Agile Computing Middleware Support for Service-oriented Computing over Tactical Networks

Niranjan Suri[1,2], Alessandro Morelli[1,3], Jesse Kovach[2], Laurel Sadler[2], Robert Winkler[2]

[1]Florida Institute for Human & Machine Cognition, Pensacola, FL, USA
[2]U.S. Army Research Laboratory, Adelphi, MD, USA
[3]University of Ferrara, Ferrara, Italy

***Abstract*—Service-oriented architectures (SoAs) are a popular paradigm for enterprise and data center computing but normally do not perform well on tactical networks, which are often degraded in terms of bandwidth, reliability, latency, and connectivity. This paper presents the agile computing middleware and in particular a transparent network proxy and associated protocols that help address the impedance mismatch that occurs between SoAs and tactical and DIL (Disconnected, Intermittent, and Limited) networks.**

Keywords—Tactical Networks, Disconnected, Intermittent, and Limited Networks, Communications Middleware, Transport Protocols, Dissemination Services, Network Proxy

I. Introduction

Service-oriented architectures (SoAs) have evolved into a popular approach that supports rapid integration of multiple software components. SoAs provide well-defined interfaces and protocols to access their capabilities and hence offer many advantages including service reusability, composability using workflows, and rapid configuration and reconfiguration. Traditionally, SoAs have been deployed in enterprise and data center networks that are well connected with few constraints on bandwidth, latency, reliability, and availability. The popularity and success of SoAs in the enterprise environment has argued for their adoption in tactical network environments. However, tactical networks are typically wireless networks with little or no fixed infrastructure and are intermittently connected, bandwidth constrained, unreliable, and exhibit high and variable latencies. Tactical networks, as well as other Disconnected, Intermittent, and Limited (DIL) networks present many problems to the application and deployment of SoAs because SoAs were developed primarily for enterprise networks. SoAs typically use connection-oriented transport protocols such as TCP and encode messages in verbose, bandwidth intensive formats such as SOAP and XML. Additionally, TCP itself does not perform well on Tactical and DIL networks, further impacting the performance of SoAs. A detailed discussion of the network challenges and the requirements for SoAs to function effectively in tactical networks is discussed in [1].

This paper describes components of the agile computing middleware (ACM) that supports SoAs on tactical and DIL networks. The middleware addresses five primary challenges – resource and service discovery, transport protocols, disconnection support, resource allocation and coordination, and finally a transparent network proxy that helps to integrate legacy applications and systems. Given space limitations, this paper will focus primarily on the network proxy (NetProxy), which improves performance of legacy SoAs. Since NetProxy relies on other components in the middleware that provide transport services and disconnection support, those components are also briefly described. The experimental results presented in the evaluation section towards the end of the paper only focus on NetProxy.

II. Middleware Overview

The agile computing middleware (ACM) has been motivated by the challenges posed by tactical and DIL networks and provides a comprehensive set of capabilities including network monitoring, data transport, data dissemination, resource and service discovery, transparent network proxy, and network visualization. Figure 1 shows the key components with a very short label identifying the purpose of each component. The components of this middleware have been primarily developed using C++ and ported to Linux, Win32, and in some cases, the Android environment. Wrappers are available for applications that are written in Java and C#. Furthermore, many of these components are available as open source under the GPLv3 license and currently hosted on GitHub [2]. The following sections describe Mockets, DisService, and the ACM NetProxy in more detail. Other components relevant to SoAs in tactical networks include the Group Manager component, which provides discovery services [3], and AgServe, which realizes a dynamic SoA with service migration [4].

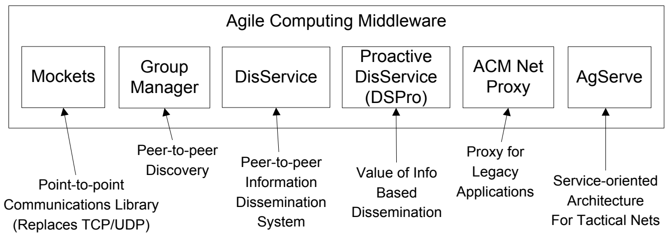
Agile Computing Middleware
Mockets
Group Manager
DisService
Proactive DisService (DSPro)
ACM Net Proxy
AgServe
Point-to-point Communications Library (Replaces TCP/UDP)
Peer-to-peer Discovery
Peer-to-peer Information Dissemination System
Value of Info Based Dissemination
Proxy for Legacy Applications
Service-oriented Architecture For Tactical Nets

Figure 1: Components of the Agile Computing Middleware

III. Mockets

Mockets (for mobile sockets) is a transport protocol designed to replace TCP and UDP and targeted for DIL networks. Mockets itself can operate over UDP for IP networks, and can also operate over non-IP networks using packet adaptors. Mockets replaces the TCP congestion control and reliable transmission algorithms with alternate, custom implementations that are designed for DIL networks. Numerous configurable options allow mockets to be easily adapted to a variety of network links and radios. Unlike TCP, mockets provides a message-oriented interface, which allows Mockets to distinguish between messages. Being able to identify message boundaries and messages allows Mockets to treat individual messages differently, which is not possible with a byte stream oriented model such as TCP. Mockets provides four different classes of service, which allows applications to choose options for reliability and sequencing for each message that they transmit. TCP only provides the equivalent of reliable and sequenced, which is the most expensive choice in terms of bandwidth and latency. Experience has shown that applications rarely need these semantics – but use them because those are the semantics provided by TCP. Many times, applications need sequencing but not reliability (e.g., audio or video streaming) and many times they need reliability but not sequencing.

Other relevant capabilities provided by mockets include prioritization of messages, policy-based enforcement of bandwidth constraints, and message replacement. The last capability allows an application to flush and replace old messages with newer versions. Message replacement is particularly useful when applications generated repeated messages (e.g., status update messages). With TCP, these messages get enqueued when the network link goes down. Since TCP provides no feedback to the application, the application would continue to generate these periodic messages, which continue to get enqueued. When the network link is restored or is available again, all of these old messages would be sent unnecessarily. On the other hand, with Mockets, the application can assign a tag for each type of message, and then request that a new update replace previous messages with the same tag. That results in the most recent message being sent out when a connection is restored, which both reduces bandwidth utilization as well as the latency in message delivery. Experimental results that show the benefits of message replacement are provided in [5].

Another important capability offered by mockets is the ability to dynamically rebind an endpoint of an open connection in a manner that is transparent to the application. For example, if a node switches networks and as a consequence switches IP addresses, mockets could transparently rebind to the new IP address without any interruption of connection from the perspective of the applications. Unlike Mobile IP [6], mockets does not need a home node to forward traffic.

Finally, mockets exports detailed statistics about the status of the network link, including queue sizes, reliability, throughput, and latency. All of these statistics can be utilized by other middleware layers or the application to adjust their behavior based on the performance of the underlying network. This is in contrast with TCP, which traditionally tries to isolate the application from the behavior of the underlying network.

For the purposes of Service-oriented Computing over DIL networks, mockets has been integrated into NetProxy, which allows existing COTS applications to benefit from the performance improvements offered by Mockets. NetProxy is further described in section V.

IV. DisService

DisService is a peer-to-peer disruption tolerant dissemination service that provides many fundamental capabilities for DIL environments. DisService supports store and forward delivery of data and caches data wherever possible in the network, thereby making it disruption tolerant and improving availability of data. The opportunistic listening capability of DisService, described by patent [7], is of particular relevance to vehicular networks, as it addresses challenges such as temporary loss of connectivity due to tunnels and other “urban canyons.” DisService also supports the notion of hierarchical groups to organize the information being disseminated and to be efficient about delivery of information. Subscriptions allow clients to express interest in particular groups. Information is published in the context of a group and is delivered to other nodes where applications have subscribed to those groups. DisService, like mockets, provides a variety of classes of services. The one major difference is that mockets is a point-to-point communication protocol (like TCP) whereas DisService is a point-to-multipoint communication protocol. Therefore, there could be many receivers for messages that are transmitted by one node. With DisService, the class of service is specified by the subscriber (the receiver) for each group. For each group, a subscriber may specify whether sequencing is desired, whether messages should be reliable, and whether missing messages should be requested. Unlike mockets, which uses a Selective Acknowledgement (SAck) mechanism for reliability, DisService uses a Negative Acknowledgement (NAck) mechanism for reliability. Therefore, it is up to each receiver to determine, based on the class of service desired, whether to request for missing pieces (fragments) of a message that has been published (and whether to request complete messages that might have been missed in their entirety).

The disruption tolerance capability of DisService provides a particularly useful foundation for Service-oriented architectures. Each service invocation may be embodied inside a DisService message that is pushed by the client to the node hosting the service to be invoked. No end-to-end connectivity is required in this scenario for the service request message to reach the provider node. When the message is received, middleware on the provider node extracts the service invocation parameters from the message and invokes the service accordingly. When the result is obtained, it is in turn embodied in a new message and pushed by the provider node back to the client node. Figure 2 shows a typical scenario for service invocation via DisService, where the invocation request and the reply may be forwarded through any number of intermediate nodes (also running DisService). Workflows can be realized using an extension of the same mechanism, where intermediate results are embedded into messages and pushed between the nodes using DisService.

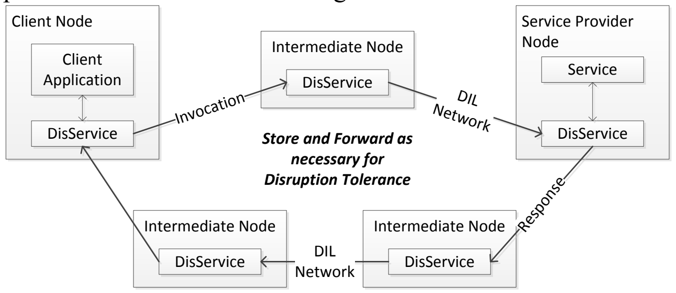

Figure 2: Service Invocation over Disrupted Links with DisService

V. NetProxy

The ACM NetProxy is the component responsible for providing transparent integration between SoA systems and the middleware. Among its many features, the most notable include network protocol remapping, connection multiplexing, data compression, intelligent buffering, flow prioritization, and packets consolidation. The support for protocol remapping in NetProxy plays a particularly important role, as it allows forwarding (part of) the traffic generated by SoA applications over Mockets and DisService transparently. This is a key step towards enabling the reuse of SoA components in DIL networks, because it gives applications access to the features of ACM without making any changes to their source code. In addition, NetProxy supports two operational modes to fit better into different network configurations and to meet various user requirements.

NetProxy works by intercepting packets generated by the applications before they are sent over the network. This gives NetProxy full control of all the different traffic flows going through it, without the need to change any network configuration or any parameter of the proxied nodes/applications. After their interception, NetProxy proceeds by analyzing those packets to extract useful pieces of information (source and/or destination IP address, transport protocol used, type of application/service, etc.). Other ACM components can provide further data, e.g. Mockets statistics can give an insight on the current network status, available bandwidth, and measured latency, which will enrich and complement the information extracted from the intercepted packets. Based on all of this information, on the status of the internal buffers, and on the configuration options specified, the decision making building block of the NetProxy will take the most appropriate actions to satisfy applications' requirements under the constraints imposed by the available network resources. If NetProxy is configured to perform any type of protocol remapping, a second instance on the other end of the communication is necessary to apply the inverse remapping. This ensures complete application transparency, it avoids any compatibility issue, and it removes any dependency from other ACM components. The operational modes chosen for the two instances do not have to match.

Supported operational modes, Host Mode (HM) and Gateway Mode (GM), differ in the way they intercept the traffic and in the role that the NetProxy assumes in the network. When operating in HM, as shown in Figure 3, a copy of the NetProxy is installed on any nodes that run SoA applications in need of proxy support. It follows that multiple instances might be running on different nodes of the network. This mode also requires the installation of a virtual network interface on those nodes, so that applications' packets cannot reach the real network unless they go through the NetProxy. Conversely, when running in GM as shown in Figure 4, the NetProxy assumes it is installed on a single node of the network equipped with two network interfaces (labeled "internal" and "external"), from which it can proxy all the traffic coming in and going out from the network.

The two modes have different advantages and liabilities. HM simply requires the installation of a piece of software on every node that runs applications needing to be proxied; however, each instance of the ACM component will only have a local view of the traffic and of the network conditions. On the other hand, GM requires NetProxy to be installed on a gateway node of the local network, which might not be possible because of the network architecture or authorization issues. However, running NetProxy on a gateway node is preferred in terms of having the best possible view of the traffic generated and the network status.

It is entirely possible to have the equivalent of the NetProxy running on a router or wireless network device of some sort. For example, in a vehicular networking scenario, it is envisioned that the vehicle has a local area network (LAN), which then connects to a wireless router / device that provides the off-vehicle connectivity. The NetProxy in GM would either run between the LAN and the wireless device, or could be directly integrated into the wireless device for complete transparency.

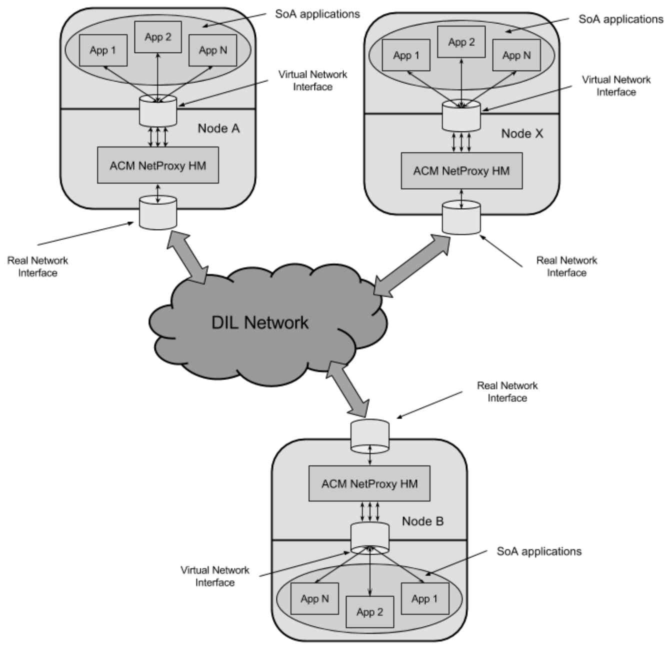

Figure 3: NetProxy Running in Host Mode

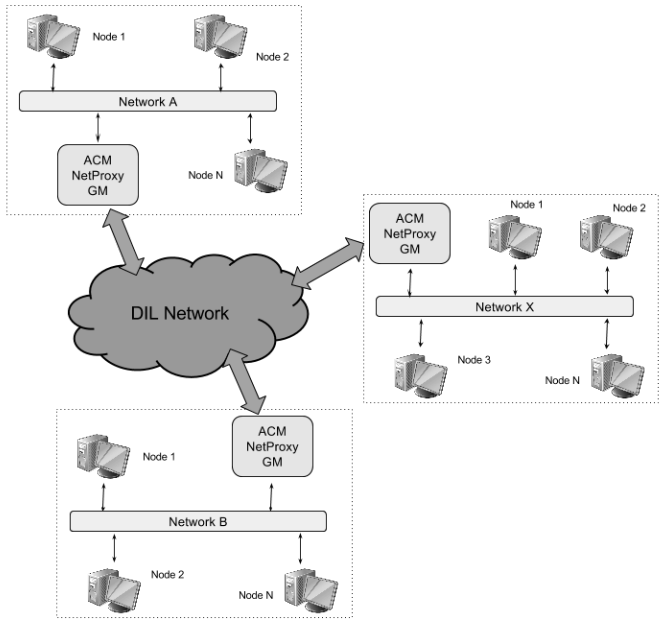

Figure 4: NetProxy Running in Gateway Mode

VI. Experimental Results

This section presents the results collected during two different experiments involving the NetProxy and other associated protocols. The first experiment is designed to reproduce the issues that SoA applications face when running in tactical networks and several tests were run in an emulated environment to collect the necessary data from it. The results of the second experiment, instead, come from a recent technical evaluation event: data represents a real use case and they were collected directly in the field, during the event.

For our first experiment, we used an enhanced version of the Mobile Ad-hoc Network Emulator (MANE) [8], a tool designed to reproduce the characteristics of unreliable environments such as tactical networks, to set up the connectivity between the two nodes involved in the first experiment. Those nodes are part of the NOMADS testbed, which comprises 96 HP DL140 servers (Dual Xeon Dual Core CPUs at 3.06Ghz, with 4GB of RAM each) connected via a 100Mbps Ethernet LAN. MANE can manage bandwidth, latency, and reliability for both directions of each link, and thus it permits evaluating different systems and configurations in a reproducible, laboratory controlled environment. The reliability parameter in MANE is a complementary measure of the Packet Error Rate (PER): for instance, 90% reliability is equal to a PER of 10%.

For the purposes of testing the performance of NetProxy, we had a client application on one node of the testbed generate an HTTP SOAP request, send it to a Web Server located on a second node, and finally wait for the response. To emulate different conditions of a tactical scenario, we kept the link bandwidth set to 1 Mbps in both directions while we ran several tests changing the reliability parameter: we used the values 87%, 90%, 93%, and 95%. The client application repeated each request 50 times under the same link conditions before we input the next reliability value in MANE. The whole experiment was repeated four times, changing the way client and server connected to each other: using TCP, using NetProxy to remap TCP over Mockets (we will refer to this configuration as “NP + Mockets” in the remainder of the paper), using NP + Mockets and enabling the lzma compression feature in NetProxy (NP + Mockets – LZMA), and finally using NP + Mockets and the zlib compression (NP + Mockets – ZLIB). Both instances of NetProxy were running in Host Mode.

Figure 3 shows the results of the experiment described above. TCP shows the lowest throughput and remapping it over Mockets through NetProxy already produces a significant improvement in performance. Several reasons contribute to this result. First of all, Mockets handles packet loss much better than TCP, which attributes it entirely to network congestion thereby triggering congestion control when not necessary. Moreover, NetProxy multiplexes all the traffic directed to a single node onto the same connection, which keeps it open for consecutive requests that may come from various applications; on the contrary, single applications usually don’t keep their TCP connections open once a request has been served, which means that every new request has TCP to go through its slow-start phase. Part of the improvement is also due to the intelligent buffering of NetProxy combined with Mockets, which results in packets with larger payloads, thereby reducing the protocol overhead.

However, enabling compression allowed us to achieve a much higher gain in the measured throughput. In fact, the verbosity of the HTTP and SOAP protocols permits compression algorithms to work very efficiently, sensibly reducing the amount of data that needs to be transferred. Despite the better compression ratio of the lzma algorithm compared to zlib, NP + Mockets – ZLIB showed the highest throughput. This result is due to the greater computational resources required by the lzma algorithm, at which point the computational time spent in compression exceeds the gains achieved in the network transmission time due to the slightly improved compression ratios.

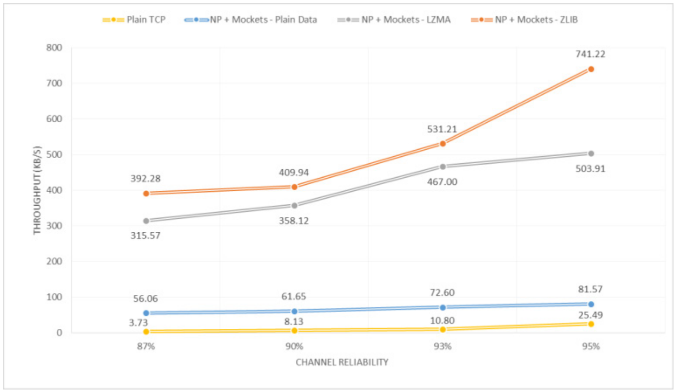

Figure 5: Performance Results of Service Invocation with NetProxy

The second experiment consisted of a practical demonstration in the field. Four networks were involved in the experiment, namely networks NetA through NetD, and only NetA shared a satellite link to every other network. Therefore, direct connections between any of the networks from NetB through NetD were not possible, and so all communications between nodes in two of those networks had to go through NetA. All satellite links have a latency of 2 seconds, hence an RTT of 4 seconds, and a bandwidth of 32KBps (256 Kbps).

NetProxy operating in Gateway Mode was installed on a dedicated Ubuntu 14.04 64-bit Linux machine in each of the four networks, and all incoming and outgoing traffic had to go through those nodes. This conferred on NetProxy complete observability of and control over the amount of traffic generated within the network and of the status of the satellite link. Tcpdump (http://www.tcpdump.org/) was used to capture all packets on both the internal and external network interfaces of the machines running NetProxy.

Due to space limitations, only a subset of the results are presented. Two graphs show the effects of NetProxy on the number of packets generated and on the bandwidth usage, filtering out all traffic but that going from NetA to NetB. Statistical analysis of the other connections, in both directions, showed very similar results. On the X axis of the graphs is the time in seconds. Due to the long duration of the experiment (several hours) and the consequent overwhelming amount of data collected, only a small subset was selected for detailed presentation. This particular data sample is about 80 seconds long and starts at 680s after the beginning of the experiment.

Figure 6 shows the number of packets sent every second by nodes of NetA to nodes of NetB (in red) against the number of packets actually generated every second by the NetProxy in NetA and transmitted over the satellite link to the NetProxy in NetB (in black). As can be seen clearly, the red bars are always significantly taller than the black ones, indicating a much lower resource consumption when using NetProxy. Actual packet counts indicated an improvement of 1.77x (in terms of reduction of the number of packets). Figure 7 shows a similar comparison – but measuring the bandwidth utilization (the unit of measurement is in bytes per second). Actual data analysis shows that the improvement (in terms of reduction of bandwidth) was approximately 2.44x.

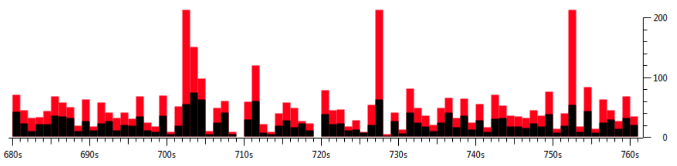

Figure 6: Packet Rate Comparison with NetProxy and Mockets

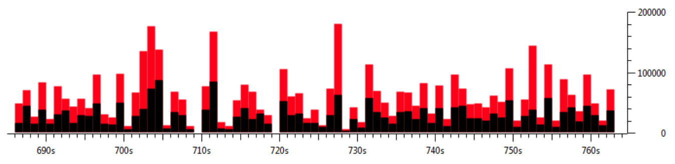

Figure 7: Bandwidth Comparison with NetProxy and Mockets

VII. Summary and Future Work

This paper has described the agile computing middleware (ACM) and its application to supporting Service-oriented Architectures (SoAs) over tactical and DIL networks. NetProxy is the primary component that addresses the challenges of enabling legacy applications and SoAs to achieve better performance over tactical networks. NetProxy integrates the mockets transport protocol that replaces TCP and DisService for disruption-tolerant dissemination. Experimental results both in the laboratory and the field show the significant improvement in performance that can be achieved using this middleware capability. All of the components described in this paper are available via GPLv3 licensing from GitHub [2].

Future work in this area consists of further enhancements to SoA capabilities. In particular, the AgServe component is being updated to support dynamic service deployment and service migration by discovering and exploiting communication and computation resources in a dynamic tactical / DIL network environment.

Acknowledgment

This work is supported by the U.S. Army Research Laboratory under cooperative agreement W911NF-11-2-0095.

References

[1] Suri, N. 2009. Dynamic service-oriented architectures for tactical edge networks. In Proceedings of the 4th Workshop on Emerging Web Services Technology (WEWST '09), Walter Binder and Erik Wilde (Eds.). ACM, New York, NY, USA, 3-10. DOI: 10.1145/1645406.1645408.

[2] Online Reference. http://www.github.com/ihmc/nomads

[3] Suri, N.; Rebeschini, M.; Breedy, M.; Carvalho, M.; Arguedas, M., "Resource and Service Discovery in Wireless AD-HOC Networks with Agile Computing," *Military Communications Conference, 2006. MILCOM 2006. IEEE*, pp. 1-7, 23-25 Oct. 2006. DOI: 10.1109/MILCOM.2006.302212

[4] Suri, N.; Marcon, M.; Quitadamo, R.; Rebeschini, M.; Arguedas, M.; Stabellini, S.; Tortonesi, M.; Stefanelli, C., "An Adaptive and Efficient Peer-to-Peer Service-Oriented Architecture for MANET Environments with Agile Computing," *Network Operations and Management Symposium Workshops, 2008. NOMS Workshops 2008. IEEE*, pp. 364-371, 7-11 April 2008. DOI: 10.1109/NOMSW.2007.56

[5] Benvegnu, E.; Suri, N.; Hanna, J.; Combs, V.; Winkler, R.; Kovach, J., "Improving timeliness and reliability of data delivery in tactical wireless environments with Mockets communications library," *Military Communications Conference, 2009. MILCOM 2009. IEEE*, pp.1-8, 18-21 Oct. 2009. DOI: 10.1109/MILCOM.2009.5379951

[6] Perkins, C (Ed). IP Mobility Support for IPv4, Revised. Internet Engineering Task Force (IETF) Request for Comments (RFC) 5944.

[7] Suri, N. and Benincasa, G. Opportunistic listening system and method. U.S. Patent 8493902 B2.

[8] N. Ivanic, B. Rivera, B. Adamson, “Mobile Ad Hoc Network Emulation Environment”, in Proceedings of 2009 IEEE Military Communications Conference (MILCOM 2009).