MOCKETS: A COMPREHENSIVE APPLICATION-LEVEL COMMUNICATIONS LIBRARY

Niranjan Suri[1,2], Mauro Tortonesi[1,3], Marco Arguedas[1], Maggie Breedy[1], Marco Carvalho[1], Robert Winkler[4]
[1]Institute for Human & Machine Cognition, University of West Florida
[2]Lancaster University
[3]University of Ferrara
[4]U.S. Army Research Laboratory
{nsuri,mtortonesi,marguedas, mbreedy,mcarvalho}@ihmc.us; winkler@arl.army.mil

ABSTRACT

Mockets is a comprehensive communications library designed to address challenges specific to mobile ad-hoc networks. Mockets have been implemented at the application-level to simplify deployment and portability. Both stream-oriented and message-oriented abstractions are supported, with the message-oriented service providing multiple classes of service (reliable, unreliable, sequenced, unsequenced), message tagging and replacement, and prioritization. Mockets also interfaces with a policy management infrastructure to support bandwidth limitation. Finally, mockets supports transparent migration of communication endpoints across hosts without the need to terminate and reestablish connections. Mockets provides similar semantics to TCP but performs better than TCP on ad-hoc networks.

INTRODUCTION

Mockets[1] (for "mobile sockets") is a comprehensive communications library for applications. The design and implementation of Mockets was motivated by the needs of tactical military information networks, which are typically wireless and ad-hoc with low bandwidth, intermittent connectivity, and variable latency. The initial implementation of Mockets was completed for use by the Army Research Laboratory as part of the Warrior's Edge initiative of the Horizontal Fusion Portfolio's Quantum Leap demonstrations.

Mockets addresses specific challenges including the need to operate on a mobile ad-hoc network (where TCP does not perform optimally), provides a mechanism to detect connection loss, allows applications to monitor network performance, provides flexible buffering, and supports policy-based control over application bandwidth utilization.

Mockets has been designed to provide the following five capabilities:

1) Application-level implementation of the communications library in order to provide flexibility, ease of distribution, and better integration between the application and the communications layer.
2) A TCP-style reliable, stream-oriented service that is designed to operate on wireless ad-hoc networks thereby making it easy to port existing applications to the ad-hoc environment.
3) A message-oriented service that provides enhanced capabilities such as message tagging and replacement, different classes of service (reliable/unreliable combined with sequenced/ unsequenced), and prioritization.
4) Transparent mobility of communication endpoints from one host to another in order to support migration of live processes with active network connections.
5) Interface to a policy management system in order to allow dynamic, external control over communications resources used by applications.

The result is a flexible user-level communications library with implementations in Java, C++, and C#. The performance of Mockets is equal to or better than TCP for the type of networks that were targeted, while providing the additional desired capabilities.

The rest of this paper is organized as follows. Section two presents the design of Mockets. Section three provides some implementation details. Section four presents experimental results and discussion. Section five briefly describes the message mockets API. Finally, section six concludes the paper.

MOCKETS DESIGN

The stream-oriented communication metaphor supported by Mockets provides TCP socket semantics and has been designed to bridge the gap between ad-hoc networks and networking applications - typically written expecting TCP stream socket communication semantics.

Prior research has demonstrated that TCP performs poorly when deployed in ad-hoc environments. Several researchers have investigated the shortcomings of TCP in the context of wireless and ad-hoc networks [1] and suggested modifications to improve performance [2] [3]. These studies show that the performance of the protocol

[1] In this paper, a capitalized "Mockets" refers to the software package while a lowercase "mockets" is the plural of "mocket".

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on November 9, 2009 at 13:53 from IEEE Xplore. Restrictions apply.

is severely affected in case of terminal mobility and lossy channels, since the protocol interprets lost packets as congestion and reduces the window sizes, with very slow throughput recovery [4] [5] [6] [7]. Although the proposed modifications to the TCP protocol show promising improvements in simulations and experiments, they have failed to be deployed in a widespread manner. The difficulty is in converging upon a standard set of modifications from the many alternatives available so that they may be ubiquitously realized in a wide variety of systems. To date, no such standard has been developed.

The main design goal for stream mockets is to maintain basic compatibility with traditional TCP stream sockets to ease the task of porting legacy applications to the new library. Mockets provides networking applications with the ability to establish unicast, bidirectional, and reliable stream-based communications with peer applications. It implements an API that is very similar to the traditional Java/BSD socket API, extended with some additional capabilities that will be described later in this section. For example, the Java implementation of Mockets provides the Mocket, ServerMocket, MocketInputStream, and MocketOutputStream classes that are counterparts to the Java Socket, ServerSocket, SocketInputStream, and SocketOutputStream classes.

For backwards compatibility with legacy TCP-based applications, the communication protocol adopted for stream mockets is similar to TCP. However, it also includes several additional features to make it better suited for deployment in ad-hoc networks and provides reasonable performance, reliability, and robustness in highly dynamic environments.

All the messages exchanged by the Mockets communication protocol are embedded in UDP packets. Reliability and stream abstractions are provided by Mockets on top of the unreliable UDP packet delivery service.

Similar to the Java/BSD sockets API, communications between two unicast datagram mockets is established by connecting an active mocket to a passive one listening on a peer that (apart from explicit endpoint migration commands issued at middleware or application level) will not change during the entire communication.

Although based on a three-way handshake, the connection setup procedure for a stream mocket is different from the TCP socket handshake. On the server side the communication is not established on the same port on which the server application listens for incoming connections, but on a new, system-assigned port.

The connection establishment process for the stream mockets transport protocol is also optimized to work on the highly dynamic and typically unreliable scenario of ad-hoc networks. To improve the robustness of the connection setup procedure, if a server mocket receives more than one connection request message (the equivalent of a SYN packet in TCP) from the same peer application, it assumes that SYN_ACK replies are not reaching the sender and it automatically increases the frequency of acknowledgements for that connection. In such cases, the frequency of acknowledgements is increased progressively, at random intervals, to increase the probabilities of packet delivery to the sender. This self-regulating acknowledgement mechanism allows mockets to dynamically adapt to specific network conditions on a per-connection basis, increasing the chances of a successful connection and avoiding unnecessary additional traffic for protocol negotiation. The connection teardown is similar to TCP except that mockets allows the application to limit the wait time for flushing data before closing the connection.

The Mockets communication protocol can also interact with lower level protocols to find out if the cause for packet loss is congestion, link error, terminal disconnection, or route change. Simulations and quantitative studies show that correctly exploiting this kind of information in ad-hoc environments can lead to a performance improvement in the transport protocol throughput of up to 700% with respect to standard TCP implementation [8].

Applications can query a mocket to retrieve information on the current status of the communication with the remote endpoint. The statistics reported are the number of bytes and packets sent and received, the number of packets retransmitted, and the number of discarded received packets. Applications can query this information if they desire to monitor the performance and reliability of the connection and adapt their behavior if these parameters fall below their desired QoS level.

The Mockets API allows the application to take full control of most of the internal behaviors (e.g. changing the default timeouts) of the transport protocol. This allows the development of robust applications that, in case of heavy packet losses along the communication path, may choose to take appropriate action (e.g., downscaling the stream of application level data whenever possible, sending all messages using reliable flows in case of message mockets, increasing the retransmission timeout) in order to adapt the communication to current network conditions while improving reliability.

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on November 9, 2009 at 13:53 from IEEE Xplore. Restrictions apply.

Keep-Alive and Connection-Loss Detection

The Mockets protocol provides a keep-alive mechanism, which is enabled by default. The communications behavior pattern of some applications is of the nature where one end-point simply listens for updates arriving from the other end-point at random intervals. If no messages have been sent to or received from the peer application during a (configurable) amount of time, Mockets will automatically send a heartbeat message to the remote communication endpoint. This simple keep-alive mechanism allows quick detection of problems at the link and network layers. Without keep-alives, the receiving end-point does not become aware of a lost connection.

This keep-alive mechanism is similar to the one introduced in TCP, but more flexible and powerful. Mockets provide the application with the option of registering a callback, which is invoked when the endpoint fails to receive any data or keep-alive packets for more than two seconds. The transmitting endpoint will generate a heartbeat every second if no data packets need to be sent. Therefore, the receiving endpoint should receive at least two keep-alive packets within the timeout period. If no data or keep-alive packets arrive, the receiving endpoint will trigger the callback to warn the application that the peer is unreachable. The application callback is provided with a value that indicates the elapsed time since last contact. Note that the application can choose the appropriate course of action when receiving this warning. If the application does nothing, the mocket endpoint will continue normally, but will continue generating the warnings every second. One simple behavior an application can adopt is to wait for some interval of time (for example, 30 seconds), and then assume that the connection is lost and ask the mocket to close the local endpoint.

The connection loss mechanism can also be integrated with a middleware that manages communications resources. In the current implementation, this mechanism is integrated with the agile computing middleware [9] [10], which can proactively physically move resources in mobile ad-hoc wireless environment to restore lost communications, or to facilitate necessary communication.

Mockets also provides an optional monitoring component – the MocketStatusMonitor. If the MocketStatusMonitor is started, all mockets on the host transmit information to the MocketStatusMonitor. This information includes connections established, failed connection attempts, statistics, and warnings about unreachable peers. The MocketStatusMonitor can react to the failed connection attempts and unreachable peer warnings appropriately.

Flexible Buffering Options

Mockets adopts the Nagle algorithm to minimize the number of packets sent over the network, thus optimizing the average header size/payload size ratio and therefore the channel usage efficiency. Instead of just supporting the ability to enable or disable this option, Mockets provides a flexible buffering time for outgoing data. Applications are free to choose any appropriate buffering time based on their requirements or turn off buffering completely, which is the equivalent of disabling the Nagle algorithm. This capability is useful for applications that need different limits on how long data may be buffered before transmission. For example, a real-time control application may set the transmission buffer time to 10 ms instead of 100 ms in order to reduce latency introduced by buffering.

Transparent Mobility

Some distributed systems rely on process migration – the ability to capture the execution state of a process and move the state to a remote system over a network connection. Process migration is used in situations ranging from load balancing to mobile agent systems that provide strong or forced migration [11]. Handling open network connections is a challenging problem with process migration. If a process which has an open connection must be migrated involuntarily, the network connection must be restored after the process reaches the destination host. Otherwise, the semantics of the migration will not be transparent to the process.

The Mockets library supports the transparent migration of a mocket endpoint. Before an endpoint is moved, the mocket connection can be suspended at which point the remote endpoint will enter a standby state. The local endpoint can then be migrated to a new host and the connection resumed. This will result in the local endpoint reconnecting transparently with the remote endpoint. The mocket on the remote endpoint is notified of the address change in the communication endpoint and acts properly; the remote application does not need to be aware of the temporary suspension of the connection. The local endpoint can be migrated using a variety of mechanisms. For example, in the Java environment, all the object instances that implement the stream mocket in use by the local endpoint can simply be serialized and transferred over a network link.

Figure 1 below shows the process of a mocket migrating from one host to another. The initial state is Process 1 running on Host A with an open connection to Process 2 on Host C. When Process 1 needs to move to Host B, the mocket in Process 1 sends a SUSPEND control message to the mocket in Process 2. Once the SUSPEND has been

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on November 9, 2009 at 13:53 from IEEE Xplore. Restrictions apply.

acknowledge with a SUSPEND_ACK, the process is allowed to migrate along with the mocket endpoint. Once the process reaches Host B and restarts, the mocket in Process 2 sends a RESUME control message. The state of both mockets returns to ESTABLISHED after Process 2 receives the RESUME_ACK control message.

Transparent mobility is currently supported only by the Java implementation, which complements the agile computing middleware [9] [10] and the NOMADS mobile agent system [12] that provides strong migration. This capability can be implemented in the C++ and C# versions if a need arises.

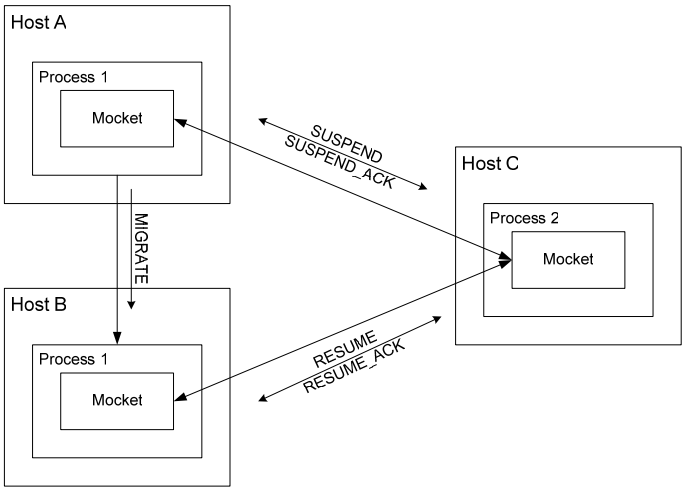

Figure 1: Migration of a mocket Endpoint

The stream mockets transport protocol is not meant to be a full-featured replacement of TCP. There is no immediate plan to support advanced features like out-of-band data and half-close mechanism, as these features are not commonly used by TCP-based networking applications. These capabilities could, however, be incorporated as needed. Path Maximum Transmission Unit (MTU) discovery, on the other hand, will probably never be implemented, since in the highly dynamic scenario of ad-hoc networks it would lead to considerable performance degradation without introducing any significant improvement to throughput or channel usage efficiency.

IMPLEMENTATION DETAILS

Figure 2 shows the implementation of a mocket and a server mocket. The exact API depends on the language. For example, the Java version of Mockets provides subclasses of InputStream and OutputStream to comply with the standard Java I/O API. The C++ version simply provides read() and write() calls modeled after the standard BSD sockets API. Each mocket contains a Transmitter and a Receiver object, which operate as independent threads. The two objects share a UDP socket that is used for the underlying communication.

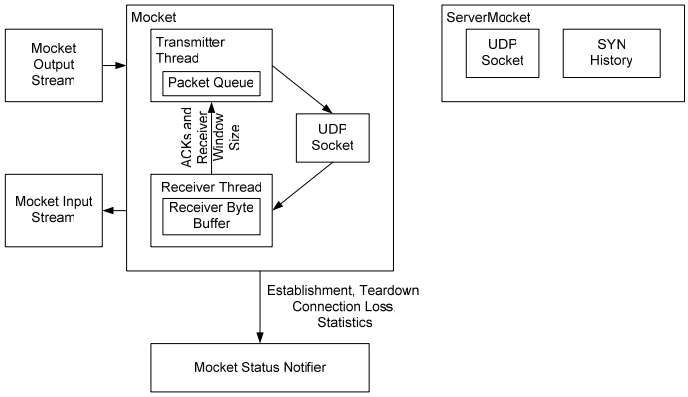

Figure 2: Mockets Implementation Details

When the application sends data over the UDP socket, the data is passed to the Transmitter object that sends it over the network (performing buffering according to the Nagle algorithm when needed and respecting the window size advertised by the remote endpoint) and stores it in the outstanding packet queue, where it will remain waiting to be acknowledged.

In a similar way, when an incoming packet arrives at the UDP socket, the receiver thread processes the data and enqueues it in the ReceiverByteBuffer. When the application issues a read data request to the mocket, the request is forwarded to the Receiver which returns the data previously stored in the ReceiverByteBuffer.

Like TCP sockets, Mockets-based communications are established by connecting an active mocket to a remote (passive) endpoint. The passive side is represented by a server mocket, which has an accept method for an application to receive incoming connection requests. All the processing of the incoming connection happens in the accept method of the server mocket, which is expected to be called continuously by the application. The server mocket keeps track of all incoming connections to handle retransmitted SYN packets properly.

Note that the current implementation uses two threads per mocket endpoint, one in the receiver to continuously read from the UDP socket and the other in the transmitter to retransmit packets if they are not acknowledged within the timeout interval. While this design results in a clean and simple implementation, the number of active threads implies a limit on the scalability and hence is an issue with applications that create a large number of endpoints. There are two alternative approaches. The first approach would be to use common transmitter and receiver threads on a per application or per VM basis. This would result in a significant reduction in the number of threads but does add some complexity to the implementation. The second

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on November 9, 2009 at 13:53 from IEEE Xplore. Restrictions apply.

alternative is to push the mocket implementation down into the operating system kernel, thereby requiring only two threads for a whole system. However, this approach is not ideal because the framework would no longer be realized purely at the application level, thereby making it more difficult to deploy and to support multiple platforms and operating systems.

The MocketStatusNotifier component uses a loopback UDP socket to send connection setup, teardown, connection loss warning, and statistical information. This information is transmitted to a UDP port on the local host. An application (or middleware) may choose to use a MocketStatusMonitor component, which would receive the notifications sent from every mocket on the local host. As described earlier, the monitoring component can use this information to implement behaviors such as manipulating the physical nodes in order to change the network configuration.

EXPERIMENTAL RESULTS

Two experiments were conducted to measure the performance of mockets and compare the performance with respect to TCP sockets. In both cases, the experiments measured the time to transmit 2 megabytes of data from a client to a server and receive an acknowledgement back from the server.

The computers for the first set of experiments were two Fujitsu Stylistic ST4000 series tablet computers running Windows XP Table PC edition. Six different network configurations were used to measure the performance of both mockets and sockets. The first network configuration was an Ethernet LAN at 10 Mbps. In order to minimize the impact of other random traffic, a crossover cable was used between the two computers. The second network configuration was 802.11b in ad-hoc mode. The Wireless LAN cards used for this test were Orinoco Gold 11 Mbps cards.

The remaining four network configurations were using ad-hoc routing nodes designed by BBN for the Warrior's Edge initiative of the Horizontal Fusion Portfolio. Four different configurations were created to measure the performance – with two nodes (one hop), three nodes (two hops), four nodes (three hops), and five nodes (four hops). In all cases, the nodes were physically positioned in order to guarantee that the data was flowing through the specified number of hops.

The tablet computers connected to the ad-hoc routing nodes using a standard 10 Mbps Ethernet LAN. The ad-hoc nodes themselves were operating on 802.11g wireless cards. All of the wireless cards were configured to use channel 1, which was not used by any other device in the test environment.

Table 1 below shows the performance comparison between Mockets and TCP sockets. For these tests, the C++ implementation of Mockets was used. For each test, the experiment alternated between Mockets and TCP sockets. The tests on the 10 Mbps LAN and 802.11b Ad-Hoc were repeated 100 times whereas the tests with the BBN nodes were repeated 200 times. The experiment shows that on the wired 10 Mbps LAN, Mockets perform worse by 4.27% whereas in the wireless configurations, Mockets outperformed TCP sockets. In the case of 802.11b, the performance improvement was 22.68% and with the BBN nodes in the 4 hop configuration, the performance improvement was 21.43%.

The second experiment measured the performance of the C++ and Java versions of Mockets on both Windows and Linux operating systems. The goal was to compare both the effect of the runtime platform as well as the underlying operating system. The computers were two IBM ThinkPad laptops running Windows XP or Linux (kernel version 2.6). The Wireless LAN cards were Orinoco Gold 11 Mbps cards. The 10 Mbps Ethernet LAN test was conducted using a crossover cable. For the Java benchmarks, version 1.4.2 of Sun's Java Runtime Environment was used as the virtual machine.

Table 1: Throughput (bytes/ms) Comparison Between Mockets and Sockets

	Mockets	Sockets	Improvement
Configuration	Throughput (bytes/ms)		
10 Mbps LAN	1078.11	1126.18	-4.27%
802.11b Ad-Hoc	503.02	410.02	22.68%
BBN 1 Hop	856.53	847.35	1.08%
BBN 2 Hops	345.30	292.64	17.99%
BBN 3 Hops	164.98	147.65	11.73%
BBN 4 Hops	97.99	80.70	21.43%

Table 2: Throughput (bytes/ms) Comparison Between C++ and Java Implementations of Mockets on Win32 and Linux Platforms

	802.11b Ad-Hoc	10 Mbps LAN
Win32/C++	492	1097
Win32/Java	499	1174
Linux/C++	508	1046
Linux/Java	544	1204

Table 2 above show the results of the experiments. Java slightly outperformed C++ on both Linux and Windows, indicating that there is some room for optimization of the C++ version.

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on November 9, 2009 at 13:53 from IEEE Xplore. Restrictions apply.

OVERVIEW OF THE MESSAGE MOCKETS API

Mockets also provides a message oriented communication protocol designed to provide additional functionality beyond that of both TCP and datagram sockets. Figure 5 below shows the Mockets library hierarchy.

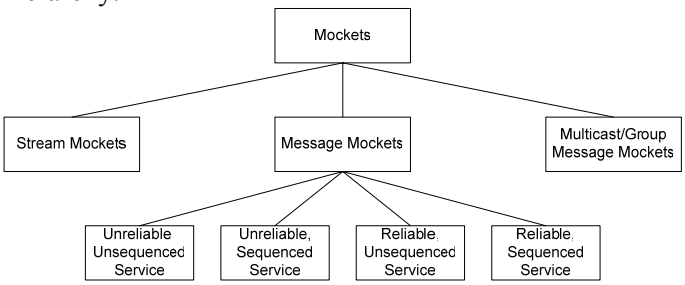

Figure 5: Mockets Hierarchy

Message-oriented mockets provide applications with the ability to send messages to a peer application by using different delivery services (called "flows") with specific communication semantics. After connection establishment, the application will be able to select one or more flows and use them to send data to the remote endpoint. The application can choose the characteristics of each flow and set the transmission parameters for the messages to be sent on the flows.

The parameters that an application can specify, when selecting a flow, are message sequencing and reliability. Flows can carry both sequenced and unsequenced messages. Using a sequenced flow ensures that the data will be delivered to the peer application in the same order in which it was sent. The application can also request reliable messages. In this case, all messages will be acknowledged by the receiver. Message sequencing and reliability are orthogonal characteristics and an application can use flows with any of combination of these two parameters.

In addition, for each flow, the application can choose several transmission parameters. Message priority is a parameter that forces Mockets to give preference to a specific class of messages when scheduling them for transmission. This can be valuable in the case of time-sensitive and control applications. However, a careful implementation of the output packet scheduler is required in order to achieve good performance and avoid starvation of low priority packets while giving the applications all the necessary flexibility in communication semantics. Mockets accomplishes this by dynamically raising the priority of enqueued messages each time they are passed over.

Applications may also classify the data sent over the flow into different message types. This classification allows applications using flows to either delete or replace all previously enqueued messages of a specific type. (Messages with no assigned type cannot be deleted or replaced.) This feature is useful in situations where applications are sending periodic updates and a new update invalidates previous updates. When using a reliable flow, the mockets framework will buffer messages until they have been successfully received by the remote endpoint. If the network is unreliable, messages can accumulate. Using the tag and replace feature, mockets will be able to remove all previous updates that are still in the queue and replace them with the latest update. This reduces transmission of unnecessary data and hence reduces network bandwidth utilization.

The tag and replace feature also supports applications that use multiple but interrelated data types such as MPEG. For instance, an MPEG streaming video application might use two different message tags, one for keyframes and a second for delta frames. The application could then choose to replace any enqueued but unacknowledged keyframes by first deleting all of the pending delta frames and then replacing the pending keyframes with the latest keyframe.

The message mocket can optionally be configured to perform cross sequencing. Messages belonging to the sequenced flows will have both an intra-flow sequence number and a cross-flow sequence number. In this way, the application will be able to make sure that messages sent over different flows will be delivered to the peer application in the desired order.

Applications can also select the values of enqueue and retransmission timeouts. The enqueue timeout is a timeout for messages to be included in the transmission/pending message queue. If the timeout expires the message is not enqueued and an error is returned to inform the application. The retransmission timeout is the timeout for the retransmission of a message on a reliable flow waiting for an acknowledgement. If this timeout expires before an acknowledgement arrives, the message will be removed from the outstanding message queue and silently discarded.

With this architecture, messages having the same application-level semantics can be sent over several flows having different communication semantics (e.g., a temperature sensor could decide to send a temperature update message over a high-priority reliable flow for every n temperature update messages sent over a low-priority unreliable flow).

However, it is important to realize that the flow-based architecture of message mockets is not designed to allow distributed applications to perform multiplexing of data at

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on November 9, 2009 at 13:53 from IEEE Xplore. Restrictions apply.

the application level. In fact, the only application which is aware of the presence of flows is the one on sender side, as all the data sent over the different flows will be delivered to the peer application through the receive method. The receiving application has absolutely no knowledge of how many different flows are used in the data transfer process.

As in stream mockets the message mocket communication protocol is UDP based. In order to make the message mocket protocol more resistant to SYN flood attacks, the connection startup procedure is a 4-way handshake like SCTP [13].

CONCLUSIONS

Mockets is an application-level communication library capable of providing TCP-like semantics over UDP. The API is designed to facilitate porting of existing networking applications easily. Mockets provides better performance than TCP over wireless ad-hoc networks. Mockets also supports transparent mobility of the connection endpoints, keep-alives and connection loss detection, and flexible buffering options. Applications can query Mockets to obtain statistics such as packets discarded and retransmission counts, thereby allowing themselves to dynamically adapt to the underlying network. Mockets has been implemented in Java, C++, and C#. Mockets also provides support for message-oriented communications, along with a suite of new features to better support applications operating in unreliable networks. The Mockets library along with the source code is available free for non-commercial use.

ACKNOWLEDGEMENTS

This work is supported in part by the U.S. Army Research Laboratory under contract W911NF-04-2-0013, by the U.S. Army Research Laboratory under the Collaborative Technology Alliance Program, Cooperative Agreement DAAD19-01-2-0009, and by the Office of Naval Research under grant N00014-03-1-0780. The authors would also like to thank Steven Choy and Jessie Kovach at the U.S. Army Research Laboratory for their efforts in integrating Mockets into their applications.

REFERENCES

[1] B. S. Bakshi, P. Krishna, N. H. Vaidya, D. K. Pradhan, Improving Performance of TCP over Wireless Networks, in: Proceedings of 17th Int. Conf. Distributed Computing Systems, Baltimore, May 1997.

[2] K. Chandran, S. Raghunathan, S. Venkatesan, and R. Prakash, A feedback based scheme for improving TCP performance in ad-hoc wireless networks, in: Proceedings of International Conference on Distributed Computing Systems, Amsterdam, 1998.

[3] Z. Fu, B. Greenstein, X. Meng, S. Lu, Design and Implementation of a TCP-Friendly Transport Protocol for Ad Hoc Wireless Networks, in: Proceedings of 10th IEEE International Conference on Network Protocols (ICNP'02), Paris, November 2002.

[4] J. Calagaz, W. Chatam, B. Eoff, J. A. Hamilton Jr., On the Current State of Transport Layer Protocols in Mobile Ad hoc Networks, in: Proceedings of the 42nd annual ACM Southeast regional conference, Special session on mobile computing, pp. 76—81, 2004.

[5] Z. Fu, P. Zerfos, H. Luo, S. Lu, L. Zhang, M. Gerla, The Impact of Multihop Wireless Channel on TCP Throughput and Loss

[6] G. Holland, N. Vaidya, Analysis of TCP Performance over Mobile Ad Hoc Networks, in: Proceedings of IEEE/ACM MOBICOM '99, August 1999.

[7] X. Chen, H. Zhai, J. Wang, Y. Fang , TCP Performance over Mobile Ad Hoc Networks, in: Canadian Journal of Electrical and Computer Engineering, April 2004.

[8] Z. Fu, X. Meng, S. Lu, How bad TCP can perform in mobile ad hoc networks, in: Proceedings of IEEE International Symposium on Computers and Communications (ISCC'02), Taormina, Italy, July 2002.

[9] N. Suri, J. Bradshaw, M. Carvalho, M. Breedy, T. Cowin, R. Saavedra, S. Kulkarni, Applying Agile Computing to Support Efficient and Policy-controlled Sensor Information Feeds in the Army Future Combat Systems Environment, in: Proceedings of Collaborative Technologies Alliance Conference (CTA 2003), 2003

[10] N. Suri, J. Bradshaw, M. Carvalho, T. Cowin, M. Breedy, P. Groth, R. Saavedra, Agile Computing: Bridging the Gap between Grid Computing and Ad-hoc Peer-to-Peer Resource Sharing, in: Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003), 2003.

[11] D. Milojicic , F. Douglis, R. Wheeler. Mobility: Processes, Computers, and Agents. ACM Press.

[12] N. Suri, J.M. Bradshaw, M.R. Breedy, P.T. Groth, G.A. Hill, and R. Jeffers. Strong Mobility and Fine-Grained Resource Control in NOMADS. Proceedings of the 2nd International Symposium on Agents Systems and Applications and the 4th International Symposium on Mobile Agents (ASA/MA 2000). Springer-Verlag.

[13] R. Stewart, Q. Xie, Stream Control Transmission Protocol (SCTP), Addison-Wesley, November 2001.

Authorized licensed use limited to: Universita degli Studi di Bologna. Downloaded on November 9, 2009 at 13:53 from IEEE Xplore. Restrictions apply.