Mobile Service Overlays: Reconfigurable Middleware for MANETs*

Balasubramanian Seshasayee
College Of Computing
Georgia Institute of Technology
Atlanta, GA - 30332
bala@cc.gatech.edu

Karsten Schwan
College Of Computing
Georgia Institute of Technology
Atlanta, GA - 30332
schwan@cc.gatech.edu

ABSTRACT

Distributed applications running on Mobile Adhoc NETworks (MANETs) can benefit from underlying middleware that provides services for self-management. Such services can address the dynamic conditions associated with a MANET's operational environment and changes in end user needs. This paper describes Mobile Service Overlays (MSOs), an overlay network-based decentralized middleware that provides basic support for online management, shown useful for services like online reconfiguration for managing energy consumption and failure resilience. Decentralization is achieved by partitioning the application's overlay network into smaller units termed *chains*, and implementing decentralized reconfigurations involving specific chains, triggered by monitoring events. The paper also presents the overheads of these services in a lightweight, non-Java implementation of MSOs targeted at an example MANET application in cooperative robotics.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Applications

General Terms

Design, Performance, Experimentation

Keywords

Mobile computing, Reconfigurable middleware

1. INTRODUCTION

Distributed applications running on Mobile Adhoc NETworks (MANETs) are being used in a wide range of domains, from autonomous robotics to emergency management. With increasingly powerful hardware becoming available even for handheld devices, the combined computational power of systems built with MANETs can be large. This promotes cooperative approaches to running applications across sets of participating machines. Examples include robots collaboratively undertaking a search and rescue mission [15], coordinated actions of geographically dispersed agents in an emergency rescue operation, distributed gaming, and offloading an application to surrounding entities in a ubiquitous computing environment [5].

Driven by applications in autonomous robotics and in vehicular control, our work specifically considers the energy constraints inherent to mobile systems. Here, cooperative approaches to running distributed applications can be used to share the loads implied by energy-intensive applications across multiple devices and/or by using remote systems without energy constraints, such as a wall-powered server system temporarily available to the application. In all such cases, the computations being performed must be distributed across participating nodes so as to suitably utilize their current computational and energy resources. In realistic applications, however, the ability to offload and move computations will be constrained. For example, consider multiple autonomous robots collaborating to identify a human survivor at an emergency site, where one robot is forced by the environment to separate from the rest of the group. In this case, it cannot offload any of its local sensing and locomotion-based computation. Another scenario is one in which said robot has found the survivor and is now critical to maintaining contact. Then, the related basic sensing and data transmission tasks cannot be offloaded, and higher-level tasks in the sensor pipeline can be carried out only by mobile sites reachable by the data-generating robot. Computation offloading, therefore, must be governed by both resource issues like connectivity, computational, and energy constraints and by semantic information about applications and their needs. Such information includes current application needs, the latter stated by SLOs (Service Level Objectives), an example being required end-to-end delays for sensor data interpretation and for responses to sensed information.

This paper presents a formulation of the management problem for MANET-based systems in which applications are represented by their dependence graphs linking different application services, and the execution environment is represented by a resource graph capturing connectivity, computational, and energy resources. Resource graphs can change since nodes may join/leave, fail, or become disconnected.

*This work is funded in part by the NSF-ITR award and Intel corporation

MobiShare'06, September 25, 2006, Los Angeles, California, USA.

For example, if a robot performing processing on behalf of another moves out of proximity, an alternative site needs to be identified. Computational graphs can change in response to new application needs. For instance, when a survivor is identified, the robot who sought the survivor now remains stationary to act as a sensor device.

The dynamic nature of the MANET systems and applications considered in our research implies that the mapping problems formulated for prior work in static systems [13] have become dynamic mapping and reconfiguration problems. This requires solutions that are efficient in terms of their implied overheads and effective in terms of their ability to continually improve the Quality of Service metrics applied to distributed MANET applications. The Mobile Service Overlay (MSO) middleware described in this paper makes the following contributions to addressing the dynamic resource management problems faced by MANET-based systems:

1. MSOs implement low-overhead computational overlays across cooperating distributed MANET devices;
2. MSOs provide efficient and scalable runtime abstractions, termed *chains*, for describing and then managing the overlays that run the computation graphs used by distributed applications;
3. MSOs provide efficient base support for online monitoring to assess the current resources used by and accessible to chains; and
4. MSOs permit developers to deploy alternative management policies based on different or multiple application-level metrics and SLOs.

In the remainder of this paper, we first describe in more detail the runtime management problem faced by MANET applications. This leads to a more precise description of the *chain* abstraction advocated in our work. The architecture of MSO is described in Sec. 3, follwed by MSO's response to dynamics in Sec. 3.5, then experimental evaluations and discussions of related work in Sec. 4 and 5. Conclusions and future work are described in Sec. 6.

2. MSO OVERLAY MANAGEMENT

MSO overlays are constructed and configured by the management layer of the MSO middleware. Our earlier work has already demonstrated that it is important to be able to efficiently map and re-map overlays to underlying MANET resources [4, 3]. Extending such work, the design goals for MSO overlay management are the following:

- Low overhead, low latency reconfiguration: in a dynamic environment, the middleware, on detecting a change, must quickly arrive at an updated mapping, since the time available before the next change might be limited. In keeping with prior work on 'missed opportunities' in the real-time domain [22], this implies the need for efficient heuristic solution methods. Optimality is not our goal, because accurate system-wide resource information and precise knowledge about current application needs are not typically available for MANET systems.

- Fault resilience: since cooperative solutions imply that participating nodes can arrive or leave dynamically, with similar effects caused by application mobility, MSO management must be resilient to change and failures. In contrast to full application-level fault tolerance, however, our goal is to have middleware stay "online" after faults occur. Applications can then realize their own fault tolerance methods on that basis.

- Scalability through decentralization: centralized solutions will not scale in MANET environments. While respecting the dependence graphs of applications, the *chain* abstractions used by MSO middleware is the basis for localized configuration heuristics.

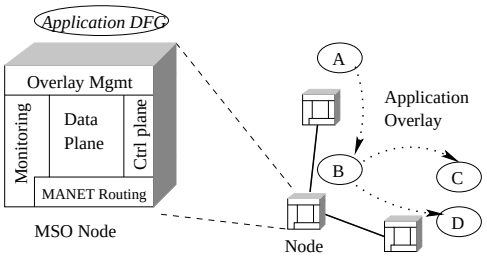

Figure 1: Overview of MSO's Architecture

In keeping with these goals, MSO middleware both (1) implements completely decentralized reconfiguration solutions and (2) enables remapping at various levels of granularity, ranging from a pair of nodes to the entire network. The MSO implementation consists of a set of services distributed over the network, so that identical instances run at each of the nodes. Each instance of MSO consists of a monitoring component, a decision-making component, and mechanisms for reconfiguration. Reconfiguration rules are used in the decision making process to decide which events obtained from the monitoring component can trigger the different levels of reconfigurations. In summary, MSO middleware support for runtime overlay management has the following functionality: (1) monitoring node resources, including remaining node energy, node movement and failures, and network connectivity. (2) Reconfiguration mechanisms based on the computational *chain* abstraction, where each computational graph representing an application is divided into multiple chains that jointly realize this graph. The purpose of chains is to capture desired end-to-end behaviors and node-node connectivity. Chains also represent a smaller unit to use for runtime reconfiguration, thus reducing the graph mapping problem described earlier to the simpler chain mapping problem. (3) Actual reconfiguration actions are triggered by monitoring events and driven by reconfiguration rules formulated with the goal of optimizing specific objective functions (such as minimizing energy usage or end to end latency etc.).

3. MSO ARCHITECTURE AND MANAGEMENT METHODS

The MSO middleware's design and implementation do not create additional dependencies across cooperating nodes. That is, each MSO node provides to the application an identical set of middleware services. The reason, of course, is that for MANET systems to attain failure resilience, each device must be able to operate independently of other devices.

3.1 Programming Model

MSO implements an event-based model of data exchange, as commonly used in pervasive applications like robotics

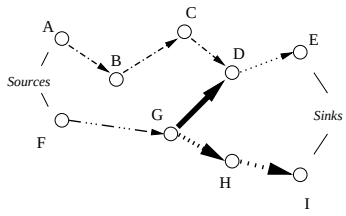

Figure 2: Partitioning the graph into chains

and distributed simulation, as well as in enterprise computing [11] for complex event processing and in real-time systems [18]. In the event-based model, the application is represented as a directed flow graph, with each vertex signifying some processing and acting as a data source and/or sink. Data exchanges are represented as directed edges. Associated with each edge is a format that describes the data it transmits. MSO programs, therefore, consist of code modules running in vertices mapped to overlay nodes, receiving formatted inputs from and generating outputs to other vertices.

3.2 Chain Formation

While the event-based model is known to provide a clean separation between different application services and their interactions, a task necessary for MSO middleware is to partition its event-based interaction graph into its constituent and independently manageable computational *chains*. Formally, a chain is a maximal set of sequential vertices and edges of the flow graph, with a single entry and a single exit. Events enter the first vertex of the chain, the *head*, sequentially pass through and are operated on at each node in the chain, and finally exit at the last node, the *tail*. Since an application's QoS can be formulated to depend on the QoS of each chain (e.g., consider end-to-end delay), each chain can be managed independently in order to guarantee such QoS properties (though not all QoS properties are composable in this manner [8].) In this fashion, chains compartmentalize management to be confined to more "local" portions of the application. The effects of compartmentalization are discussed further in Sec. 3.4. Chains have also previously been used to enable computational offloading [10].

The algorithm used for chain formation is straightforward. Intuitively, it selects some vertex and then creates a chain starting at this vertex, considering its successors and predecessors in the application's data flow graph (DFG). An example is shown in Fig. 2, where the DFG with source vertices A, F and sink vertices E and I is partitioned into the list of chains (A-B-C-D, D-E, F-G, G-D, G-H-I). These chains have the property that there is one entry and one exit. A chain may have both a source and a sink, or only either of them, or even none at all. More importantly, the chain construction algorithm is of complexity $O(|V| + |E|)$ and can be run by each node in the overlay.

3.3 Deployment

Application deployment involves mapping chains onto the underlying physical network, by assigning a node to each vertex in each chain, then constructing the overlay network based on this mapping, and finally, commencing data movement in the constructed overlay. Since more than one node participates in this effort, this is a distributed procedure initiated by the node to which the head of the chain is assigned. Deployment itself is subject to dependencies, where each chain is not deployed until all of the chains leading to it have been deployed. Hence, the deployment process is only partially parallel, dependent upon the average number of outputs of DFG vertices.

Algorithm 1 Deployment

1: If a chain has no dependences and the head of the chain is assigned to this node, then start the assignment by exploring the route taken by a packet to the sink node furthest away, collecting the capacities of all nodes along the route.
2: Allocate nodes to vertices of the chain in proportion to the fraction of the overall costs contributed by the vertices. Perform assignments of nodes to vertices.
3: Activate the node corresponding to the next chain by decreasing its dependence count and assigning the node to the head of its chain.
4: Repeat steps 1-3 until all chains have been assigned to nodes.
5: Each node that has a chain head mapped to it constructs the actual overlay network along its chain.

In realistic systems, source and sink vertices can often be assigned only to certain nodes (e.g., those that possess sensors/actuators that produce or consume data). We therefore, assume the source and sink vertices to be preassigned to particular nodes. If this is not the case, a distributed process can be used by which nodes possessing the capabilities required by the source/sink vertices are chosen according to their abilities and assigned to nodes before deployment.

As indicated in Algorithm 1, route exploration is a key step of the deployment process. MSO uses probing for this purpose, by sending a probe packet to a sink node furthest away, and obtaining the capacities of each node along the route taken by the probe packet. These capacities constitute metrics like processing capability, battery lifetimes, etc. To better illustrate the route exploration procedure, consider a simple example, where a flow graph consisting of three chains A-B-C-D, E-D, and D-F is mapped onto a network with four nodes, with the connectivity between them represented as N1-N2-N3-N4. Suppose that A is preassigned to N1 and F to N4. Now, while assigning the first chain, the route taken is obtained (N1-N2-N3-N4), and each node's capacities are queried in the process. Then, chain A-B-C-D is assigned a fraction of nodes N_{abcd} from among N1,N2,N3,N4 such that $\frac{cost(N_{abcd})}{\Sigma(Nx)} = \frac{cost(A-B-C-D)}{cost(A-B-C-D)+cost(D-F)}$. Now assume that this leads N_{abcd} to be N1-N2-N3. Then, B, C, and D are mapped among N1-N2-N3 (A is already mapped to N1). Note that deployment is linear on the number of vertices and network nodes, and uses a greedy method to assign vertices to nodes. While it is therefore not optimal, the MSO middleware continuously seeks to improve local optimality metrics through its intra-chain remapping procedures. These are discussed in the next section.

3.4 Reconfiguration

Reconfiguration involves remapping portions of the overlay network, to create a different assignment between vertices in the overlay and underlying machines. The primary goal of MSO is efficient reconfiguration, and for this purpose, it provides capabilities to perform remapping at various granularities, as detailed below. Several assumptions

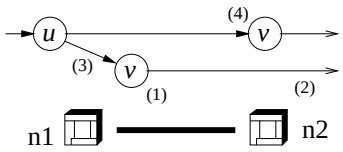

Figure 3: Upstream Relocation Example

made by the current mapping and remapping methods will be reviewed in our future work: (1) the flow graph is a directed acyclic graph, and it remains static throughout the lifetime of the application, (2) node discovery, naming, and message routing are performed at a lower layer independent of the middleware, and (3) all faults are fail-stop

3.4.1 Intra-chain Remapping

Intra-chain remapping is used continuously throughout a chain's existence. It is triggered by monitoring events reporting changes in local resource availability and/or in application requirements or resource usage. An intra-chain remapping constitutes either an upstream or a downstream relocation of a vertex. For example, consider two vertices u and v, assigned to nodes $n1$ and $n2$ respectively, with a directed edge in the overlay from u to v. Further, assume that u and v belong to the same chain. Now, an upstream relocation "moves" v from $n2$ to $n1$, i.e., to the same node as u. The steps involved in such a relocation, as shown in Fig. 3, involve (1) creating a new v vertex at $n1$ and (2) connecting it to the same destination as the previous v, followed by (3) linking u's output to the new v, and then (4) freeing up the old v. Similarly, a downstream relocation "moves" the successor vertex to the next node downstream in the data flow. Note that these remapping strategies affect only the nodes in the vicinity of the change (i.e., local nodes), and other nodes in the network are neither involved in nor aware of this remapping. This process has low overhead and can be run fairly frequently.

Intra-chain remapping is useful in obtaining a local optimum. That is, based on the metric we seek to optimize, each node housing a vertex evaluates the cost of relocating the successor vertex upstream or downstream, against making no changes. The costs of relocating state associated with the vertices are to be included in estimating total costs. Each vertex can independently (but sequentially) perform this check, but the end vertices of a chain should not participate in this remapping. Otherwise changes in its location will affect the preceding or succeeding chains, possibly leading to an undesirable cascade effect.

3.4.2 Chain Remapping

In contrast to intra-chain remapping involving only two nodes, chain remapping affects all of the nodes to which a chain is mapped. The "head" and "tail" of the chain remain unchanged. The operation is performed in three stages. First, the chains chosen for remapping are reassigned to the nodes after a route exploration, ignoring the existing mapping. Next, the edges that lead to the older instances of the chains are rerouted to the just-assigned ones. Finally, the overlay network corresponding to the older instances of the chains is freed. This procedure is illustrated in Fig. 4, where the shorter route from A to C achieved due to node mobility results in remapping the chain along the new route, with the result that vertex 3 is now mapped to node C, from node B.

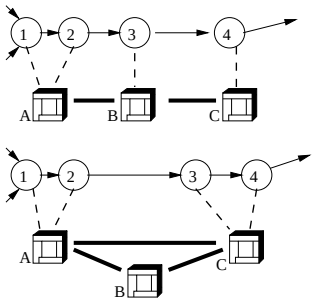

Figure 4: Chain Remapping Example

Chain remapping is useful in the event of a change in route between the nodes housing the head and tail of the chain. Under these circumstances, only the affected chain needs to be remapped, leaving other chains unchanged.

3.4.3 Global Remapping

Global remapping is carried out to effect changes that involve more than a single chain, and it involves performing the mapping operations described in Sec. 3.3, over a set of the chains. Synchronization between the nodes during the remapping process is enforced by the dependence relationship between chains, and it is achieved by remapping a chain only after all chains depending on have been remapped. Note that the procedure is identical to the mapping methods described earlier, except that in general, only a subset of the chains are remapped. Global remapping is useful when node failures or movements affect the head/tail of a chain, thus affecting multiple chains. It can also be used to remap all the chains, i.e., entire flowgraph.

3.5 Response to Dynamics

MANETs undergo constant changes in communication topology, connectivity, and node characteristics. Depending on the severity of the change, a remapping can be performed at the appropriate level of granularity. The decision as to what type of remapping is to be carried out, and when, is taken based both on continuous monitoring of the environment and on the formulated management rules.

Each MSO node runs a separate monitoring thread that is used to maintain metrics like a measure of the amount of computation carried out, the amount of data transferred from/to the node and the expected lifetime of the battery. In addition, monitoring also periodically checks for the liveness of its neighbors (as determined by the routing layer) and for changes in the routing layer. The metrics monitored are available for use by the higher layers for remapping decisions. This part of the subsystem requires access to the routing layer to detect any changes, but does not depend on the type of routing protocol used. Monitored metrics are also available for sharing, as each monitor exposes its data to other nodes through SOAP calls. Currently per-node metrics are monitored, but are being modified to gather application specific metrics as well. Management rules describe the action that needs to be taken in response to monitored events. Experimental work is ongoing in rules such as include intra-chain remapping on a local load imbalance, chain remapping in response to a route change within a chain, and global remapping on more severe events.

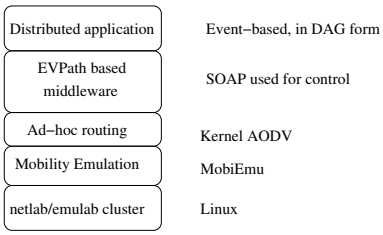

Figure 5: Experimental Framework

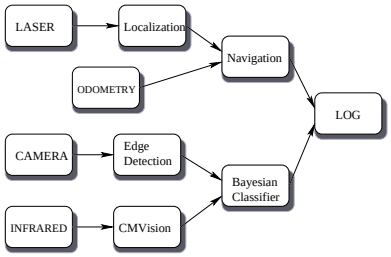

Figure 6: DFG of Robotics Application

4. EVALUATION

MSO has been implemented in C/C++ and tested on Intel Xeon nodes (with frequency scaled down to 350MHz to better match the capabilities of current mobile systems) running Linux-2.4.20, in the Netlab experimental testbed [19]. Netlab uses the Utah Emulab software, but for these experiments, we also use MobiEmu [23], based on software MAC filtering, to emulate mobility of participating nodes. Kernel AODV [16] is used as the routing protocol. The middleware itself uses an overlay network creation and management library called EVPath. EVPath allows the creation of overlays from lightweight vertices called "stones", and it can use arbitrary network links to connect such stones. It also uses an efficient binary format for data exchange. Remote management actions are implemented with the gSOAP SOAP toolkit, enabling any node in an MSO to configure any other node. The overall evaluation framework is illustrated in Fig. 5. MobiEmu can be used both in wired or wireless environments, as it is independent of the physical layer. For the experiments described in this paper, we use wired 100Mbps ethernet due to the resulting ease of conducting experiments, but MSO requires no changes to work with wireless ethernet and few changes to work with other wireless communication layers.

Experimental Results

We conduct simple experiments to measure the costs involved in remapping operations described in Sec. 3.4. For evaluation purposes, we construct a robotics application to mimic a scenario with four robots proceeding in a convoy, in a search-and-rescue mission. The application consists of a navigation pipeline, where laser and odometry data from sensors are used to locate the current position of the robots in a map, and to plan a path through it. It also consists of a target recognition pipeline, where images from visible light and infrared cameras are processed and Bayesian inference used to estimate the chances of finding a target in an area. The results from both these pipelines are then stored for later analysis. Of the four robots that cooperatively run this application, the leader robot is assumed to possess the sensors, and the trailing robot, the storage. Clearly, running all of the above tasks in the leader robot due to the proximity of processing to data, and communicating the results to the trailing robot, will drain the former's batteries quickly. Hence, MSO can be applied to offload the computations to the intermediate robots.

We model this scenario using four nodes lying in a straight line, with each node connected only to its physical neighbors. The application makes use of the localization and navigation modules from CMU's CARMEN robotics software suite, and CMVision for image analysis. Other parts of the application are developed in-house. The data for laser and odometry sensors were taken from a sample simulation run of CARMEN. The data sizes from these sensors are 967 bytes and 38 bytes, respectively, and are sent at the rate of 5 events/sec. The images used in place of the cameras are scaled down to roughly 3KB in size and sent at 0.5 events/sec. The resulting flow graph is shown in Fig. 6. All the computational vertices of the flow graph are assigned the same cost, and the capacities of the nodes are allocated such that each node in the network is assigned atleast one vertex.

Table 1: Overheads for the Robotics Application

Operation	Cost (ms)	σ(ms)
Mapping	1,390.0	77.7
Upstream relocation	94.3	6.4
Downstream relocation	56.3	3.3
Chain remapping	2,759.1	217.0
Global remapping (part)	2,943.1	265.8
Global remapping (whole)	3,327.2	335.2

The overheads of mapping and remapping the flowgraph onto the four node network are measured over several iterations (Table 1). The cost of mapping the flowgraph is around 1.4 seconds. A large portion of this cost is due to route exploration (Sec. 3.3) by the nodes along the linear path. However, since data transfer has not commenced, the nodes are lightly loaded and overhead is low. The intra-chain remapping primitives (upstream and downstream relocations), performed on the laser sensor to navigation chain, incur very low overheads, permitting their frequent use. The primary cost here arises from the SOAP calls made in the MSO control plane. Since upstream relocation requires an extra SOAP call than a downstream one, its costs are higher. Next, we show the costs of performing remapping at various granularities – remapping the laser-navigation chain alone, the entire navigation pipeline, and finally, the entire flowgraph, respectively (Fig. 6). These costs are significantly higher, arising due to the data volume passing through the nodes. The high variations in their values (σ in the table) is also due to this. This causes a slowdown in the SOAP-based control messages, which form a dominant portion of the remapping processes. Indeed, when run at 2.8GHz, these costs went down by as much as 10x. To overcome this effect, a more efficient cross-platform RPC protocol could replace SOAP. Other optimizations like queueing data during remapping, asynchronous remapping, etc. are being studied.

5. RELATED WORK

Peer-to-peer systems (P2P) and data sharing in MANETs face similar research challenges, as both are designed to work in dynamic, unreliable environments. While much of the research on P2P has focused on scalability, the emphasis of

MANETs has been on load balancing and energy conservation [1]. Recent work has attempted to apply P2P principles to MANET systems. In *ORION* [17], an overlay network is constructed and maintained by the application layer, along with protocols for search and transfer of files, as well as the routing itself. As overlay networks embody the data sharing needs of the application and can be managed by the application itself, they have found use in both P2P systems over the Internet [7], to build resilient overlays in the event of poor wide area routing, and in MANETs ([14] explores this in detail). In addition to data sharing, however, recently, there has been increased interest in using overlays to run distributed applications. Implementations of such systems include *Expeerience* [12], a JXTA-based middleware that has been adapted to run on MANETs, *SOLAR* [2], a publish-subscribe based middleware that allows injecting processing modules to specific nodes for distributed processing, and our prior work on the JECho Java-based publish-subscribe middleware [3]. An earlier version of the same for event-based systems, *opportunistic channels* [4], is realized by JECho (predecessor of the EVPath middleware used here). In *MagnetOS* [9], a monolithic Java application is partitioned into its constituent classes and cooperatively run in a MANET to conserve energy.

Self configuration, an important issue in MANETs running distributed applications, is a key aspect of MSO. [6], underscores the need for high-level rules in concert with monitored context to enable autonomous behavior. However, in contrast to their approach, where a set of nodes act as "management bodies" to direct reconfiguration, MSO maintains no such separate control infrastructures, and any node detecting a change can trigger a reconfiguration. [21] identifies the design paradigms for self-configuring networks to be (i) maintaining local behavior to achieve global properties, (ii) enabling simple coordination, (iii) minimizing long-lived state and (iv) adapting to changes. Our approach attempts to address these requirements by (i) providing local remapping, (ii) making simplifying assumptions during remapping and conservatively resorting to global remapping if these are violated, (iii) maintaining state locally, and (iv) monitoring for and reacting to changes.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we describe Mobile Service Overlays, a decentralized middleware for running distributed applications on MANETs. MSO provides services for low overhead and low delay reconfiguration by partitioning the application into smaller entities called chains, and enabling redeployment of the chains onto underlying nodes, at multiple granularities. This is supported by monitoring that looks for and detects changes in the underlying network and triggers the appropriate reconfiguration services in response. We describe an implementation of MSO and indicate their usefulness in practical MANETs, based on a sample application.

Since the development on MSO is ongoing, features like state transfer during remapping, load balancing, support for heterogeneity and fault tolerance through chain replication, are in progress. Our future work involves developing an integrated framework for energy efficiency, load balancing, and autonomic self-management under failures, to be applied to a distributed simulation infrastructure for traffic management via cooperating vehicles. We are also currently upgrading our evaluation framework from one of wireless network emulation on wired links to a fully wireless testbed [20].

7. REFERENCES

[1] G. Anastasi, M. Conti, and M. Kumar. Energy management in mobile and pervasive computing systems. In *HICSS*, 2005.

[2] G. Chen and D. Kotz. Solar: An open platform for context-aware mobile applications. In *Intl. Conf. on Pervasive Computing (short paper)*, 2002.

[3] Y. Chen and K. Schwan. Opportunistic overlays: Efficient content delivery in mobile ad hoc networks. In *Middleware*, 2005.

[4] Y. Chen et al. Opportunistic channels: Mobility-aware event delivery. In *Middleware*, 2003.

[5] A. Messer et al. Towards a distributed platform for resource-constrained devices. In *ICDCS*, 2002.

[6] A. Malatras et al. Self-configuring and optimizing mobile ad hoc networks. In *ICAC*, 2005.

[7] D. G. Andersen et al. Resilient overlay networks. In *SOSP*, 2001.

[8] G. Swint et al. Event-based QoS for a Distributed Continual Query System. In *IRI*, 2005.

[9] H. Liu et al. Design and implementation of a single system image operating system for ad hoc networks. In *Mobisys*, 2005.

[10] K. J. OHara et al. Autopower: Toward energy-aware software systems for distributed mobile robots. In *ICRA*, 2006.

[11] K. Schwan et al. Autoflow: Autonomic information flows for critical information systems. *Autonomic Computing: Concepts, Infrastructure, and Applications*, 2006.

[12] M. Bisignano et al. Expeerience: A jxta middleware for mobile ad-hoc networks. In *Peer-to-Peer Computing*, 2003.

[13] T. Braun et al. A comparison study of static mapping heuristics for a class ofmeta-tasks on heterogeneous computing systems. In *HCW*, 1999.

[14] Y. C. Hu, S. M. Das, and H. Pucha. Peer-to-peer overlay abstractions in manets. *Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless and Peer-to-Peer Networks*, 2005.

[15] J. S. Jennings, G. Whelan, and W. F. Evans. Cooperative search and rescue with a team of mobile robots. In *ICAR*, pages 193–200, July 1997.

[16] http://w3.antd.nist.gov/wctg/aodv-kernel/.

[17] A. Klemm, C. Lindemann, and O. P. Waldhorst. A special-purpose peer-to-peer file sharing system for mobile ad hoc networks. In *VTC*, 2003.

[18] D. Luckham and B. Frasca. Complex event processing in distributed systems. Technical report, Stanford University, 1998.

[19] http://www.netlab.cc.gatech.edu.

[20] http://www.orbit-lab.org.

[21] C. Prehofer and C. Bettstetter. Self-organization in communication networks: principles and design paradigms. *IEEE Communications Magazine*, 2005.

[22] D. Rosu and K. Schwan. Faracost: An adaptation cost model aware of pending constraints. In *RTSS*, 1999.

[23] Y. Zhang and W. Li. An integrated environment for testing mobile ad-hoc networks. In *MobiHoc*, 2002.