KNOWLEDGE-DRIVEN INTERACTIONS ACROSS MOBILE AD HOC NETWORKS

ROHAN SEN, RADU HANDOREAN, GRUIA-CATALIN ROMAN, GREGORY HACKMANN, AND CHRISTOPHER GILL

Department of Computer Science and Engineering
Washington University in St. Louis
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130, U.S.A.

The decoupled nature of computing in mobile ad hoc networks (MANETs) can result in disconnections at inopportune times during an interaction between a pair of hosts. We introduce the notion of a priori selection of partner hosts to reduce the likelihood of disconnection during such interactions. An application may specify the times when and the physical locations where it requires interactions with peer applications on partner hosts. A knowledge base of the physical motion profiles of various hosts maintained on each mobile host is used to select partner hosts that are co-located with the reference host at the required time and are least likely to disconnect. In this paper, we present a formal model for such knowledge management, along with an algorithm used to determine suitable partner hosts. We also provide details of our implementation of partner selection, which has been used in the context of a service-oriented computing middleware for MANETs, developed previously by our group. Finally, we present simulation results of our approach.

Keywords: Mobile Computing, Service Oriented Computing, Models, Software

1. Introduction

While the popularity of mobile computing continues to grow rapidly each year, it is our opinion that the full potential of mobile computing is still far from being realized. This is in part due to the fact that most mobile devices today are used in strictly nomadic environments, i.e., environments where mobile devices use their wireless capabilities to connect to an existing wired infrastructure through wireless gateways, and connections stay up for long periods at a time, e.g., in an 802.11b compliant WaveLAN. Mobile ad hoc networks (MANETs) on the other hand are dynamic, open environments characterized by opportunistic interactions among hosts. Connections between hosts are available only for short periods of time and disconnections are common. In addition, MANETs do not rely on any external infrastructure; the network infrastructure is borne completely by the participating hosts. It is in such environments, free from dependence on any fixed infrastructure, that we must address the challenges and exploit the opportunities associated with mobile computing, such as: perceiving and reacting to rapidly changing physical and computational contexts; interacting and working with hosts that have never been

encountered previously; and exploring new application domains that were previously not feasible due to the limitations of current mobile device technology.

While MANETs represent an exciting new frontier for computing, they also pose many challenges. In MANETs, hosts are small portable devices that are constrained in terms of processing power, memory, and battery life. Such devices can communicate via a wireless radio that has a limited range (typically 25 to 100 meters depending on the environment and obstructions in the area). This limited communication range, combined with the fact that hosts are typically mobile, results in an environment where hosts may constantly move in and out of communication range with each other, making unpredictable disconnections among host pairs a fact of life. Engineering distributed applications that survive in such volatile environments is a challenge that has been addressed only to a limited extent. Current solutions to this problem involve attempting to eliminate the possibility of disconnection. For example, in a *hoarding* strategy, the code for an entire service implementation is copied to the client machine and is used locally so that disconnections cease to be a factor. However this strategy cannot be employed in cases when the software footprint is very large or when the code is proprietary. *Nomadic* strategies also assure connectivity because they assume that mobile hosts will stay within communication range of a set of access points attached to a wired infrastructure. Such strategies are limiting, as they impose bounds on the physical mobility of hosts in a manner which is often neither practical nor desirable.

Our approach does not seek to eliminate the possibility of disconnection. Instead, we focus on choosing partner hosts carefully so that a disconnection does not occur at a critical juncture. In this paper, we introduce knowledge-driven interactions between hosts in MANETs as a strategy that exploits the spatiotemporal aspects of a distributed application's requirements to carefully select partner hosts that are likely to remain in communication range for the necessary duration. This strategy thus reduces the likelihood of a disconnection at a crucial juncture, without compromising other aspects of program execution or host operation. The essential idea is to use knowledge about hosts' physical motions to compute the time at which two hosts are likely to be within communication range for a reasonable interval of time. This information is then matched with the application's *requirement profile*, which is a list of applications to be provided by remote hosts and the intervals of time when they are required. The result is a proactively planned *satisfying set*, a list of specific peer applications that are resident on hosts that are most likely to be within communication range of the client at the times that they are required, and which will remain within communication range for the projected duration of that need.

While our model for and implementation of knowledge management can be used in any setting, in the interest of a focused presentation, we discuss the knowledge management system in the context of a service-oriented computing middleware for MANETs. The contributions of this paper can be summarized as follows:

- A **formal model for knowledge management in MANETs** and a formal specification of the *the satisfying set* under different assumptions of host and code mobility.
- A **software architecture for knowledge management** for building, maintaining, and managing a knowledge base on each host.
- A **simulation evaluation** of our approach.

The remainder of the paper is organized as follows. Section 2 provides background information on service-oriented computing and covers related work in that area. Section 3 describes our computational model and provides a formalization of knowledge management and of a satisfying set, with respect to that computational model. Section 4 presents our software architecture and details of our implementation. Simulation results appear in Section 5, and we offer concluding remarks in Section 6.

2. Background

In this section, we provide background information on the service-oriented computing (SOC) paradigm and motivate our choice of a proxy-based SOC architecture for the MANET environment. We follow this with a presentation of selected related work in the field.

In the SOC paradigm, a *service provider* advertises services, which represent some capability that it is willing to share. These *service advertisements* are placed in a publicly accessible *service directory.* Interested clients browse this service directory for suitable services. When an appropriate service is found, the client obtains an address (potentially a local handle) for the service. The client can then use the service directly.

2.1. *The Case for Proxy-based SOC for MANETs*

Most SOC architectures, such as the Service Location Protocol (SLP) [9], Jini [17], Salutation [14], and those employed for Web Services (WS)[1] [12] are designed for infrastructure-rich centralized wired networks and, as such, are unsuitable for MANETs. For example, in WS, the service directory is maintained at a central well-known location which is acceptable in the environment of the World Wide Web but unacceptable in a MANET where a particular host may not be accessible to all other hosts as it can move out of communication range or shut down. In addition, it is not reasonable to impose on a single, resource poor host, the responsibility for handling all directory functions.

The addressing scheme used in WS is also geared toward a reliable network. WS uses a uniform resource identifier (URI) to indicate the logical location at which a service may be accessed, and relies on the DNS infrastructure of the Internet to map the address to a physical machine. In MANETs there is no DNS server and a logical address may not necessarily map to a device at a particular physical

location due to the movement of hosts, thus rendering this approach ineffective. SLP takes some steps towards accomodating mobility through a special mode of operation that does not require a functioning directory agent, which is analogous to a distributed service directory. In this mode, services and clients directly seek each other out using a peer-to-peer model. However, this approach still does not solve problems of addressing and service access.

In our experience, the most effective kind of SOC model for MANETs is a proxy-based model. In a proxy-based model, a proxy object is included in the service advertisement. When a client retrieves the advertisement, it obtains the proxy which it installs locally. The proxy then becomes a local handle to the service on a remote machine. The client can then interact with the service by making local method calls on the proxy, which then delegates the requests to its parent service. The idea of proxy-based service oriented architectures was first proposed in Jini [17] which was designed for wired networks. It has since been adapted for use in MANETs as was shown in [5].

Proxy-based architectures are especially effective in MANETs for two reasons: (a) They help reduce the amount of software required on the client side, thereby making thin clients possible; and (b) they abstract details of the protocol to be used between the client and the service provider. Since the proxy is a self contained piece of code that can communicate with the provider host, the client is not required to be aware of the communication protocol or to possess any other code to communicate with the provider. The client is only required to carry the code that allows it to browse for services and discover proxies. This results in a small footprint for the client software, which is useful when running such software on mobile devices. The fact that proxies abstract the communication protocol is especially useful in MANETs, where standardized application level protocols are not prevalent. Hence, the abstraction of a heterogenous set of protocols by proxies allows a large set of hosts to communicate with multiple service providers without the overhead of needing to know the specific protocol for each provider.

Though proxies are a solution to certain problems associated with SOC in MANETs, their usage does raise certain issues, some of which we have addressed in our previous work. In [4], we proposed an automatic code management system which transparently ships and installs proxy code on the client host (once the client has declared an interest in the service) as a solution to the problem of distributing the binary code required by clients to execute proxies. A proxy upgrade system [15] ensures that these proxies are upgraded transparently at run-time with very little impact on the client application, so that the proxy software is kept consistent with upgrades to the software on the provider host without the client application having to handle this procedure explicitly. These mechanisms, which are portions of a larger system supporting SOC in ad hoc networks solve some of the issues associated with proxy-based SOC architectures. An important remaining problem–that of ensuring that interactions between the proxy and the service are interrupted to the least extent possible–is the subject of this paper.

2.2. *Related Work*

Our work in this paper encompasses various topics such as meta information management, information gathering and dissemination, information semantics, and planned behavior. As such, we now present a selection of related work from each of these topic areas.

Our overarching goal for introducing knowledge driven interactions to SOC in ad hoc networks was to establish a greater sense of order in an otherwise chaotic environment. One method of achieving this is to use the notion of perception of the environment to make decisions, as in the Task Control Architecture (TCA) [16] designed for autonomous agents that control robots. Among other things, the TCA uses information about the environment to ensure that the robot is not in any physical danger. Our approach performs a similar kind of knowledge aggregation, though it is used to protect clients of a service from unexpected disconnection.

All knowledge aggregation and dissemination that must be done to support knowledge driven architectures occurs at a meta-level, in the sense that the knowledge that is traded is independent of the particular applications that are running on the individual hosts. Thus, meta-information management and the structure of meta-information structures become key facets of the system. One system proposed by Costa et al [2] uses a centralized type repository to maintain meta data. However, such a centralized solution creates a potential single pointof failure. Because of this, we chose to use multiple decentralized repositories called *knowledge bases* on each host so that there would be a decreased probability of failure.

Another issue that relates to our work is information dissemination. Essentially, to support knowledge driven interactions, each host must disseminate knowledge about itself so that others may react to it. In [8], the authors propose three schemes for information dissemination that are especially tailored to mobile ad hoc networks, and which focus on rapid dissemination of information without duplication. Three strategies - *select then eliminate* (STE), *eliminate then select* (ETS), and a hybrid of the two, are described as ways to ensure that only the relevant hosts get the information. While this is an important concern, we chose to design a less complex system in the interest of saving computational resources and also because the tuple space paradigm we adopt (described in detail in Section 4) ensures that information is distributed to all connected hosts, which is exactly the set of hosts to which we want to disseminate knowledge.

Finally, there is an inherent issue of scheduling in our work, with the added restriction that such a "scheduler" must make its decisions in the presence of information about the environment, i.e., the scheduler should support the parametrization of its behavior. An example of such a scheduler is [7], implemented for the DECAF architecture. Essentially, the DECAF scheduler is able to take in functions that enable it to do planning. Two strategies are suggested. The first is contingency planning, where every possibility is evaluated and the putatively best one is chosen, and if that fails for some reason the next best option is then chosen. The second

strategy is to give the scheduler a utility function, which it can use as a metric to choose among options.

3. Formal Model

As we stated in Section 1, one of the contributions of this paper is a formal model for knowledge management. In this section, we present the details of this formalization. However, before we do so, we use a brief example to motivate the scenarios where our work is applicable. From this point on, the terminology we use is consistent with the SOC paradigm. It should be observed, however, that our knowledge management model applies more generally beyond SOC–services in our model could be easily replaced by components in a component-oriented middleware context [10] or application fragments in a path-oriented context [11].

3.1. *Motivating Example*

We consider a scenario involving cars travelling on a highway. The occupants of the cars carry PDA's which offer various kinds of services. In one of the cars, Robert is currently reading news items that were downloaded to this PDA, but would like to listen to some music in 10 minutes time, when he will have finished reading the news. Robert already has a pre-compiled playlist of his favorite songs stored on his PDA. However, his PDA must contact a streaming music service to obtain the songs. Robert instructs his PDA to have a music service discovered and ready to use at 4:10 PM, which is 10 minutes from now. The PDA queries PDA's in other cars that are near the car that Robert is in for a streaming music service. When the replies come back, it turns out that there are three cars that can offer the service, as is shown pictorially in Figure 1.

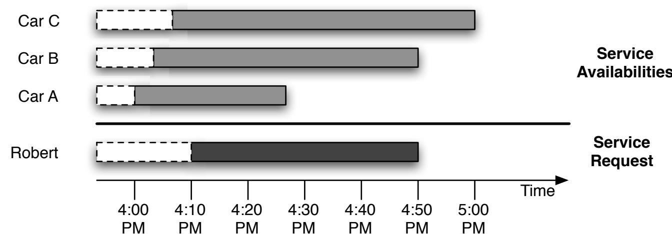

Fig. 1. A client host obtaining information from two hosts with the purpose of planning interaction patterns

Robert has also indicated that he would like to listen to the music for an hour and a half, so the service must be available for that duration. From the figure it is clear that the service offered by Car A will not remain connected for the required duration. Car B will remain connected for the duration, but it is farther away

from Robert's car than Car C, which could result in signal degradation resulting in a poor quality music stream. Additionally, since the service on Car B becomes unavailable at the exact moment that Robert's requirement ends, there is no room for accomodating errors. Hence, Robert's PDA chooses the service on Car C as the service that it will use. It stores this information until it is time to invoke the service.

The determination of the correct candidate services is done by exploiting knowledge about the other hosts in the network, i.e., their intended *motion profile*, which gives the host's projected location as a function of time. The client collects hosts' motion profiles via a gossipping protocol. It feeds the profiles it collects to an algorithm that helps it choose the *satisfying set* of services. Once the algorithm returns the satisfying set, the services in the set are ranked according to some criteria (currently we use time to disconnection from the client host as our criteria to rank services). The client connects to the service that is ranked highest from among those in the satisfying set. It should be noted that in this example, the client had only one service requirement. We expect a client application to have more than one requirement during normal operation, in which case there would be multiple satisfying sets, one per requirement. One service would be chosen from each set to satisfy the corresponding requirement.

3.2. *Formal Model*

In our model, we have a finite set of hosts (mobile devices), each distinguished by a unique identifier, that are capable of physical motion in a predetermined area. Points outside this area are considered undefined. A host may have one or more agents executing on it. Each agent represents a service, i.e., an instance of a process that offers some functionality, and is capable of logical mobility. Note that we use agents simply as units of modularity. We also assume global clock synchronization. More precisely, we define:

H - the set of identifiers for hosts in the MANET
S - the set of identifiers for services available in the MANET
L - the set of valid locations which hosts may occupy
T - the time domain
K - the global knowledge in the MANET

While K represents the global knowledge, we use $K(h)$ to denote the knowledge available on host h. More formally, we can say that:

$$K = \prod_h K(h)$$

where the global knowledge K is defined as the logical union of the various local knowledge bases on hosts in the MANET. The knowledge composition process $\prod$ is made possible by the assumption that the knowledge held by any one host is always accurate albeit possibly incomplete. For instance, a host h cannot have knowledge

about another host i in its local knowledge base K(h) that the host i does not have about itself in its own knowledge base K(i), i.e., a host cannot know more about others than others know about themselves. These and other similar restrictions are not within the scope of this discussion. A more extensive discussion of issues may be found in [13].

Having introduced the key elements of the model, we now describe how we characterize individual hosts, services, service requests and the knowledge associated with them.

Hosts. Hosts are devices capable of moving in physical space. In our model, a host possessing a unique identifier h is characterized by its motion profile $\mu(h)$ and knowledge base $K(h)$. The motion profile of a host describes that host's mobility pattern, essentially a function that takes in a time value and returns the location of the host at that time. Formally:

μ: $H \rightarrow (T \rightarrow L)$
Note that for convenience, we write $\mu(h)(t)$ simply as $\mu(h,t)$

The motion profile function can be used to express various types of mobility. For example:

$\langle \forall t : \mu(h,t) = l \rangle$ where $(l \in L)$ indicates a stationary host
$\mu(h,t) = \mu(h,t_0) + (t - t_0)v_h$ captures motion in a straight line at velocity v_h.

Similarly, more complex motions can be captured with more complex specifications for $\mu(h)$. The motion profiles of hosts are examples of knowledge since they help us understand how various para-functional (i.e., beyond the primary computation) components of the system are behaving. Motion profiles are exchanged freely among hosts using a gossiping protocol which results in an epidemic flooding of motion profile information in the network. We defer the presentation of how these motion profiles are used to a later subsection.

Services. Services are software processes that provide functionality that can be used by interested clients. Each service s is characterized by its unique identifier $(s \in S)$ and the following attributes:

$\chi(s)$ - the capabilities of the service
$\pi(s)$ - the performance attributes of the service
$\delta(s)$ - the external dependencies of the service
$\alpha(s)$ - the allocation profile of the service

The capabilities $\chi(s)$ describe the functionality of the service, e.g., a "printer" service, while the performance attributes $\pi(s)$ describe how well the functionality can be discharged, e.g. resolution, pages per minute, etc.

The external dependencies of a service, i.e., other services that are required to discharge the advertised functionality, are captured by δ, the set of dependencies.

We define:

$\delta : S \rightarrow \wp(S)$ which induces the binary relation
$x \; \overline{\delta} \; y \equiv (y \in \delta(x)) \vee \langle \exists z :: x\overline{\delta}z \wedge z\overline{\delta}y \rangle$

The first formula indicates that δ is a function from the set of service identifiers S to the power set $\wp$ of service identifiers, i.e., the dependency of any service is on zero, one, or more other services. The second formula captures the closure of the dependency function. We assume that any dependency needs to be migrated to the host on which the primary service is executing and be available for the entire duration of the service requirement. We admit that such a strategy may be considered wasteful–dependencies could be available only when they are required, i.e., during a small fraction of the service requirement interval overall, and could be used remotely. However, this complicates the presentation, and we defer a formal treatment of this issue to future work.

Since services can be logically mobile, they can be resident on various hosts over time. This is captured by the service's allocation function $\alpha(s)$, which is specified as follows:

$\alpha : S \rightarrow (T \rightarrow H)$
Again, for convenience, we write $\alpha(s)(t)$ as $\alpha(s, t)$

The function $\alpha(s, t)$ returns the host on which the service is resident at a given point in time. If a service is not logically mobile, its allocation function always maps to the host that created the service. In other words, $\alpha(s, t)$ gives the logical location of the service at any given point in time.

Services also possess a motion profile μ, which like the motion profiles of hosts gives the physical location of the service at a given point in time. However, unlike hosts, the motion profile of a service cannot be defined in an arbitrary manner. Rather it must be derived from the allocation profile. The motion profile of the service at a given instant is the same as the motion profile of the host on which it resides at that instant (which is indicated by its allocation profile). If a service moves to another host, it "inherits" the motion profile for that host. We overload the definition of μ to include mobility of services. Formally:

$\mu(s, t) = \mu(\alpha(s, t), t)$

Like hosts, the motion profile of a service along with its allocation profile is considered knowledge that is associated with that particular service and is freely traded in the epidemic manner described earlier.

Service Requests. Service requests are used by clients to specify the kind of services in which they are interested. Traditionally, services were requested according to their capabilities and performance attributes. In our model, due to the exploitation of knowledge, we can support *pre-planning*, i.e., the client can decide a priori the types of services it requires and the *time intervals* during which it requires them. The system then uses all available knowledge to choose the services that are

best suited to the client's needs and are actually within communication range of the client during the interval of the requirement. Each request is assumed to have a unique identifier. Thus we characterize a service request r using the attributes defined as follows:

$\beta(r)$ - the host that made the request
$\chi(r)$ - the capabilities desired
$\pi(r)$ - the performance level desired
$st(r)$ - the starting time for the service requirement
$et(r)$ - the ending time for the service requirement

Observe that while we specify explicitly the time interval for which we require the service, we do not indicate the location at which we require the service. This is because the location value is implicitly specified. For a service request $\langle r, h, c, p, t_1, t_2 \rangle$, the locations at which the service is desired are given by the motion profile of the client host h over the time interval $[t_1, t_2]$. In other words, we can obtain the locations by evaluating the motion profile of the requesting host for all points in the time interval specified explicitly in the request.

Having defined hosts, services and service requests, we now turn our attention to how services are selected.

3.3. *Defining the Satisfying Set*

Recall that in traditional SOC systems, service selection was done according to capabilities and performance attributes. In addition to matching capabilities and performance attributes, our system must also take into consideration the spatiotemporal characteristics of the service. In this subsection, we formally describe how services are chosen.

Basic Satisfiability. When looking for a service s which can *satisfy* a service request r, the first thing that needs to be considered is whether that service is actually what the client is looking for. We call this the *basic satisfiability requirement.* For a service requirement r and a service s, we formally define a basic satisfiability relation σ as:

$$\sigma(r, s) \equiv \chi(r) \leq \chi(s) \wedge \pi(r) \leq \pi(s)$$

The " $\leq$" operator is overloaded in the formula. In the case of capabilities, the " $\leq$" operator indicates that the public interface of service s completely covers the set of operations specified in the service requirement r. In the case of the attributes, " $\leq$" indicates that the service s has all the attributes specified in the service requirement r and for every such attribute, the value of the attribute in the service subsumes that in the requirement. The determination of whether one value is subsumed by another is determined by the attribute type itself which is assumed to have a built-in comparator. Observe that the notion of basic satisfiability is identical to the satisfiability requirements in traditional SOC systems. However, this is where limitations

of the current state of the art stops becomes apparent. A service that has all the capabilities and performance is useless if it is not within communication range of the client at the time of the requirement. Hence, we must take into consideration the spatiotemporal characteristics of the service in addition to its capabilities.

Reachability. In addition to meeting the basic satisfiablity requirements, a service must be *reachable* for the entire duration of the service requirement. A service is considered reachable if its allocation profile $\alpha(s)$ evaluates to some host h that is within communication range of the client host h_c for the time interval of the service requirement $[t_1, t_2]$. Formally, we define reachability ρ as

$$\rho(h_c, s, t_1, t_2) \equiv \langle \forall t : t_1 \leq t \leq t_2 :: |\mu(h_c, t) - \mu(\alpha(s, t), t)| \leq \Delta \rangle$$

where Δ is the communication range.

For convenience, we overload the definition of ρ to encompass the reachability between two hosts. It is simply defined as:

$$\rho(h_c, h, t_1, t_2) \equiv \langle \forall t : t_1 \leq t \leq t_2 :: |\mu(h_c, t) - \mu(h, t)| \leq \Delta \rangle$$

Observe that the definition of reachability ρ depends on having access to the motion profiles of hosts and the allocation profiles of services in question. We remind the reader that these profiles constitute knowledge and are disseminated freely among hosts in the MANET via a gossipping protocol. This knowledge is stored in a local knowledge base on each host. The motion profiles stored in the knowledge base are used to determine whether a service will be allocated to a host within range at the time of the request, thereby meeting the reachability requirement.

Under the assumption that services do not exhibit logical mobility, a service s is said to satisfy a service requirement r made by a client on host h if the following conditions are met:

1) $\sigma(r, s)$ - basic satisfiability
2) $\rho(h, s, t_1, t_2)$ - reachability
3) $\langle \forall d : d \in \delta(s) :: \rho(\alpha(s, t), d, t_1, t_2) \rangle$ - dependencies
   all w.r.t. $K(h)$

Condition 1 above states that the service should meet the basic satisfiability requirements, i.e., capabilities and attributes, and that the service should be reachable for the entire duration of the requirement. Further, all dependencies of the service should also be reachable from the host on which the primary service is executing. Note that we use $K(h)$ to denote the knowledge base on the host idenified by h. Since the formulas depend on motion and allocation profiles, they have to be evaluated with respect to some knowledge base. Since planning normally takes place at the point of origin of the request , i.e., host h, the satisfiability is relative to its knowledge base, i.e., $K(h)$.

Logical Mobility. Thus far in our presentation, we made the assumption that services were not logically mobile. Here, we remove this assumption and allow services to migrate from host to host. The logical mobility of services adds a degree

of freedom. We can now migrate services from a host that is not in range of the client to one that is in range. We consider a scenario where we can move an idle service from the host on which it is currently resident (which ostensibly would not be in communication range at the time of the requirement) to a host that will be in communication range at the appropriate time. For this, we introduce a function called Γ defined as follows:

$$\Gamma(h_1, h_2, t) = \langle \exists t_s, t_e : t_s < t_e \leq t :: \rho(h_1, h_2, t_s, t_e) \rangle$$

The predicate Γ indicates whether there is a time interval before a deadline time t, during which hosts h_1 and h_2 are in communication range (this interval can then be used to logically move the service between the two hosts). Note that for this case, once the service has moved to this host, the service is resident on the host at least until the end time of the client's request. Thus, under these assumptions, a service s satisfies a requirement r made by host h by relying on host h_t if the following conditions are satisfied:

1) $\sigma(r, s)$ - basic satisfiability
2) $\rho(h, h_t, st(r), et(r))$ - reachability of some host h_t
3) $\texttt{migrate}(\alpha(s, t), h_t, et(r))$ - migrate service to host h_t
4) $\langle \forall d \in \delta(s) \exists h_d :: \rho(h_d, h_t, st(r), et(r)) \wedge \texttt{migrate}(\alpha(d, t), h_t, st(r)) \rangle$ - reachability to some host h_d and migrate dependencies to host h_d w.r.t. $K(h)$

where

$$\texttt{migrate}(h_1, h_2, t) = \Gamma(h_1, h_2, t) \vee \langle \exists h, t' : h \in H \wedge t' < t :: \texttt{migrate}(h_1, h, t') \wedge \texttt{migrate}(h, h_2, t) \rangle$$

When we allow logical mobility, the service must still meet the basic satisfiablity requirement. However, the reachability requirement is not a strict one. In combination with the `migrate` operation, the reachability requirement can be stated as: there should be a suitable host h_t that is reachable, and the service should be able to migrate to this host before the start time $st(r)$ of the request and remain there for the duration of the request.

To summarize, we are moving a service that is otherwise suitable in terms of capabilities and performance but is not resident on a reachable host to a host that is reachable by the client before the client actually needs the service (proactive logical movement). Naturally, any dependencies the service requires must also be moved in a manner similar to the service itself.

Thus far, we have formally defined the conditions that result in a service satisfying a request. We reiterate that the consideration of the spatiotemporal aspects of a service and the proactive selection of services is only possible due to the knowledge that is gathered by each host. In the next section, we describe the software architecture and the implementation of the knowledge management system which gathers and stores such knowledge.

4. Software Architecture and Implementation

In this section, we present the software architecture and implementation details of our knowledge management system. We begin with a brief overview of SPAWN, which is our SOC middleware designed for MANETs within which we have implemented knowledge management. We then present the basics of the knowledge management system and explain its role within the context of the SPAWN system. We follow this with a more detailed discussion of each of the components of the knowledge management system. We conclude the section with several code examples that show how an end user might access the system to request services in a knowledge managed architecture.

4.1. *SPAWN Overview*

SPAWN is a proxy-based SOC middleware for MANETs. It is based on the Jini model and is written entirely in Java. To adapt the Jini model to MANETs, several changes were required. The centralized service directory of Jini was replaced with local service directories on each host. When a host needs to advertise a service, it places an advertisement (and the appropriate proxy) in its local service directory. Hosts that are within communication range logically merge their local service directories to form a transiently shared, federated service directory. In this way, the advertisements in the local service directories become accessible to other hosts in the MANET. The leasing mechanism of Jini was discarded because it is no longer essential.

When a host moves away, it unshares its local service directory. Hence, the service advertisements belonging to that host are no longer available in the federated directory, which is consistent with the fact that the services offered by that host are also not available (due to the host not being in communication range anymore). The Java RMI mechanism for invoking services was replaced with a tuple-space-based communication mechanism (described later) due to it being more resilient against disruptions. Finally, the centralized code repository for proxy code was replaced with a transiently shared, federated repository, much like the service directory itself.

In addition to the modifications we made to the Jini approach, we added two new features. The first is an automated upgrade system where the services and their proxies can be upgraded while they are running with very little interruption in communication. The second is a mobile thread system for Java where a service can be migrated from host to host to stay in range of the client. More details on these enhancements can be found in [6].

The dynamism of MANETs also precluded the direct use of traditional socket streams as communication channels between service providers and clients. Instead, SPAWN uses a tuple space abstraction to wrap sockets resulting in a generative style of communication [3]. The service directory and code repository are implemented in terms of tuple spaces which are automatically federated among hosts in proximity. 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 using the `out(tuple)` operation, making it available to all other agents that are sharing the same tuple space. To read a tuple from the tuple space using the `in(template)` operation, an agent needs to provide a template, which is a pattern describing the tuple that the agent is interested in. A template is a sequence of fields, each of which can contain a formal (wildcard) 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. Service advertisements are implemented as tuples that contain a description of the service's capabilities while service requests are implemented as templates.

The `ServiceDirectory` class is actually a wrapper class that owns a generic tuple space. Together, this class and tuple space comprise a service directory. The `ServiceDirectory` class provides standard SOC operations such as advertise, request, and invoke. A service is advertised by placing a tuple in the tuple space owned by the `ServiceDirectory` class using the `out` operation. A request is implemented as a `rd` operation, with the interface of the desired service being passed as the template. Invocation is done through a targeted remote `out` operation where the tuple is stamped to indicate its destination host.

To provide asynchronous interactions, SPAWN offers a reaction mechanism. An agent can declare interest in a tuple by registering a reaction on a tuple space using a remote operation parametrized by an appropriate template and by providing a callback function to be called when a matching tuple becomes available. All reactions in SPAWN are *weak* reactions, meaning that once the condition for the reaction becomes true, the callback function is guaranteed to be called eventually, and not necessarily within a single atomic step.

4.2. *The Knowledge Management System*

The Knowledge Management System described in this paper has been implemented as an extension to the SPAWN system. The knowledge management system fits between the API layer and the communication layer of SPAWN. This required the extensions to SPAWN shown in Figure 2.

The Knowledge Management System is responsible for handling the exchange of knowledge among hosts in the MANET and managing the knowledge base on each host so that the information it contains may be obtained easily by interested applications. More precisely, the Knowledge Management System performs the following functions: 1) aggregation of knowledge about other hosts in the MANET, 2) dissemination of knowledge about the local host to other hosts, and 3) management of the local knowledge base.

The knowledge management system is represented by a singleton `KnowledgeManager` that runs on each host in the MANET. Like the rest of SPAWN, the `KnowledgeManager` has been implemented in Java. On startup, the knowledge manager starts a SPAWN agent which we call the *knowledge agent*.

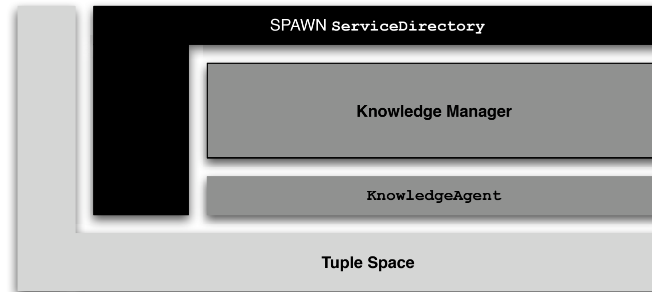

Fig. 2. The architecture of SPAWN with the knowledge management infrastructure added.

This agent is the interface between the knowledge manager and the tuple space that is owned by the `ServiceDirectory` class (the agent is necessary due to a design feature of SPAWN which allows only SPAWN agents to access tuple spaces).

The SPAWN `ServiceDirectory` class now owns a `KnowledgeManager`, which encapsulates all knowledge management functions, in addition to a tuple space. All service advertisements and requests are now directed to the `KnowledgeManager` instead of being placed directly in the tuple space (service invocations and other communication are still placed directly in the tuple space). The `KnowledgeManager` has access to the tuple space owned by the `ServiceDirectory` (through the `KnowledgeAgent`) for communication related to the exchange of knowledge about services. To discover services, the `ServiceDirectory` now calls the `findService(...)` method on the `KnowledgeManager` which returns an appropriate service if one is available. Note that the `KnowledgeManager` has access to the tuple space from which it retrieves all service advertisements, but it only makes services that meet spatiotemporal requirements available to the `ServiceDirectory` class. Thus, the `ServiceDirectory` contains only those services with an acceptable level of connectivity.

The `KnowledgeManager` has three subcomponents: 1) a `KnowledgeDisseminator` that distributes knowledge about its parent host and services running on it to other hosts in the MANET, 2) a `KnowledgeAggregator` that gathers knowledge about other hosts and services in the MANET, and 3) a `KnowledgeBase` that stores this gathered knowledge. The complete structure of the knowledge management system is shown in Figure 3. Before we present the details of these three components, we describe the representation of knowledge that is used for these components.

4.3. *Knowledge Representation*

The question of how knowledge is represented and codified is crucial as it determines how it can be organized and the flexibility with which it can be exchanged. In our

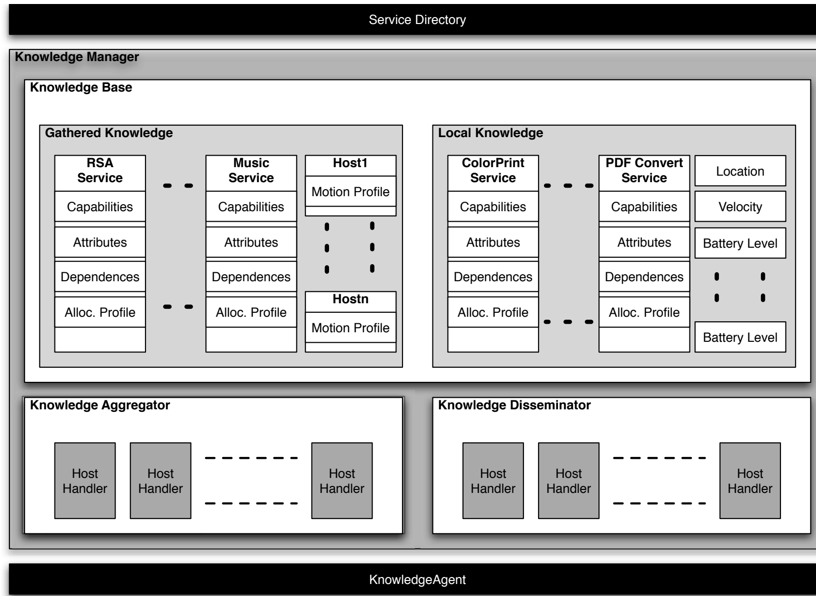

Fig. 3. The structure of the Knowledge Management System

system, each piece of knowledge (excluding identifiers) is referred to as a *parameter*. For example, a host h by our definition has one parameter, its motion profile μ. A service s has multiple parameters, such as its capabilities χ, performance attributes, π, and so on. Every parameter is associated with an identifier which identifies the host or service with which the parameter is associated. Parameters are also stamped with the time of their last update. While our system can support an arbitrary number of parameters, we restrict ourselves to those described in Section 3.

Since the parameters represent knowledge and are intended to be communicated from host to host, we chose to encapsulate parameters within tuples, just like all other communications in our system. We defined a special class of tuples called "knowledge tuples" that carry only knowledge. Knowledge tuples are distinguished from other tuples by virtue of the fact that its first field contains the reserved string "Knowledge". The general form of the knowledge tuple is:

<"Knowledge", String:owner , ParamType:type, Value:value>

Thus, for example, a knowledge tuple containing the allocation profile of some service "printService" would look like:

<"Knowledge", "printService" , AllocationProfile.class, profileValue>
where profileValue is an object of type AllocationProfile

4.3.1. *The Knowledge Base*

The Knowledge Base on each host is divided into two sections: gathered knowledge and local knowledge. Gathered knowledge is knowledge associated with other hosts and services in the MANET. Local knowledge is knowledge associated with the local host. The local knowledge section is organized as a `Vector` of *entries*. Entries can be only of type *service entry* since the only host being tracked in this section of the knowledge base is the local host whose parameters are stored separately. Each entry has within it a vector of parameters and their values, e.g., the host entry will have a parameter called "motion profile" with a value that is a function definition. The local knowledge portion offers the following basic API for access and updates:

```
Object:value getLocalParamValue(Entry:entry, String:paramName)
setLocalParamValue(Entry:entry, String:paramName, Object:newValue)
```

The gathered knowledge portion of the knowledge base is different from the local knowledge portion in that more than one host entry may be present in the gathered knowledge section, each host entry representing a host in the MANET of which the local host is aware. The API of the gathered knowledge portion is:

```
Object:value getGatheredParamValue(Entry:entry, String:paramName)
setGatheredParamValue(Entry:entry, String:paramName, Object:newValue)
```

The second method is only accessible to the Knowledge Aggregator (described later) to ensure that local applications do not tamper with the gathered knowledge. Both parts of the knowledge base share a common API to create new entries and parameters:

```
createHostEntry(String:hostName)
createServiceEntry(String:serviceName)
createHostParam(Entry:hostEntry, String:paramName, Object:paramValue)
createServiceParam(Entry:serviceEntry, String:paramName, Object:value)
```

Entries and parameters in the knowledge base are created on an on-demand basis. A host entry for some host h is created only when some knowledge pertaining to that host is received for the first time. Similarly, parameters are created as they are needed. This allows for hosts to register interest in only some parameters of a host or service, which is not essential in our limited scope but can be useful if a large amount of knowledge is traded and hence is an accomodation for extensibility.

Finally, the knowledge base supports several convenience methods:

```
Service:service findService(ServiceRequirement:requirement)
Vector:services getAllKnownServices()
Vector:motionProfiles getMPOfServicesWith(Capabilities:cap,
    Attributes:attrib)
MotionProfile:mProfile getLocalMP()
```

4.3.2. *The Knowledge Aggregator*

The knowledge aggregator is responsible for aggregating knowledge from other hosts in the MANET. As we stated in previous sections, a gossipping protocol is used to exchange knowledge. Simply put, when a host encounters another host, they exchange all the knowledge in their respective knowledge bases. Given our knowledge manager design, it is also possible for hosts to exchange more restricted sets of parameters about certain hosts. However, this is outside the scope of this paper. We assume here that all knowledge is freely exchanged.

The complete process from the creation of the knowledge manager to the aggregation of knowlege is shown in Figure 4. Numbers in parentheses in the following text refer to steps shown in the figure. To aggregate knowledge, the knowledge manager registers reactions on the tuple space(7) (via the knowledge agent (6) since no other class can perform tuple space operations). The reactions can be registered for each type of knowledge that the knowledge manager wants to aggregate or for all types of knowledge. For example, to aggregate any type of knowledge, the knowledge manager registers a reaction with a template of the form <"Knowledge", String.class, String.class, Object.class,>. The first field indicates that we are interested in knowledge tuples while the second field is a wildcard indicating that we are interested in any host that provides knowledge. The third field is a wildcard indicating that we are interested in all types of knowledge, while the fourth is a class that is a wildcard for the actual value of the knowledge parameter. Wildcards are used since we want any knowledge associated with any host in the MANET.

When a host comes within communication range of the reference host, SPAWN raises an event indicating that a new host has been detected (12). This event causes the reaction registered (for knowledge) to fire. When the reaction fires, indicating that new knowledge of the type we are interested in has been placed in the tuple space, a `HostHandler` is created to retrieve a copy of the knowledge tuple from the tuple space (13). Note that a *copy* of the knowledge tuple is retrieved so that other hosts in the network may also read the same knowledge. It is the responsibility of the host that disseminates the knowledge to remove any old tuples using the `in` operation before outing a tuple with updated knowledge. Once the tuple has been retrieved (14), the second field is examined. If the identifier in that field is not present in the knowledge base, then a new entry is created in the knowledge base using `createHostEntry(...)` (15) or `createServiceEntry(...)` (17) as appropriate. If the entry exists, the next thing that is checked is whether the parameter type (indicated by the third field) is present under the host or service entry. If not, the parameter type is created using `createHostParam(...)` (16) or `createServiceParam` (18) as appropriate. If the parameter type already exists, the timestamp of the value in the fourth field of the tuple is checked against the value in the knowledge base. If the newly retrieved value is more recent the knowledge base is updated using the `setGatheredParamValue(...)` method (19).

One concern is that the knowledge base may grow forever. We address this

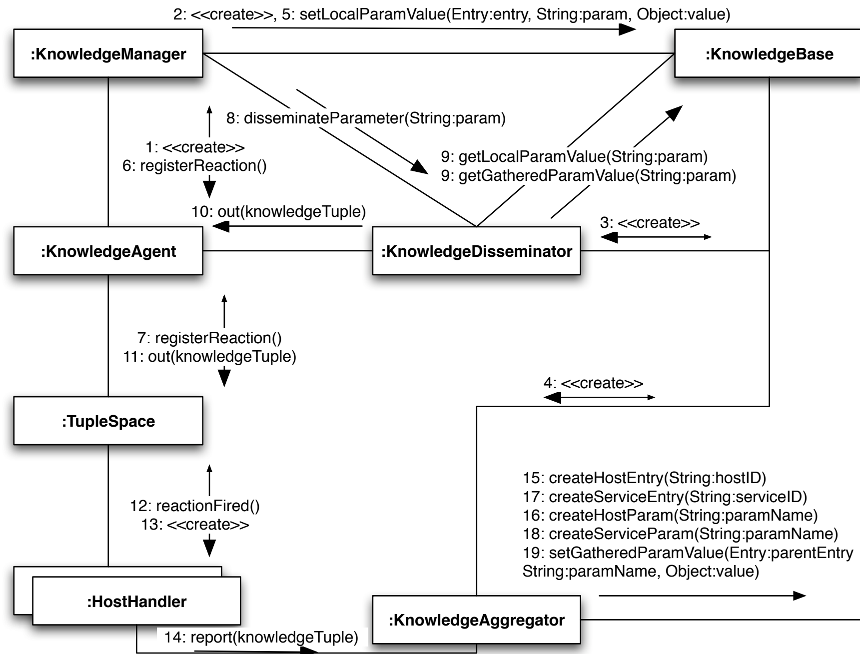

Fig. 4. The interaction of the various components of the knowledge management system

concern in two ways: 1) we examine the motion profiles of the hosts and services and maintain the entry only for the duration that the motion profile is known; 2) if the MANET is a closed system, the amount of knowledge has an upper bound, to which every knowledge base would converge, thus preventing unbounded growth.

4.3.3. *The Knowledge Disseminator*

The knowledge disseminator is responsible for disseminating knowledge that is in a host's local knowledge base. This knowledge can be knowledge about the local host or knowledge about other hosts that has been aggregated by the local host in the past. The `RequestHandler` within the disseminator listens for requests from aggregators on other hosts. When such a request is received (8), it obtains the appropriate contents from the knowledge base and packages it. Each parameter in the knowledge base is packaged into a knowledge tuple of the form

<"Knowledge", String:owner , ParamType:type, Value:value>

Access to the knowledge is possible via the `getLocalParamValue` method (9) for local knowledge and `getGatheredParamValue` method (9) for aggregated knowledge as described earlier. Like the aggregator, the disseminator supports the dissemination of a subset of all the information it has, if so desired. This is useful when two

hosts have a similar amount of knowledge in their knowledge bases and need send only small updates rather than the complete contents of their knowledge base. Once the knowledge tuples are ready, they are passed to the `Transmitter` which instructs the knowledge agent to place the knowledge tuples into the federated tuple space (10).

4.4. *Anatomy of a Service Request*

Having described the structure and implementation of the Knowledge Manager, we conclude this section with a description of the process that occures when a client wants to discover a service. The client makes a request to the `ServiceDirectory` class, which has been updated to take parameters for the time duration for which the service is required. Thus, for a client to request a service, it needs only three lines of code:

```
//Get handle to local directory singleton
ServiceDirectory dir = ServiceDirectory.getLocalDirectory();
RequestID rid = dir.requestService(new ServiceRequest(
    capabilities, attrib, startTime, endTime));
//At time startTime:
ServiceProxy proxy = dir.getProxy(rid);
```

The client can make the request at any time and gets a request ID which identifies that particular request. A qualifying service is found, and its proxy stored locally. At the time the client actually needs the service, it queries the `ServiceDirectory` with the request ID and obtains the proxy which it can then use as needed.

While the client API is simple, complex operations are going on under the API layer. Since this paper addresses concerns at the middleware level, we now present the code that executes within the `ServiceDirectory` when a service request is received at the API layer. The `ServiceDirectory` calls `findService(req)` on the `KnowledgeBase` class. That method executes the following code.

```
ServiceProxy proxy = null;
boolean[] qualifies = true;
Pair[] mProfiles = KnowledgeBase.getMPOfServicesWith(
    req.getCapabilities(), req.getAttributes());
for(int i = 0, i < mProfiles.length), i++){
    for(int j = req.getStartTime(), j < req.getEndTime(), j++){
        if(!(inRange(mProfiles[i].getMP().evaluate(j),
            KnowledgeBase.getLocalMP().evaluate(j))) {
            qualifies[i] = false;
        }
    }
}
```

```
for(int i = 0; i < qualifies.length; i++){
    if(qualifies[i]){
        proxy = KnowledgeBase.getGatheredParamValue(
            mProfiles[i].getServiceEntry(), "Proxy")
        return proxy;
    }
}
return null;
```

The second line of code gets the motion profiles of all services that meet the capabilities and attributes requirements (but which may not meet the spatiotemporal requirements). Details of how this method works is shown below.

```
Vector qualifyingServices = new Vector();
ServiceEntry[] services = KnowledgeBase.getAllKnownServices();
for(int i = 0; i < services.size(); i++){
    if(KnowledgeBase.getGatheredParameter(services[i],
        ‘‘Capabilities’’).matches(req.getCapabilities()) &&
         KnowledgeBase.getGatheredParameter(services[i],
        ‘‘Attributes’’).matches(req.getAttributes())) {
        qualifyingServices.add(KnowledgeBase.getGatheredParameter(
            services[i], ‘‘MotionProfile’’));
    }
}
```

The matches method for capabilities returns true if the interface offered by the service proxy is a subclass of the interface specified in the service requirement. For attributes, it checks to see that the service proxy has all the attributes specified in the requirements. If this condition is satisfied, it checks each attribute individually. Every attribute in the implementation is actually a three-tuple [attribute, value, comparator]. The comparator indicates whether a greater value or a lesser value of the attribute is desired. If the attributes offered by the service compared using the comparator are greater than those in the requirement, the `matches(...)` method returns `true`.

Once the motion profiles are obtained, we check whether the motion profiles of the local host and the service evaluate to locations that are in communication range of each other. We repeat this procedure for the duration of the service request. The first qualifying service that meets the spatiotemporal test is returned to the `ServiceDirectory`.

On the back end, the knowledge manager watches the SPAWN tuple space for knowledge tuples using the reaction method described previously. When such tuples are located, the information in them is added to the `KnowledgeBase` using the `createHostEntry(...)`, `createServiceEntry(...)`, `createHostParam(...)`,

`createServiceParam(...)`, and `setGatheredParamValue(...)` methods.

5. Evaluation

Having described the formal model, architecture, and implementation details of our knowledge management system, we now present results of simulation experiments we conducted to evaluate our approach. In our simulations, we focused on the case where hosts are mobile but services do not move between hosts, as this case is perhaps the most practical given considerations of security, copyright, and licensing. We first describe our experimental setup, and then present results of several experiments, which demonstrate the validity of our approach.

5.1. *Experimental Setup*

To simulate the environment of a knowledge-driven MANET, we developed a custom simulator in which hosts move within a well defined space according to predetermined motion profiles and can offer or request services. The details of our experimental setup are now presented:

Simulation Setup. The simulation space is a grid of 250 X 250 squares. A host occupies a single square, though a single square may accomodate more than one host. The number of hosts used in our experiments ranges from 25 to 250. Each host possesses a knowledge base, motion profile, service requirements profile and a service offerings profile. Details on how these were generated are given below. All simulations were run for 500 time points. At each time point, all hosts moved to the next location as given by their motion profiles. They then exchanged knowledge with neighbors within their communication radius (which ranged between 2 and 20 grid squares). Once the knowledge exchange phase was completed, the hosts tried to satisfy their requirements. Finally, it should be noted that each data point we show from our experiments represent an average value over 20 different data sets collected in repeated runs.

Mobility Model. We ran our experiments using two mobility models: (1) Random Walk and (2) Random Waypoint. In random walk, we generated a starting point at random. For each subsequent entry in the motion profile, we randomly selected one of four directions (UP, DOWN, LEFT, RIGHT) with equal probability and moved to that grid square. In random waypoint, we selected a random start point and a random waypoint. We then moved with constant velocity from the start point to the first waypoint. Once the waypoint was reached, another waypoint was randomly generated and the process was repeated. In both models, hosts could move only one grid square in a single iteration. All motion profiles were generated prior to the actual experiment as hosts were required to know their motion plan a priori (to be able to give it out as knowledge). As may be expected, the random walk profiles tended to keep the host within a smaller region of the simulation space while random waypoint tended to provide better coverage. We also defined a

variable MAX_FUT, which restricted the amount of time into the future for which a motion profile was valid.

Service Model. Each host in our simulation had a service offerings profile and a service requirements profile. The service offerings profile for a host was generated by selecting a random number of services from the master set of six services (printer, pdfconvert, information, mp3convert, rsaencrypt, imagecrop). The service requirements profile is generated by selecting a random number of requirements for services that are not in the service offerings profile of the host. This results in (1) the host not being able to satisfy requests locally, thereby biasing the statistics and (2) the number of requirements being inversely proportional to a hosts capabilities. All requirements in the system have a fixed length as defined by the global variable REQ_LEN. A host in considered *satisfied* if all its requirements are met.

5.2. *Experiment I: Varying the Communication Radius*

The communication radius of a mobile device determines the extent of its reach and affects the number of hosts with which it can exchange knowledge. In this experiment, we show how the communication radius of hosts affects (1) the percentage of hosts that have their requirements satisfied and (2) the average size of the knowledge base on each host as a percentage of the size of the global knowledge base. These results are shown in Figure 5.

As may be expected, we see an increase in both the percentage of hosts that have their service requirements satisfied, as well as the average size of the knowledge base, with an increase in communication range. The expanded communication range fosters more interactions thereby expanding the knowledge base which in turn increases the chances of finding suitable services. One may question the fact that even with a high level of knowledge, a relatively low percentage of hosts requirements were satisfied. There are several reasons for this low number. (1) We only count the hosts that have *all* their service requirements fulfilled as having been satisfied. Thus, there were a number of hosts that had most of their requirements satisfied but were not included in the count because they were not completely satisified. (2) In some cases, the host acquired knowledge of a required service after the time of requirement had passed. Thus, even though the knowledge was acquired, it did not help identify a service. (3) In many cases, even though the knowledge about a host was acquired, it was not suitably located to meet the timing constraints of the service requirements, i.e., the fraction of *exploitable* knowledge was low. It is exploitable knowledge that explains the difference in numbers between the random walk (WALK) and random waypoint (WAYPT) models. Since WAYPT has better coverage of the entire space, a host meets a lot of other hosts and gathers a lot of knowledge but those hosts are seldom on hand when a service is required. In WALK, the host stays within a small region and meets fewer hosts and thereby has a smaller knowledge base, albeit one containing a higher fraction of exploitable

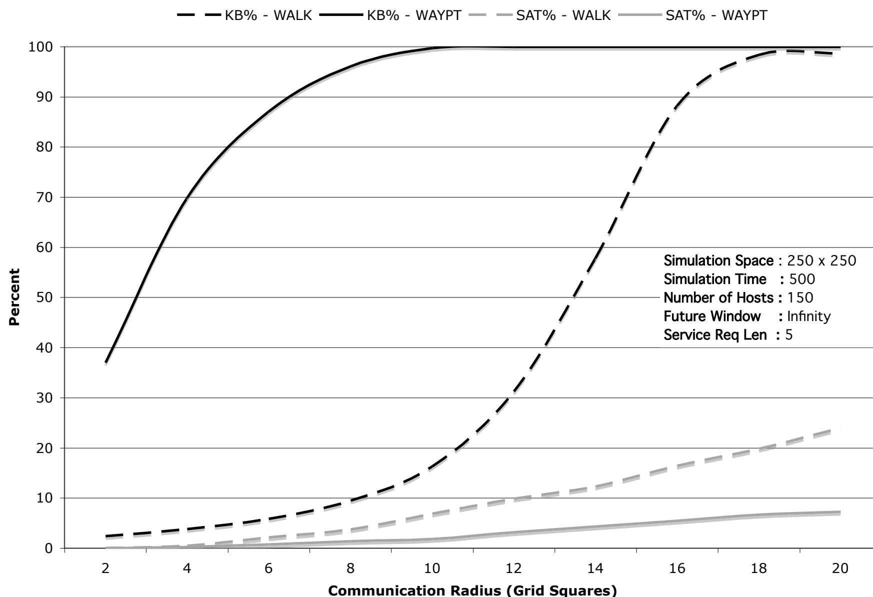

Fig. 5. Effect of communication range on knowledge base size and percentage of hosts satisfied

knowledge, i.e., the hosts in the knowledge base are those nearby and which remain in close proximity due to their limited mobility. Thus, in WALK, there is a greater correlation between knowledge base size and exploitable knowledge, i.e., knowledge about hosts that can satisfy requirements.

5.3. *Experiment II: Varying the Number of Hosts.*

As we mentioned for the previous experiment, the number of interactions a host has with other hosts affects the percentage of hosts that are satisfied and the average size of the knowledge base on each host. In this experiment, we examine how host density affects these parameters, the rationale being that a greater host density will, for a fixed communication range, increase the potential number of interactions. The results are shown in Figure 6.

Increasing the host density increases the percentage of hosts satisfied, though the rate of increase is not as marked as when the communication range is increased. In fact, for WALK, the average knowledge base size actually decreased initially. The explanation for this is simple. Adding hosts is not effective unless these additional hosts interact with other hosts frequently. In the case where the average knowledge base size decreased, the additional hosts were isolated from the rest, and hence their knowledge bases were empty resulting in the average being brought down. Note that this problem was not seen with the WAYPT model which tends to foster more interactions and seldom isolates hosts in a specific region.

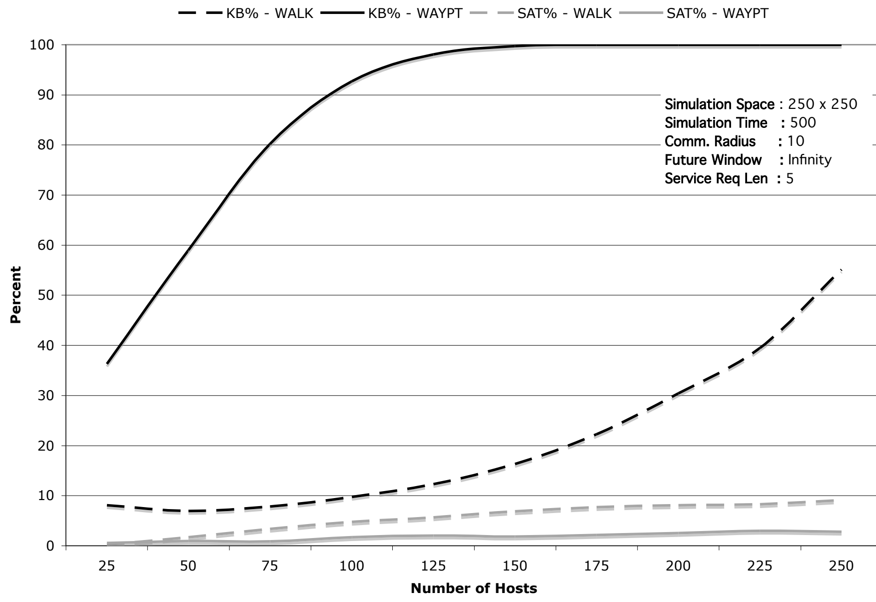

Fig. 6. Effect of host density on knowledge base size and percentage of hosts satisfied

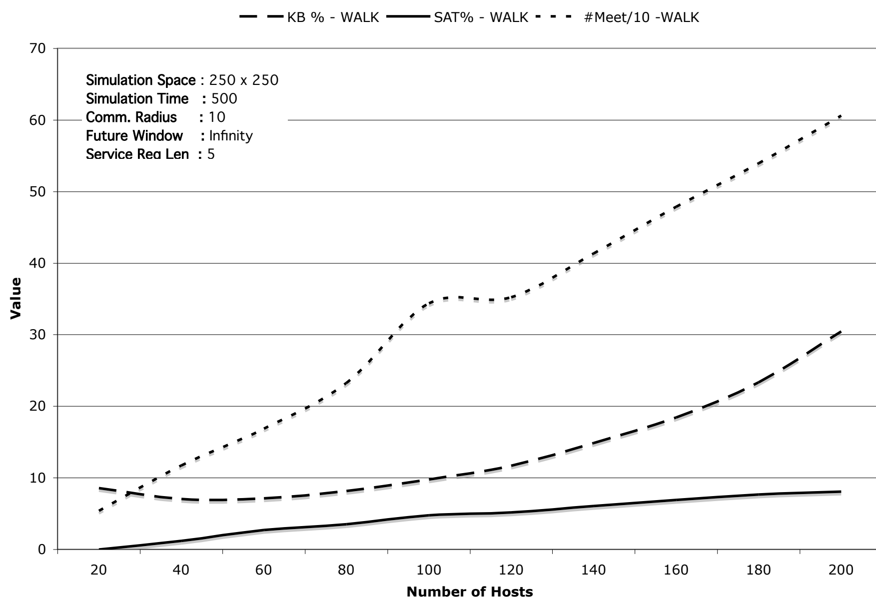

Fig. 7. Number of meetings, knowledge base size, and satisfied hosts (random walk model)

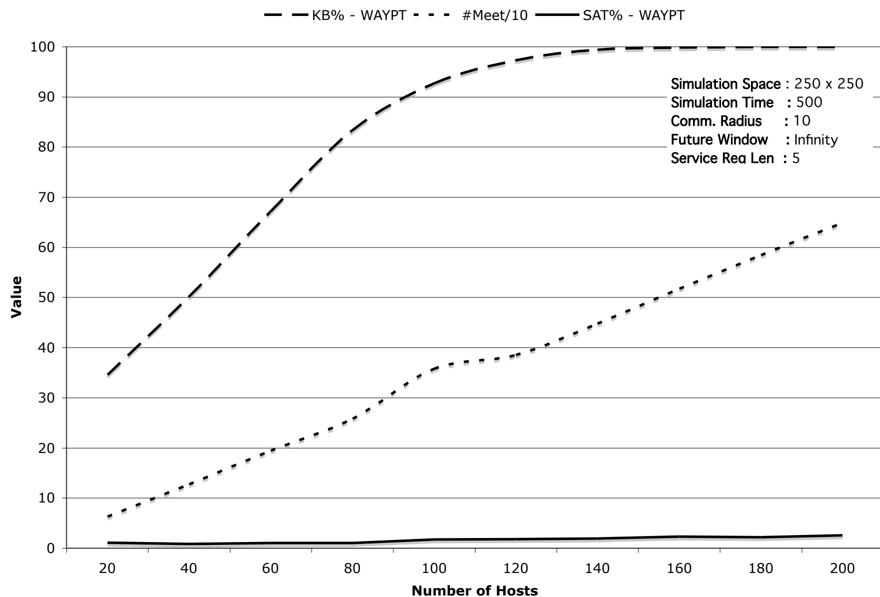

Fig. 8. Number of meetings, knowledge base size, and satisfied hosts (random waypoint model)

Manipulating the communication range and the number of hosts serves to increase or decrease the number of meetings between hosts. The number of meetings is crucial since a higher number of meetings theoretically translates to more opportunities for knowledge exchange and service usage. In our experiments, we found that an increase in the number of meetings improves the percentage of hosts satisfied for both mobility models. However, the size of the knowledge bases were much smaller in WALK for a similar number of meetings. This was due to the localized nature of WALK which caused multiple meetings but seldom with unique hosts. The results are shown in Figures 7 and 8.

5.4. *Experiment III: Varying the Length of Requirements*

Another factor that affects the percentage of hosts that are satisfied is the length of the service requirement interval, with longer duration requirements having a lower chance of being satisfied because they require the client and the service provider to be within range and to have very similar mobility patterns over the requirement interval. Our experiments (Figure 9) confirmed this expectation. However, it is interesting to note that even when the request length was longer than the maximum time a host could be in communication range, a small percentage of requirements were still being satisfied, which would be impossible without forward looking knowledge.

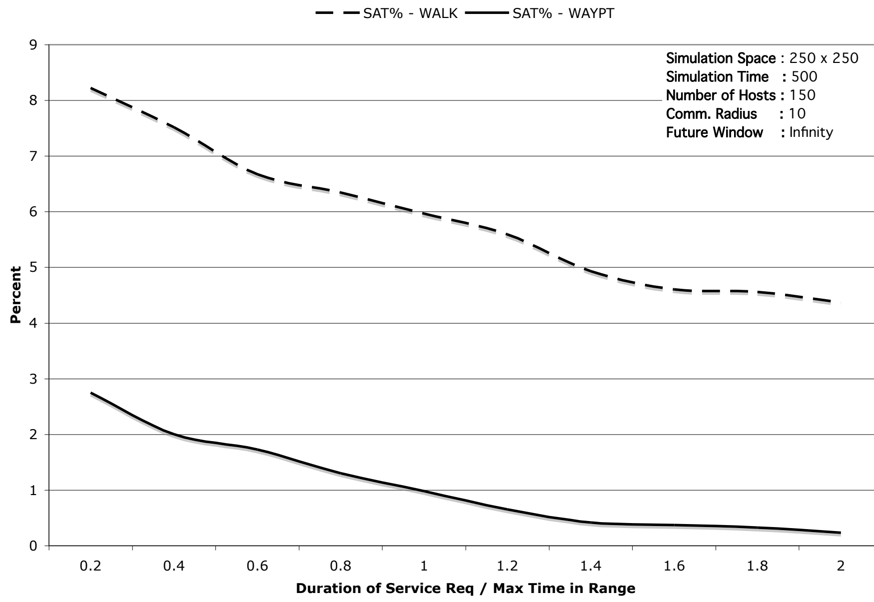

Fig. 9. Effect of requirement length on percentage of hosts being satisfied

5.5. *Experiment IV: Varying the Forward Looking Window Size*

Our final experiment examined the relationship between the amount of time into the future for which a valid motion profile is available and the percentage of hosts that are satisfied. If a larger forward looking window is available, then the chances of finding a suitable service are greater than if the window is smaller. However, in our experiments (see Figure 10) we found that even with a window size that is 10% of the optimum window size, approximately 50% of the hosts that would be satisfied with the unlimited window still end up being satisfied. This is an important result since it means that hosts need to predict their motion for only a short time into the future, which is useful if the host "changes its mind" frequently or does not know its long term mobility pattern.

5.6. *Some Remarks*

It should be noted that all the experiments shown above were conducted in a knowledge managed environment. When the knowledge management feature was turned off, the percentage of satisfied hosts fell to zero. We admit that it will not always be the case that the satisfied hosts percentage will be zero as a non-knowledge-managed system depends purely on chance to satisfy requests. The point we wish to emphasize is that with knowledge management we can do better.

The results presented were obtained using randomly generated data sets. While the experiments show general trends, we acknowledge that our system will not be

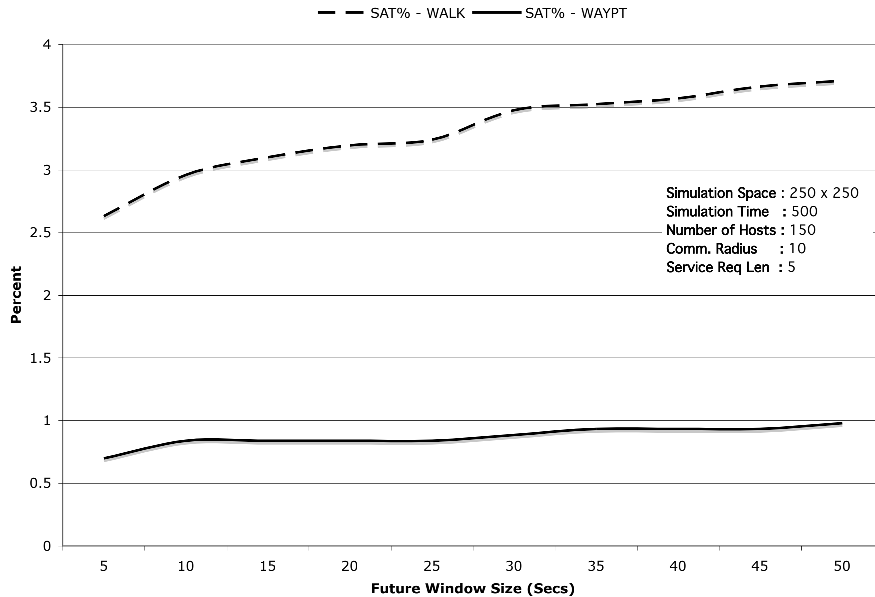

Fig. 10. Effect of future window size on percentage of hosts being satisfied

effective in particular scenarios where the requirements are in conflict with mobility patterns, e.g., an isolated host can never hope to satisfy its requirement profile, nor can a host that requests services before it has met another host. Our aim with this paper is to show that for reasonable patterns of mobility and requests, our approach can bring a greater degree of predictability.

Given our results, one might question whether the overhead of knowledge management is worth the relatively small gains in satifaction percentage. A study of the actual overhead of our system was not within the scope of this paper. In our future work, we plan to analyze the overhead of our system in the context of several real-world mobile ad hoc computing applications.

Our final remark relates to those hosts that had part of their requirements satisfied. We see a possibility for hosts that have most of their requirements satisfied to be fully satisfied with the help of logically mobile services which can relocate themselves with the aim of satisfying hosts that are close to being fully satisfied. We plan to analyze this as part of our future work.

6. Conclusion

Using knowledge about other hosts in a MANET to plan interactions between clients and service instances can yield benefits in terms of predictability and stability of applications that expand their capability via the opportunistic use of external services. In this paper, we have described a formal model for proactive service selection in a

knowledge managed MANET. We have shown that such an architecture is desirable in the dynamic environment of a MANET. Details of a proof-of-concept implementation and evaluation results obtained through simulation are also presented. This is a first step towards introducing the new concept of knowledge managed MANETs, which we believe can play a role in achieving close to wired network-like predictability in MANETs where application-service interactions have a well defined structure. However, many complex issues pertaining to the efficiency and effectiveness of such a system are still outstanding, and we plan to address these in future work.

Acknowledgements. 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

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services-Concepts, Architectures and Applications*. Springer, 2004.
2. F. M. Costa and G. S. Blair. The role of meta-information management in reflective middleware. In *Proceedings of ECOOP'2000 Workshop on Reflection and Meta-level Architectures*, June 2000.
3. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.
4. R. Handorean, R. Sen, G. Hackmann, and G.-C. Roman. Automated code management for service oriented computing in ad hoc networks. Technical Report WU-CSE-2004-17, Washington University Department of Computer Science, 2004.
5. Radu Handorean and Gruia-Catalin Roman. Secure service provision in ad hoc networks. In *Proceedings of The First International Conference on Service Oriented Computing (ICSOC 03)*, number 2910 in Lecture Notes in Computer Science, pages 367–383, 2003.
6. Radu Handorean, Gruia-Catalin Roman, Rohan Sen, Gregory Hackmann, and Christopher Gillpher Gil. Spawn: Service provision in ad-hoc wireless networks. Technical report, Department of Computer Science, Washington University, 2005.
7. T. Harvey and K. Decker. Planning ahead to provide scheduler choice. In *Proceedings of Autonomous Agents Infrastructure Workshop*, May 2001.
8. G. Karumanchi, S. Muralidharan, and R. Prakash. Information dissemination in partitionable mobile ad hoc networks. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 4–14. IEEE Computer Society, 1999.
9. James Kempf and Pete St. Pierre. *Service Location Protocol For Enterpriser Networks: Implementing and Deploying a Dynamic Service Resource Finder*. John Wiley and Sons, 1999.
10. Sun Microsystems. http://java.sun.com/products/ejb/c-plain-text, December 2005.
11. A. Montz, D. Mosberger, S. O'Malley, L. Peterson, T. Proebsting, and J. Hartman. Scout: A communications-oriented operating system. Technical report, Department of Computer Science, University of Arizona, 1994.
12. M. Paolucci, T. Kawmura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *Proceedings of the 1st International Semantic Web Conference*, 2002.

13. Gruia-Catalin Roman, Radu Handorean, and Rohan Sen. Tuple space coordination across space & time. Technical report, Department of Computer Science, Washington University, 2005.
14. Salutation Consortium. Salutation web page. http://www.salutation.org.
15. Rohan Sen, Radu Handorean, Gregory Hackmann, and Gruia-Catalin Roman. An Architecture Supporting Run-Time Upgrade of Proxy-Based Services in Ad Hoc Networks. In *Proceedings of the International Conference on Pervasive Computing and Communications PCC-04*, pages 689–696, 2004.
16. R. Simmons. Towards reliable autonomous agents. In *Lessons Learned from Implemented Software Architectures for Physical Agents*, pages 196–203, 1995.
17. Jim Waldo. The Jini Architecture for Network-Centric Computing. *Communications of the ACM*, 42(7):76–82, 1999.