jini Architecture for Network-Centric Computing. Communications of the ACM **42** (1999) 76–82

[7] Edwards, W.K.: Core Jini. Sun Microsystems Press (1999)

[8] Handorean, R., Roman, G.C.: Secure service provision in ad hoc networks. In: Proc. of The 1^{st} Int'l Conference on Service Oriented Computing (ICSOC 03). Number 2910 in LNCS, Springer-Verlag (2003) 367–383

[9] Johansen, D., van Renesse, R., Schneider, F.B.: An introduction to the TACOMA distributed system—version 1.0. Technical Report 95-23 (1995)

[10] Baumann, J., Hohl, F., Rothermel, K.: Mole - concepts of a mobile agent system. Technical Report TR-1997-15 (1997)

[11] Picco, G.P.: μCODE: A Lightweight and Flexible Mobile Code Toolkit. In: Proc. of Mobile Agents: 2^{nd} Int'l Workshop MA'98. Volume 1477 of LNCS., Springer Verlag (1998) 160–171

[12] Peine, H., Stolpmann, T.: The architecture of the Ara platform for mobile agents. In: 1^{st} Int'l Workshop on Mobile Agents. Volume 1219 of LNCS., Springer Verlag (1997) 50

[13] Gray, R.S.: Agent Tcl: A flexible and secure mobile-agent system. In: Fourth Annual Tcl/Tk Workshop (TCL 96). (1996) 9–23

[14] Claypool, M., Finkel, D.: Transparent process migration for distributed applications in a Beowulf cluster. In: Proc. of the Int'l Networking Conf. (2002)

[15] Handorean, R., Sen, R., Hackmann, G., Roman, G.C.: Automated code management for service oriented computing in ad hoc networks. Technical Report WU-CSE-2004-17, Washington University Department of Computer Science (2004)

[16] Gelernter, D.: Generative communication in Linda. ACM Trans. on Programming Languages and Systems **7** (1985) 80–112

[17] Murphy, A., Picco, G., Roman, G.C.: LIME: A middleware for physical and logical mobility. In: Proceedings of the 21^{st} International Conference on Distributed Computing Systems. (2001) 524–533

[18] Huang, Q., Julien, C., Roman, G.C.: Relying on Safe Distance to Achieve Partitionable Group Membership in Ad Hoc Networks. (To appear in IEEE Trans. on Mobile Computing)

[19] Sander, T., Tschudin, C.F.: Protecting mobile agents against malicious hosts. LNCS **1419** (1998) 44–60

[20] Perkins, C., Myles, A.: Mobile IP. In: Proc. of the Int'l Telecommunications Symposium. (1994)