Context Aware Session Management for Services in Ad Hoc Networks

Radu Handorean, Rohan Sen, Gregory Hackmann, and 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
Email: {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 interactions occur between a client and a service, rather than a specific provider or server. We allow the client to switch service providers, if needed. 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.

I. INTRODUCTION

Mobile devices today have limited computational power and persistent storage. In scenarios which require the execution of a small piece of code which is computationally intensive, e.g., running a public/private key encryption algorithm, the mobile device is stymied by its own lack of computational power. Similarly, in scenarios which are storage intensive, e.g., decompressing an MP3 file to a raw wave file, the limited storage is an undesired restriction. However, most modern mobile devices have built-in 802.11b wireless capability that allows them to communicate with proximal hosts. This capability on a reference host can be used to leverage off the capabilities of proximal hosts to achieve the required goal, e.g., CPU intensive code can be pushed over the wireless link onto more powerful hosts who execute the code and return the result to the caller. Similarly, large files can be stored on neighboring devices which have larger storage media or programs can be discovered, copied to the reference host on demand, used and then discarded, thereby alleviating the problem of limited storage. In both cases, the code another host (the service provider) runs for the client is called a service. Observe that the client controls the service even if it is executing on a remote host.

Allowing such behavior requires solving a key problem. Physical mobility of the devices coupled with the modest range of 802.11b wireless cards results in limited intervals of time when two devices can communicate with each other. This is especially true in Mobile Ad Hoc Networks (MANETs), which are the target environment for our work. However, an interaction (defined as some bounded sequence of messages exchanged) between two hosts may need more time to complete than the interval of connectivity between them. For reliable service provision, it is imperative that an interaction between the client and the service provider, once begun, reaches completion. In MANETs, physical movement of hosts is independent of application semantics and it is undesirable for the application to impose mobility restrictions. One solution is to have the client partially complete the task with the help of some host, pause its work, and resume it on another host. This stretches the processing of a task over multiple hosts as they fall within the client host’s communication range, each contributing pieces of computation towards finishing the entire task. This adds a degree of complexity, which we mask by introducing a layer of abstraction we call a follow-me session, defined as a mechanism that preserves the aura of interaction between a client application and a service by masking the disconnections between intervals of connectivity.

The basis of the “follow-me session” idea is that there must be a persistent link between the client and the service, rather than specific hosts that provide the service. Hence, to accommodate transient host connectivity, we adopt two strategies: 1) the client can automatically connect to a new host as the current provider disconnects and a new one comes within communication range, assuming that the new provider offers the same functionality, and 2) the process offering a service can migrate from host to host, ensuring that it is always on a host that is within communication range of the client’s host as it moves through space. In essence, follow-me sessions offer, within limits, the continuity of service provision: the mechanics of having the service follow the client host are handled in a manner transparent to the client application using strong process migration, a novel concept called context-sensitive binding, and location agnostic communication protocols.

The remainder of the paper is organized as follows. Section II presents background material and core concepts associated with the “follow-me” session. Section III describes the server migration mechanism. In Section IV we discuss the context sensitive binding mechanism. The implementation of the system is detailed in Section V. Section VI discusses the contributions of our work, and plans for future work. We conclude in Section VII.
II. BACKGROUND AND CONTEXT AWARE SESSION MANAGEMENT

In this section, we provide a brief overview of service-oriented computing (SOC) before describing the concept of “follow-me” sessions and how they can enhance stability of services in MANETs.

A. Service-oriented computing architectures

In SOC, a service provider advertises some functionality it wishes to share by placing a service advertisement in a publicly available service directory. Clients discover and use these functionalities at run time. SOC also decouples the interface for accessing some functionality from its realization.

Currently, there are two main approaches to SOC: proxy-based services and Web services [1]. In proxy-based SOC, the service advertisement is in the form of a proxy object which can be retrieved by clients and used as a local handle to the service process on a remote server. In some cases, the proxy itself delivers the entire functionality of the requested service. Normally, the proxy tunnels requests to the provider’s server. The proxy-server communication is abstracted from the client (all calls to the proxy appear to be local) but is relevant to our work because we address mobility concerns from its realization.

In Web Services (WS), the client interacts directly with the service which requires the client to be aware of the interaction protocol and the location of the service. Strict enforcement of standards ensure that clients do not need to remember multiple protocols. Web services use XML [2] for syntax, the Resource Description Framework (RDF) [3] to describe entities and relations, and Web Services Description Language (WSDL) [4] to describe what data is sent across the wires.

Both approaches are targeted to wired infrastructures, evidenced by centralized architectures that are not viable in more dynamic networks. Adapting these approaches to MANETs is more complex than a simple port of the software. In [5] we have motivated the use of proxy-based services over WS in MANETs, some of the reasons being flexibility, adaptability, and the ability to accommodate thin clients.

Jini [6] is an implementation of the proxy model, targeted towards a wired and fairly reliable networks. This is evidenced by its use of a centralized service directory which is a single point of failure for the system. Jini’s leasing mechanism, cleans up orphan advertisements which remain in the service repository after their lease expires because of to non-renewal due to disconnection or choosing not to offer the service anymore. The leasing mechanism prevents orphan advertisements, but does not solve certain consistency problems.

In MANETs, there is no fixed infrastructure - the network is comprised of member hosts only. Centralized architectures fail because the availability of a service directory on a certain host cannot be guaranteed when hosts are mobile (hosts depend on the directory being accessible to find services). Two scenarios are shown: Figure 1(a) is a consistency issue - a client can read an advertisement in the directory for a service that is not accessible, while in Figure 1(b), the client is in range of the service but cannot access the directory which informs him of the existence of the service.

As long as two devices are connected (directly or multi hop), they can exchange information and interact as if they were wired. In MANETs, a multi-hop connection is seldom stable since host mobility can change the topology of the network at a very fast rate. Adopting the proxy model mitigates network communication and RMI issues, facilitating the use of thin clients. However, disconnections continue to be a problem. This paper seeks to mitigate the challenges of dealing with disconnection by introducing mechanisms that abstract frequent disconnections inherent in MANETs by pushing them down into the middleware layer.

B. “Follow-me” sessions

Traditionally, a session is a lasting connection between a client and a server during which several packets of data are exchanged. Usually, a session is not interrupted by a disconnection. In “follow-me” sessions, the lasting connection is between a client and a service, rather than a specific host. Since the hosts can move in and out of communication range, the “follow-me” session can span multiple connectivity intervals. This can be due to the session involving 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 the server implementing the service may be mobile code relocating to follow the client as it moves in space. Also note that during a “follow-me” session, the client (in actuality the service proxy used by the client) interacts with its server sporadically (e.g., during methods calls and returns). Thus, the protocol stack need not enforce a live connection if it is not being used (i.e., if a disconnection occurs when nobody is talking, the application should not perceive it).

Several important elements are required to manage a “follow-me” session in MANETs. An 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 is considered to

![Fig. 1. (a) The service discovered is unreachable. (b) The service is reachable but a lookup is not possible.](image-url)
be better. While better can be defined in many ways, in this paper we consider the time to disconnection between the two hosts as the definition of better. Another important issue is to be able to move the implementation of the server to which a proxy connects from one host to another, so as to remain in range of the client over time. 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 is also required. We call this a location agnostic communication protocol. Since the transfer of an ongoing computation as well as source data deployment and (partial) result retrieval affect the computation and transfer time, we need to evaluate the time this overhead takes and factor it along with connectivity data into the session management algorithms.

![Fig. 2. Follow-me session.](image)

A summary of the key challenges we face is presented in the scenario depicted in Figure 2. 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 service moves; a variation of this scenario might allow the client to simply discover the service 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 service 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 to which the service could jump and therefore the client will “take back” the service, computation state, and partial results (the dashed arrow from H2 to the client). The client cannot run the service on its own host because the resources available on that host do not allow it and therefore the client will only transport the service 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 service process, and have the new server “resume” from a predefined intermediary progress point. While the client interacts with H4 the task is completed, and the results are shipped back to the client.

To deliver “follow-me” sessions we developed three mechanisms: a) strong process migration of the server thread for situations when the service is not offered on a certain host, b) context-sensitive binding, and c) location-agnostic communication protocols to smooth the transition from one provider to another (via migration or re-binding).

III. SERVER THREAD MIGRATION

Process migration consists of code migration which transfers the binary code, object state migration which transfers the state of objects’ instance variables, and execution state migration which transfers the program counter and call stack content, all to the target host.

A. Types of migration

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

Weak code migration requires that the process can run, but not resume, on a destination host. This involves making the code available on the destination machine 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 [7], [8]. 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 restarted, except that the memory is initialized to contain potential partial results (e.g., \(\mu\text{Code}[9]\)).

Strong code migration [10], [11] entails the migration of the execution state as well. Processes can be stopped, transferred, and resumed on a new host. To deliver the desired semantics, we developed a strong migration mechanism, implemented for Java threads. While capturing and transferring the execution state, the program counter of the Java virtual machine (JVM) and the call stack are captured and transferred in a serializable format to the new destination. Strong mobility is more powerful but it is also more expensive to deliver.

Ideally, the migration would occur in a manner completely transparent to the subject process. However, this is extremely dangerous since a process could be transferred at a moment when it held locks on resources. Without support from the operating system, these locks would never be released, since the owner process would not continue to exist on the system. A system that achieves transparent process migration, in cooperation with the operating system is [12]. In our case, the JVM 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 or application programmer. 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.

B. The Migration Mechanism

We deliver our strong migration in two forms: lightweight and heavyweight. Lightweight migration entails only the transfer of state information. In Figure 2, when the client is in...
position \( e \) or \( h \), between \( H2 \) and the client (in position \( e \)) or between \( H3 \) and \( H4 \) (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 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 \( H1 \). Likewise, in position \( c \) when the server migrates from \( H1 \) to \( H2 \), a heavyweight migration is needed since \( H2 \) does not offer the required service either.

During migration, the serialization process wraps only the values of member variables of an object and not the bytecode from which the object was created. This includes all objects inside the initial object. Therefore, a separate mechanism to transfer the bytecode for each object along with its dependencies to the destination host is required. We developed such a mechanism, described in [13] which automatically scans an object and ships its bytecode and the bytecode of all its dependencies to the target host.

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. The service on the “old” server continues to run until it reaches the next checkpoint. As shown in Section V, a Java thread cannot be reliably stopped from the outside. Thus, our middleware sets a flag which the thread verifies every time it enters a checkpoint. Thus, the termination initiative appears to belong to the service itself.

C. Checkpointing and state saving

Checkpointing improves the fault tolerance of software systems by saving the current state of a program and partial results to non-volatile storage. Thus, if a program is interrupted by some event beyond its control (e.g., hardware failure), it can be restarted at the last checkpoint without losing the processor time invested before the last checkpoint was encountered. We use checkpointing to protect against hosts’ disconnecting during a client-server interaction. If a disconnection occurs, the task has to be completed during a subsequent window of connectivity between the client and another host running an instance of the same service. Checkpoints allow the interaction to resume from an intermediary point, preventing waste of processor time.

At each checkpoint we record the execution state of the thread, i.e., its program counter and call stack. The standard JVM does not expose any information about the program counter. Hence, we introduced an artificial program counter, which is updated at each checkpoint. Its value is transferred to the destination host and is used to resume the execution. The data state, composed of the values of instance variables (live objects) are easily transferred as serializable objects.

When the server migrates from one host to another, the state recorded at the last checkpoint visited is transferred. Any further computing since the last checkpoint is lost. For example, if the checkpoint is placed just before a \( \texttt{for} \) loop, the loop will be restarted from the beginning. 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. Finally, note that 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 it an acceptable constraint.

IV. CONTEXT SENSITIVE BINDING

In this section, we present another strategy for “follow-me” sessions through context-sensitive binding (CSB), a novel mechanism that decouples the interface of a service from its realization. The realization 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 highest remaining battery power” or “the server that is physically closest to the client.” The server change occurs dynamically and transparently in a context-sensitive manner. Using Figure 2 as an example, if we consider time to disconnection as the criteria for switching servers, then once \( H4 \) comes within the client host’s communication range the CSB mechanism will disconnect the proxy from the server on \( H2 \) and connect it to the server on \( H4 \).

A. Novel features of context-sensitive binding

Policy based selection—The set of qualifying providers is chosen based on programmer specified policies, e.g., “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 server, 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 discovery step) or that offer to run code provided by the client. Other policy parameters may include security constraints (e.g., cannot print secret documents on any printer), geographical restrictions imposed on the provider’s location (e.g., do 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—CSB provides for dynamic switching between servers to offer 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. From the client’s perspective, the binding appears continuous for the time interval it uses the service.
B. Mobility and context-sensitive binding

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

In MANETs, the primary reason for context changes is physical mobility. As hosts move in space, they encounter a changing set of hosts over time. Since other hosts too move in space, the set of hosts that the client host is connected to changes continuously. Given these changes, the CSB 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 Limone [14] 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 minimize or maximize a parameter. Support for more complex metrics is 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 III) 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. If it is determined that the server needs to be switched, the middleware preempts the interaction between the client and the server. 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 III, the calls are directed to the new server. This is achieved using location agnostic protocols, which are described later in this section. The rebinding to a new host can be influenced by other parameters than those related to the physical movement of hosts. Consider a printer running low on paper. If the client prints multiple documents, at some point the CSB 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 when there are licensing issues. Rather than move the process, we simply switch to another instance of it, allowing the client to continue its work without raising issues of security and ownership of code.

C. Location agnostic protocols

A crucial element in the delivery of the follow-me session management is the communication protocol between the client and the server. All these interactions logically belong to the same “follow-me” session. Thus, the client side should not perceive disconnections from the 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, 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 transparent to the client.

To deliver this, we employ location-agnostic communication protocols. The client uses a unique session identifier which is used to stamp all messages exchanged in a “follow-me” 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 [15] coordination model.

The location agnostic communication protocol we employ uses the concept of content-addressable communication. Essentially hosts (clients or services) exchange messages based on their content. Hence, messages do not have to be addressed to a specific host or network location. If a process migrates from one host to another, it will receive a message intended for it if it uses the same criteria (in terms of content of the message). Similarly, if a client uses CSB to connect to a similar instance of a service, it is not unreasonable to assume that it will have the same criteria as the original instance of the service that was used, allowing messages to automatically be propagated to the new instance of the service.

V. IMPLEMENTATION

“Follow-me” sessions have been implemented in Java, using Limone [14] as a middleware to handle the implications of a MANET, i.e., physical mobility of hosts. Here, we present a brief overview of Limone, a description of the implementation and a proof of concept demo.

A. Limone overview

Limone is a Java implementation of the Linda [15] co-ordination model, designed for MANETs, which masks details associated with coordination and communication from the application programmer. A host running Limone runs a LimoneServer supporting one or more agents, which are
analogous to application modules. Coordination in Limone occurs via transiently shared tuple spaces. Every tuple space in Limone is identified by a name. Tuple spaces having the same name are 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 a 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), and 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). Limone offers probe variants of the traditional blocking operations (inp, rdp), and group operations (outg, ing, rdg, rdp, and inp). 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, Limone, 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.

B. Checkpoints

The implementation of strong code migration presents several technical problems. (1) the standard JVM does not allow programs to save or restore the program counter, (2) 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, and (3) 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.

The application programmer creates mobile applications by extending the MobileThread class, which adds several methods and fields to Java’s standard Thread class (described below). The programmer defines checkpoints by calling the addCheckpoint() method. While appearing to the programmer to be an ordinary method call, it serves as a marker in the bytecode to indicate the location of checkpoints. After compiling the Java source code, the resulting bytecode is passed into the bytecode rewriter which converts the code so that it is capable of being strongly migrated.

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 are 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 this is the case, 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 sets 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 unpacing it.

C. Migration

When migrating applications across hosts, it is likely that they will not have their bytecode available at destination. The solution to this problem is an automated code management system. In the interest of space, we omit details of analyzing the source code and discharging code which can be found in [13]. Instead, we focus on its application to “follow-me” sessions.

When an application is migrated, our middleware attempts to deserialize it. If this fails due to missing bytecode, the middleware catches the ClassNotFoundException and fetches the needed bytecode. This is done by using a custom classloader and a custom ObjectInputStream which 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. If this read operation succeeds, the class loader loads the JAR package contained in this tuple into memory.

D. 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. In our implementation, each proxy-server pair stamps its 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 2, 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 matches the session ID.

The CSB handles redirecting the traffic to a new server. That is, it makes a service instance on the new server pick up messages and service requests in a manner transparent to the client. If the service instance is the result of migrating the old service onto a new server, the state information the service 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).

E. Demo application

Figure 3 shows a map and five mobile hosts on a street. The experiment follows the scenario described in Figure 2. 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, except for the intervals spent during migration.

Figure 3 captures the client in a position equivalent to $e$ in Figure 2. The positions and the motion profiles of the five 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 dialog boxes indicate partial progress on each host. We simulated the application indoors but we used five 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 (on average found to be around 25m). We developed a virtual space simulator (VSS) for location information. The VSS 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 indoor simulations realistic. The servers encoded a radio show from a live feed.

VI. 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, e.g., the downloaded code should not have access to the local file system and certain parts of 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 inspection.
A solution for this is presented in [16] where the authors describe a method to compute using encrypted functions. 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. This concern does not fall within the scope of this paper.

Another important issue is to determine the time needed to complete a migration so we can start it early enough. Factors influencing this time interval include 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, and 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 challenge 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. A solution is to have a sample piece of code used as unit of measure and relate the execution of the rest of the code to this unit. Thus, after many runs, a certain procedure can be labeled 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 could get a sense of the system’s load and the execution speed of the application’s code in real time.

We previously 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 JVM 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 host by competing for the locks like any other process that has been running on that host. To do this, we need to develop custom locks, recognizable by our middleware so that they can be manipulated automatically.

A “follow-me” session 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 on mobile IP [17]. Mobile IP allows for a host to “drag” its IP address along as it moves in space. This 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 to its “home” its new care-of address. The traffic destined for this host is directed towards the home address. At this location, a server relays the packets to the care-of address. The approach is location-agnostic and supports mobile computing but does not entail software migration and assumes stable connections.

VII. CONCLUSIONS

We presented a middleware implementing “follow-me” sessions, used to support SOC in MANETs. We described the core features and concepts of “follow-me” sessions. To cope with the challenges of ad hoc networking and to transparently maintain the session with the entities involved, we employ strong process migration, context-sensitive binding and location-agnostic protocols. The main contribution of this paper is a programming abstraction and accompanying middleware that simplifies application development in volatile settings and provides 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.

REFERENCES