MOCKETS: A NOVEL MESSAGE-ORIENTED COMMUNICATIONS MIDDLEWARE FOR THE WIRELESS INTERNET

Mauro Tortonesi, Cesare Stefanelli
Department of Engineering, University of Ferrara, Via Saragat 1, 44030 Ferrara, Italy

Niranjan Suri, Marco Arguedas, Maggie Breedy
Institute for Human & Machine Cognition, 40 S. Alcaniz, 32502 Pensacola, FL, USA

Keywords: Novel Network Programming Model, Application-level Middleware, Endpoint Mobility, TCP Replacement.

Abstract: Wireless networking is becoming increasingly important for ubiquitous access to the Internet and the Web. However, wireless networks exhibit significant reliability and performance problems, with frequent disconnections, congestions, and packet losses. For these reasons, the traditional TCP/IP suite, designed for wired networks, offers poor performance and inadequate communication semantics in this scenario. There are several research efforts in both protocols and communication infrastructures aimed at producing solutions better suited to wireless network characteristics. This paper presents Mockets, a novel communications middleware specifically designed for wireless networking scenarios. The Mockets middleware permits a communication endpoint to be moved from one node to another without interrupting the communication session. In addition, Mockets provides several delivery services with different communication semantics, semantic classification of data, cancellation/replacement of enqueued data, and priority/lifetime assignment to messages. Initial experimental results in a wireless network scenario show that the Mockets middleware achieves better performance levels than traditional TCP-based infrastructure.

1 INTRODUCTION

Wireless networks are quickly becoming prevalent and their popularity is expected to grow even more, as they permit to easily extend the wired Internet infrastructure thus facilitating the ubiquitous access of mobile users/terminals to the Internet and the Web (Stallings, 2005). In this paper, we refer to the above mentioned environment with the term *wireless Internet*.

However, the radio frequency medium of wireless networks induces some peculiar operating condition characteristics, such as low reliability levels, network disconnections, severe fluctuations in network resources availability, and a dynamic topology. These characteristics deteriorate the performance of traditional communication protocols so much (Altman et al., 2000) (Abouzeid et al., 2003) that several research studies have tried to modify the inner workings of traditional protocols to better suit the wireless Internet scenario (Tian et al., 2005). Although this approach would allow existing applications to remain unchanged, the performance results are not satisfactory. In addition, traditional protocols and communication infrastructures do not provide support for the mobility of users and terminals (Fu et al., 2006).

The peculiar characteristics of the wireless Internet scenario suggest that the programming model offered by traditional communication protocols and infrastructures is not adequate. Researchers have proposed different programming model approaches aimed at making it possible for distributed applications to adapt their behavior dynamically to current network conditions (Gross et al., 1999) (Kim and Noble, 2001). To this end, there is a need for a novel communications middleware that, on the one hand, can offer applications the needed network level information and, on the other hand, can handle peculiar wireless Internet characteristics such as user/terminal mobility (Snoeren and Balakrishnan, 2000). This permits the realization of distributed applications that can cope with packet losses, network disconnections, highly dynamic channel conditions, and also user/terminal mobility (Cheng and Marsic, 2002) (Sun et al., 2003) (Chang et al., 2001).

In this context, the paper presents a novel communications middleware, called Mockets, specifically designed to address the peculiar challenges of the wireless Internet scenario. A mocket (mobile socket) is a communication endpoint that can move from one node to another without interrupting the communication session.

Mockets-based applications can exploit several delivery services with different communication semantics for different types of information. The Mockets middleware also provides the semantic classification of messages to permit applications to perform group operations on specific types of messages, such as cancellation/replacement of some kind of enqueued messages. In addition, Mockets permits fine-tuning the performance of applications by setting the transmission priority and the maximum lifetime of messages.

Finally, the Mockets middleware permits the design of applications that can exploit information about network resources availability in order to adapt to the unreliability of the wireless Internet and the intrinsic mobility of terminals and users.

We have decided to implement Mockets as an application layer middleware in order to achieve portability and ease of integration, and to facilitate its deployment in all platforms supporting the TCP/IP protocol suite, regardless of the underlying hardware and operating system.

The experimental results show that applications in the wireless Internet scenario achieve better performance with the Mockets transport than with TCP. In fact, in our tests Mockets outperformed TCP in terms of throughput and latency on both a real IEEE 802.11b wireless network and a simulated environment with random network disconnection intervals.

2 APPLICATIONS IN THE WIRELESS INTERNET

The wireless Internet is significantly different from the wired networking environment and presents peculiar challenges to the development of distributed applications. In fact, wireless communications exhibit lower reliability levels and severe fluctuations in network resource availability. This stems from the inherent characteristics of radio communication systems, which present channel degradation due to fading and interferences, resulting in highly variable bandwidth with time and spatial dependencies. In addition, mobility causes additional problems because the communication path can change significantly as mobile terminals/users roam from one network to another.

The unreliability of wireless communications and the mobility of terminals/users have a significant impact on the development and deployment of distributed applications. In fact, to perform continuous service provisioning in the wireless Internet, applications must be capable of withstanding abrupt disconnections and changes in both network topology and resource availability. However, this requirement clashes with the traditional programming model for distributed applications that is network transparent and makes use of an abstraction of the network as a reliable stream-oriented communication channel. Applications simply hand over their data to the transport protocol and rely completely on the protocol implementation to perform reliable and sequenced information delivery. Unfortunately, this simple interaction model makes it impossible to design applications that can adapt to current network conditions and resource availability. This limits both performance and robustness of applications when deployed in the wireless Internet scenario (Gross et al., 1999) (Kim and Noble, 2001).

As a result, researchers in several areas have proposed novel programming models that do not masquerade communication channel characteristics but instead expose network conditions to applications (Cheng and Marsic, 2002). This allows applications to react to changes in the underlying network in a timely manner by choosing the proper adaptation strategy according to service logic, user preferences, and network status. For instance, a video streaming application could decide to downscale its service level, e.g., the video resolution or the frame rate, in order to adapt to reduced bandwidth availability.

A programming model that can exploit the significant information about the network conditions in order to permit application adaptation requires specific support in terms of communication protocols and/or middleware (Sun et al., 2003). In particular, the paper focuses on the middleware approach because a solution at the application level facilitates portability and ease of integration. In addition, the middleware solution allows the deployment in all platforms supporting the TCP/IP protocol suite, regardless of the underlying hardware and operating system.

Middleware solutions for the support of applications in the wireless Internet should provide several important characteristics.

First of all they should provide the mechanisms to monitor network conditions and to detect the mobility of users/terminals. In fact, it is crucial for applications to have precise and up-to-date knowledge of the network conditions in order to perform correct decisions about service adaptation. The middleware should also support user/terminal mobility by maintaining the current service session in the presence of temporary network disconnections. The mobility of the service session, that requires detecting changes and binding/rebinding connections transparently to applications is still an active area of research (Fu et al., 2006) (Snoeren and Balakrishnan, 2000) (Hsieh et al., 2004).

In addition, a middleware in the wireless Internet has to provide several delivery services with different communication semantics. This would allow applications to choose the best suited message delivery service depending on application logic, network conditions, and user preferences. For instance, applications could assign a higher transmission priority and a shorter lifetime to time-sensitive information, and use reliable delivery only for critical data.

The unreliability of wireless communications suggests the introduction of specific mechanisms at the middleware level to support applications with critical response time requirements. For instance, a remote control application needs to convey time-sensitive data, such as the commands for the movement of a robot working in a hazardous environment and needs to send periodic update messages that change the application status. These updates invalidate all previous messages and must be delivered and processed with the utmost precedence, for example to immediately stop the robot in an emergency situation.

Applications could benefit from a novel middleware that enables a more effective fit to the wireless Internet environment. For instance, let us consider a MPEG video streaming application that transmits only a small amount of reference video frames (key frames) entirely, adopting differential encoding for all the other frames (delta frames). A distributed application could then exploit both sequential and time-bound message delivery for MPEG video frames. In addition, it could select reliable delivery and longer lifetime for messages containing key frames and unreliable delivery and short lifetime for messages carrying delta frames. In the case of channel condition degradation, the application might choose to reduce frame rate or stream resolution, according to service logic and user preferences. In case the client becomes unreachable (or when sending a new key frame), the server invalidates all previously enqueued messages, thus replacing old frames with up-to-date information.

3 THE MOCKETS MIDDLEWARE

Mockets is a middleware that supports the mobility of communication endpoints, with the goal of facilitating user/terminal/code mobility. It provides applications with several types of communication semantics, permits message differentiation, and offers advanced fine-grained configurability to achieve best performance tuning. The Mockets middleware offers application level control and monitoring of the connection status and network conditions.

Mockets adopts the traditional client/server programming paradigm of Sockets (Mockets stands for Mobile Sockets) and provides a message-oriented communication API with advanced functionalities to manage endpoint mobility and monitor network conditions. Mockets also offers a second, stream-oriented API compatible with TCP Sockets to facilitate the task of porting legacy applications to the new middleware. However, applications using the stream-oriented API will not benefit from the advanced functionalities of Mockets. The TCP-like stream-oriented API of Mockets has been presented in a previous paper (Suri et al., 2005) and hence this paper focuses on the message-oriented API only.

3.1 Support for Mobility

One of the main goals of the Mockets middleware is the support for mobility. Therefore, one of the design guidelines for the middleware is the introduction of the *mocket* as a communication endpoint that can move from one host to another. A mocket can move when receiving either an application level event or a network one.

In the first case, the application explicitly commands the middleware to move the mocket endpoint. Note that the migration is completely transparent to the remote application (apart from a temporary increase in message arrival latency). The second case refers to a situation where a host moves from a network locality to a different one. The network layer notifies the Mockets middleware about the host migration, which then performs the migration of all the active mockets in the moved host, transparently to the application.

The importance of the application driven mobility is clear in mobile code systems. For instance, in the case of mobile agent applications, one of the main research areas deals with the problem of the bindings of the agent itself with its current context of execution. The Mockets middleware permits a mobile agent to easily move all its network connections. This solution is currently being tested in the NOMADS platform, which supports strong agent mobility.

The Mockets network driven migration is fundamental to permit applications to continue the execution even in presence of device mobility. In fact, the Mockets middleware detects the changes in the network layer address and consequently moves all the mockets without forcing the application to shut down and to reopen the network connections.

3.2 Communication Semantics

Mockets allows applications to establish message-based and connection-oriented communications and supports a wide range of message delivery services. Applications can exploit one or more delivery services by choosing orthogonally between reliable/unreliable and sequenced/unsequenced message delivery on a per-message basis.

The sequenced reliable delivery service provides semantics similar to TCP and incurs the same performance penalties. It is best suited to the delivery of important but time insensitive data such as critical notifications of application state change. The sequenced unreliable delivery service is best suited to convey time sensitive data such as multimedia information, as it provides the same communication semantics of the Real-Time Protocol (RTP). The reliable unsequenced message delivery service allows applications to transmit important but unrelated messages such as signalling information. Finally, unreliable unsequenced delivery service is useful for less important report messages.

Notice that the constraint on sequential delivery of messages is usually applied only to messages belonging to the same (sequenced) delivery service. However, a connection can optionally be configured to perform sequencing across both reliable and unreliable sequenced delivery services. In this case, sequenced messages will be delivered to the peer application after all previously sent sequenced messages, regardless of their reliability.

If an unreliable sequenced message with sequence number N is lost, when the next message with a sequence number greater than N arrives, the Mockets middleware waits for a small amount of time in case the first message was transmitted on a route that was slower. If the message with sequence number N is not received when the timeout expires, Mockets considers it lost and delivers the next message in the sequence.

3.3 Semantic Differentiation of Application-Level Traffic

Mockets supports the classification of messages into different group types. Applications can perform group operations on messages of a specific type, e.g., to enforce a maximum transmission bandwidth, to assign a specific lifetime, or a transmission priority value. For instance, an application can decide to either cancel all previously enqueued messages of a specific type, or replace them with a new message of the same type.

Message cancellation and replacement is a very useful feature in situations where applications are sending periodic updates and a new update invalidates previous ones. For instance, when using a reliable flow, the Mockets middleware will buffer messages until they have been successfully acknowledged by the remote endpoint. If the network is congested or the peer is unreachable, messages can accumulate. Using the message replacement feature, applications will be able to remove stale information that is still in the queue and replace it with up-to-date data. This reduces transmission of obsolete information and therefore minimizes network bandwidth consumption.

It is worth noticing that the message classification and replacement features implemented by Mockets have been especially designed to support applications that use multiple but interrelated data types such as MPEG. For instance, an MPEG video streaming application might use two different message types, one for key frames and a second for delta frames. Upon the generation of a new key frame, the application could choose to discard all enqueued MPEG frames by first cancelling all of the enqueued delta frames and then replacing all the enqueued key frames with the new one.

3.4 Network Conditions Monitoring

An important design guideline for the Mockets middleware is to monitor network conditions and to pass the gathered information up to the application level. In this way, applications can make informed decisions about how to tailor services according to both service logic and user preferences.

Applications can either directly interrogate the Mockets monitoring or request to be notified when a specific event occurs by registering callback functions. For instance, one of the events that the Mockets monitoring facility can notify applications about is peer unreachability, which is detected by a keep-alive mechanism that allows quick discovery of problems at the link and network layers.

Let us note that the adoption of the subscribe-notify paradigm is rather unusual in network programming, as historically transport protocols have always been designed to masquerade varying network conditions to applications. However, the need for a richer model of interaction between applications and transport protocols emerges also in other recently proposed transport protocols such as the Datagram Congestion Control Protocol (DCCP) (Kohler et al., 2003).

3.5 Advanced Low-Level Communication Features

The Mockets middleware provides advanced communication features to allow fine-grained performance tuning in the case of time-sensitive data exchange, e.g., multimedia and control applications. For this purpose, applications can choose several transmission parameters on a per-message basis, like message priority, maximum lifetime, and a timeout for the insertion of messages in the transmission/pending message queue.

Applications can reorder the transmission of messages by assigning them priority values. The Mockets middleware schedules messages with higher priority before lower ones. This priority differentiation provides applications with a low-latency message delivery service that can be used for important application status updates.

Applications can also assign a maximum lifetime to outgoing messages in order to automatically discard outdated information. The Mockets middleware enforces a timeout for the transmission of every message with an associated lifetime. If this timeout expires before the message is sent, then the message is silently discarded. On the other hand, if the timeout of reliable message expires after the message is transmitted but before an acknowledgement is received, the message is discarded and a notification of the message lifetime expiration is sent to the destination endpoint.

Finally, applications can set a time limit for the insertion of a message into the transmission queue. If the timeout expires, the message is not scheduled for transmission and an error is returned to inform the application. This feature allows applications to discard information with a short lifetime in case the transmission queue is full, minimizing latency in information delivery.

4 MOCKETS ARCHITECTURE

Mockets operates at the application layer on top of the traditional TCP/IP protocol suite. This design guideline supports portability and ease of integration and facilitates the deployment of Mockets in all available platforms and scenarios, regardless of the underlying hardware and operating system.

Figure 1 shows the architecture of the Mockets middleware. The main components supporting applications are the Session Management, Message Management, Connection Status Monitor, and Traffic Differentiation modules.

The Session Management module implements control operations such as session suspension and resumption. It performs the migration of a mocket communication endpoint upon application requests, by following the procedures described in section 5.1. It also interacts with the underlying operating system to detect changes in the network layer address of the host, which are triggered by device movements. In this case, the Session Management module reconfigures all the existing mocket endpoints to use the new address, transparently to the applications.

The Message Management module handles the delivery of application messages. In particular, it is in charge of dividing large messages into several packets and reassembling them before application delivery. It also guarantees the ordering of messages when required by the chosen delivery service.

The Connection Status Monitor receives information about connection status and network conditions from the underlying layer. It provides upper layers with monitoring functions by both permitting explicit queries from the application and performing notifications when a subscribed event occurs.

The Traffic Differentiation module is in charge of scheduling application messages for transmission by applying the specified traffic differentiation policies, such as message priority, delivery services, etc.

Underneath the components directly interacting with the application, the Transmitter and Receiver modules take care of message transmission and reception, by interfacing with the underlying network via UDP. The Receiver module continuously listens for incoming messages, dispatching data messages to the Message Management module, information on communication status and network conditions to the Connection Status Monitor, and control messages to the Session Management module. The Transmitter module performs message transmission operations, interacting with the Receiver module to implement ACK management.

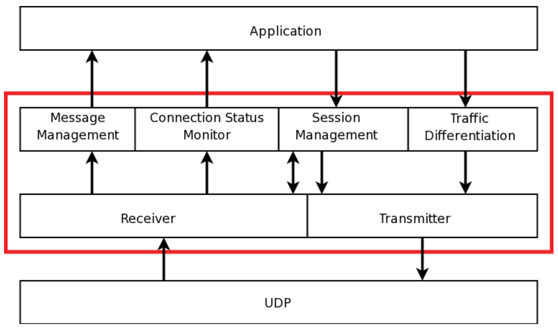

Figure 1: Architecture of the Mockets middleware.

4.1 The Mockets API

In order to facilitate the porting of existing applications, the Mockets middleware exports an object-oriented API designed to be as similar as possible to the traditional Java Sockets API. In particular, Mockets offers a core set of API that is exactly the same as connected Java datagram sockets. All the advanced Mockets functionalities are available through a novel set of API.

The core set of the API manages the *mocket*, the fundamental communication entity of the middleware that represents a connection endpoint and is identified by a network layer address/port couple. Similar to a socket, the mocket can be either active (the MessageMocket class) or passive (the MessageServerMocket class). The communication is established by connecting an active mocket to a passive one listening on a remote endpoint. The *connect* method of a MessageMocket is used for connection establishment. On the server side, the *accept* method processes incoming connection requests. *accept* returns an instance of MessageMocket which represents a new connection.

Applications send and retrieve messages by calling the *send* and *receive* methods respectively on the MessageMocket instance. Connection teardown is performed by calling the *close* method of the MessageMocket class on either side of the connection.

The advanced Mockets API permits applications to exploit all the advanced functionality of the Mockets middleware. In particular, the *getSender* method of the MessageMocket class selects the delivery service for messages. *getSender* returns an object that represents the selected flow and provides the *send* method for message transmission.

The *enableCrossSequencing* method enables cross sequencing on the current mocket.

The *replace* method permits applications to replace previously enqueued messages with a new message. Messages to be replaced are identified by a specific message tag. Applications can also use the *cancel* method to cancel previous messages.

Applications can also retrieve information on the current status of the communication with the remote endpoint, via the *getStatistics* method. The statistics reported are the number of bytes and packets sent and received, the number of packets retransmitted, and the number of discarded packets. If these parameters fall below the desired QoS level, applications can adapt their behavior according to the network conditions and user preferences. For instance, in the case of heavy packet loss, applications may choose to downscale the data stream.

Applications use the *subscribe* method of a MessageMocket instance to register for a specific event among those supported by the Mockets middleware. In particular, applications provide Mockets with callback functions, which are used by the framework to notify the applications when the registered events occur.

Finally, applications can suspend the operation of a Mocket endpoint and retrieve its serialized state via the static *suspend* method of the MessageMocket class. A mocket can then be resumed by calling the static *resume* method of the MessageMocket class. Mockets only provides mechanisms for connection suspension/ resumption and serialization of a communication endpoint. The middleware does not enforce any security policy on these operations and leaves the task of transferring the mocket status to the new host to the application.

5 IMPLEMENTATION

With implementations in both Java, C++, and C# programming languages, Mockets achieves significant portability, and is available for most existing operating systems and development platforms. This section describes the most significant implementation details of the Mockets middleware.

5.1 Endpoint Mobility

The mobility of a mocket is a distinguished feature that requires the middleware to perform several operations. When the migration procedure initiates, the mocket connection is suspended and the remote endpoint enters a standby state. The local endpoint is then migrated to a different host where it reconnects transparently with the remote endpoint. The mocket on the remote endpoint is notified of the address change in the communication endpoint and resumes normal operations.

Figure 2 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 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.

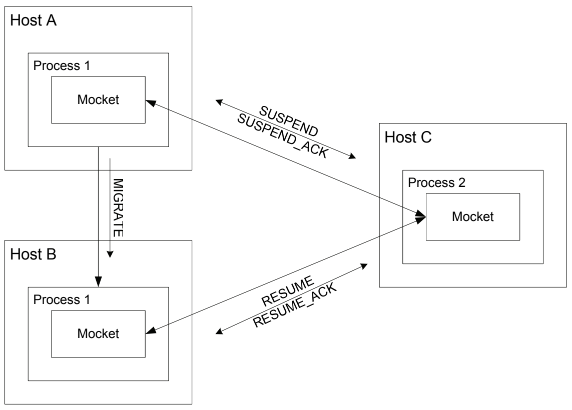

Figure 2: Migration of a Mocket endpoint.

Endpoint mobility is currently supported only by the Java implementation of the Mockets middleware, which complements the Agile Computing middleware and the NOMADS mobile agent system. In this environment, migration is realized by serializing all the object instances representing the current state of the mocket connection in use by the local endpoint and transferring them over a network link.

5.2 Message Management

The Transmitter, the Receiver, and the Message Management are the main components of the Mockets middleware in charge of transmitting/receiving application messages.

The Message Management is in charge of fragmenting a message into several packets when needed. Resulting packets are enqueued into a pending packet queue. The packet insertion algorithm uses a dynamic priority scheme to favor the transmission of high-priority packets while preventing starvation of low-priority packets. If the remote window allows the transmission of a new packet, the transmitter then retrieves the first packet from the pending packet queue and transmits it to the remote endpoint via UDP packets.

To support reliability, after the transmission the Transmitter puts reliable and control packets into the appropriate unacknowledged packet queue from where they will be periodically retransmitted until they are acknowledged. For efficiency reasons, we have implemented the unacknowledged packet queues as data structures composed by two double-linked lists, one sorted by the retransmission timeout (for efficient retransmission) and one sorted by the sequence number (for efficient processing of acknowledgements). The Transmitter also handles acknowledgements and sends keep-alive messages in case of inactivity.

The Receiver component continuously listens for incoming packets on the UDP socket associated with the mocket. It reports peer unreachability problems to the application, handles acknowledgements and control messages (e.g., cancelled packets information), and dispatches incoming messages according to their reliability and sequencing values.

Unsequenced packets are passed to the Message Management module for the defragmentation (if needed) and for their final arrival into the received data queue.

Sequenced messages are enqueued into one of three sequenced packet queues (one for control messages, one for reliable sequenced packets, and one for unreliable sequenced packets). After each insertion into the sequenced packet queue, the Message Management module examines the queue and delivers all messages whose sequencing requirements are satisfied. In the case of unreliable sequenced messages, the Packet Processor also applies a timeout to stop waiting for missing messages.

5.3 Mockets Transport Protocol

The Mockets transport protocol relies on the exchange of UDP packets between endpoints. The connection establishment procedure of the Mockets transport protocol is based on a 4-way handshake like SCTP. The exchange of a cryptographically-protected cookie between peers makes the protocol resistant to SYN flooding attacks (Stewart and Xie, 2001).

To allow the multiplexing of data and metadata on UDP packets transmitted between two mocket endpoints, the Mockets transport protocol divides packets into chunks. Every chunk is identified by a unique type identifier and has a specific purpose. For example, data chunks carry application level data, while SACK chunks contain acknowledgement information.

Mockets performs selective ACKs. This acknowledgment strategy provides significant improvements in throughput when compared with the traditional cumulative acknowledgement scheme of TCP. SACK is done mainly via piggybacking, but in case no piggybacking can be done, a message containing SACK information is sent periodically or when duplicate packets are received (indicating that previously sent SACK information was lost).

Message cancellation and replacement is implemented by removing the corresponding packets from the pending packet queue and the unacknowledged packet queue. A control message is also sent to the peer endpoint about the cancellation and/or replacement of messages. To avoid stacking of control messages related to message cancellation and replacement, Mockets checks if there is already such a control message in the pending packet queue, which might occur because the peer is temporarily unreachable. In this case the old control message is replaced by a new one containing information about the whole set of cancelled messages.

Mockets allows applications to take full control of most of the internal behaviors (e.g. changing the default timeouts) of the transport layer. For instance, applications can change MTU, default connection timeout, pending packet queue size, keepalive timeout, maximum window size, and receive timeout.

6 EXPERIMENTAL RESULTS

We have measured the performance of the Mockets middleware in several scenarios with different working conditions. This section reports two separate experiments that compare Mockets-based with TCP-based message delivery.

The first experiment measured the raw throughput of the Mockets reliable sequenced delivery service compared to TCP sockets. The testbed is an 802.11b wireless environment where the wireless cards are configured to use an unused channel to ensure that there is no interference or other traffic on the wireless link. Table 1 shows the average time to transfer 2 MB of data using both Mockets and TCP Sockets. The results show that Mockets performs better than TCP in raw transfers of data.

The second experiment shows the benefit of the message replacement capability of the Mockets middleware in the presence of an unreliable network. The experimental setup is composed of two laptop computers interconnected via a wired network through a third computer running the NISTNet software (Nist NET). NISTNet provides control over parameters such as latency and packet loss and can simulate an unreliable link with periodic loss in network connectivity. To simulate disconnections, NISTNet is configured to drop all packets for a series of exponentially distributed random intervals of time with an average of 1 second. The intervals themselves are separated by an exponentially distributed random length of time with an average of 20 seconds. The client application generates an update message at a frequency of 1 Hz and transmits it to the server application.

Table 1: Throughput comparison over 802.11b.

	Mean Transfer Time (ms)	Standard Deviation (ms)
TCP Sockets	2151	121
Mockets	1836	80

Table 2 shows the average and worst-case latency of messages in the case of Mockets- and TCP- based communication. The results show that Mockets outperforms TCP Sockets. This is due to the fact that in TCP update messages are buffered on the client side and retransmitted when connectivity is restored. In Mockets, instead, update messages replaces previously enqueued messages. Old and outdated messages are simply discarded, thus reducing the consumption of network and computational resources on the both the client and server hosts.

Table 2: Latency Comparison.

	Average Latency (ms)	Maximum Latency (ms)
TCP Sockets	132.90	6228
Mockets	13.44	922

7 RELATED WORK

Many research efforts investigate how to allow the realization of efficient and robust distributed applications in the wireless Internet. In particular, there are solutions at both network and application layers with different trade-offs between performance and flexibility.

Researchers have proposed to modify the behavior of TCP in order to improve its performance in the new scenario (Tian et al., 2005). In fact, TCP is still the most widely adopted transport protocol in the wireless Internet and is also often proposed for time-sensitive applications such as multimedia streaming (Wong et al., 2005). However, the performance of TCP is severely affected by terminal mobility and lossy channels (Bakshi et al., 1997) (Altman et al., 2000) (Abouzeid et al., 2003). In fact, TCP interprets every packet loss as a symptom of network congestion and therefore reduces the congestion window size in the sending host, with very slow throughput recovery. Recent proposals addressed this problem by employing cross-layer techniques to get feedback on the network conditions and using this information to tune the congestion control procedure (Singh and Iyer, 2002).

The proposed modifications allow a limited performance improvement for TCP-based communications in the wireless Internet without changing existing applications. However, let us note that the modification of a communication protocol raises backward compatibility problems with existing protocol implementations, which limits the benefits of this approach.

Researchers have also proposed the Stream Control Transport Protocol (SCTP) as an alternative to TCP on the wireless Internet (Stewart and Xie, 2001). SCTP provides applications with several message delivery semantics (reliable/best-effort and sequential/out-of-order). SCTP also allows applications to define conditions (typically a time limit) upon which reliable messages are considered stale and thus discarded. In addition, SCTP can maintain multiple streams of messages inside a single connection, mitigating the head of line blocking problem of TCP (Atiquzzaman, 2003). Finally, the recent introduction of dynamic address reconfiguration for SCTP connection endpoints allows partial support of device mobility. However, SCTP still suffers from the same congestion-related performance problems of TCP and does not provide applications with any feedback on network conditions.

Other research approaches developed novel communications middlewares on top of existing network stacks. I-TCP (Bakre and Badrinath, 1995), Mobile-TCP (Haas, 1995), and the Remote Sockets Architecture (Schlager et al., 2001) address both the performance and the mobility issues in TCP by proposing proxy-based architectures. In these proposals, connections are routed through proxies deployed at the edge of the wireless and wired portion of the network. This improves the performance of communications on the wireless portion of the communications, but it requires the deployment of dedicated proxies with a modified network stack.

Other proposals focus on providing a network-aware programming model to applications but do not offer support for user/terminal mobility (Sun et al., 2003), or supporting mobile computing applications by adding endpoint mobility functionality to traditional communication protocols (Snoeren and Balakrishnan, 2000) (Hsieh et al., 2004).

Mockets goes beyond the above mentioned proposals by providing applications with a wide range of communication semantics and a richer programming model, which are better suited to the wireless Internet. In addition, Mockets implements mechanisms to effectively support user/terminal mobility and allows applications to monitor current network conditions. Finally, Mockets does not require the deployment of dedicated devices with a modified network stack or any other entity which breaks the traditional end-to-end communication semantics.

8 CONCLUSIONS AND FUTURE WORK

The Mockets middleware is a comprehensive solution for the development of robust and efficient distributed applications suited to the wireless Internet scenario. In particular, Mockets-based applications can cope with packet losses and network disconnections and can handle the mobility of terminals and users. The first experimental results show that the Mockets middleware performs better than TCP in wireless networking environments.

Although the results are encouraging, we are working to improve the performance and the features of the Mockets middleware. For instance, we are evaluating a new architecture for advanced I/O operations and internal buffer management and the adoption of cross layer techniques in order to provide applications with more information on network conditions.

We are also planning to integrate Mockets with the Agile Computing Middleware to take advantage of proactive resource manipulation and with the KAoS policy management system to allow policy-based control over utilization of network resources.

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, by the Air Force Research Laboratory under grant FA8750-06-2-0064, and by the Italian MIUR in the framework of the Project “MOMA: a middleware approach to MObile MultimodAl web services”.

REFERENCES

Abouzeid, A., Roy, S., Azizoglu, M., 2003. Comprehensive Performance Analysis of a TCP Session Over a Wireless Fading Link With Queueing, *IEEE Transactions on Wireless Communications*, Vol. 2, N. 2, pp. 344-356, March 2003.

Altman, E., Avrachenkov, K., Barakat, C., 2000. TCP in presence of bursty losses, *ACM SIGMETRICS Performance Evaluation Review (Special issue on proceedings of ACM SIGMETRICS 2000)*, Vol. 28, N. 1, pp. 124-133, June 2000.

Atiquzzaman, M., Ivancic, W., 2003. Evaluation of SCTP Multistreaming over Satellite Links, in: *Proceedings of 12th International Conference on Computer Communications and Networks*, Dallas, TX, USA, October 2003.

Bakre, A., Badrinath, B., 1995. I-TCP: Indirect TCP for Mobile Hosts, in: *Proc. of 15th IEEE International Conference on Distributed Computing Systems (ICDCS '95)*.

Bakshi, B., Krishna, P., Vaidya, N., Pradhan, D., 1997. Improving Performance of TCP over Wireless Networks, in: *Proceedings of 17th International Conference on Distributed Computing Systems,* Baltimore, MD, USA, May 1997.

Chang, F., Karamcheti, V., 2001. A Framework for Automatic Adaptation of Tunable Distributed Applications, *Cluster Computing*, Vol. 4, N. 1, pp. 49-62, March 2001.

Cheng, L., Marsic, I, 2002. Piecewise Network Awareness Service for Wireless/Mobile Pervasive Computing, *Mobile Networks and Applications*, Vol. 7, N. 4, pp. 269-278, August 2002.

Fu, X., Hogrefe, D., Le, D., 2006. A Review of Mobility Support Paradigms for the Internet, *IEEE Communications Surveys and Tutorials*, Vol. 8, N. 1, pp. 38-51, 1st Quarter 2006.

Gross, T., Steenkiste, P., Subhlok, J., 1999. Adaptive Distributed Applications on Heterogeneous Networks, in: *Proceedings of the 8th Heterogeneous Computing Workshop*.

Haas, Z., 1995. "Mobile-TCP: An Asymmetric Transport Protocol Design for Mobile Systems", in: *Proc. of 3rd International Workshop on Mobile Multimedia Communications*.

Hsieh, H., Kim, K., Sivakumar, R., 2004. An End-to-End Approach for Transparent Mobility across Heterogeneous Wireless Networks, *Mobile Networks and Applications* Vol. 9, N. 4, pp. 363–378, August 2004.

Kim, M., and Noble, B., 2001. Mobile Network Estimation in: *Proceedings of the 7th annual international conference on Mobile computing and networking (MOBICOM 2001)*, Rome, Italy.

Kohler, E., Handley, M., Floyd, S., 2003. Designing DCCP: Congestion Control Without Reliability. *ICIR Technical Report.*

Nist NET. The Nist NET network emulator. Available at: http://snad.ncsl.nist.gov/nistnet/

Schlager, M., Rathke, B., Bodenstein, S., Wolisz, A., 2001. "Advocating a Remote Socket Architecture for Internet Access Using Wireless LANs", *Mobile Networks and Applications*, Vol. 6, N. 1, pp. 23-42, Jan./Feb. 2001.

Singh, A., Iyer, S., 2002. ATCP: Improving TCP performance over mobile wireless environments, in: *Proceedings of 4th IEEE Conference on Mobile and Wireless Communications Networks*, Stockholm, Sweden.

Snoeren, A., Balakrishnan, H., 2000. An End-to-End Approach to Host Mobility, in: *Proceedings of 6th ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '00)*, Boston, MA, USA.

Stallings, W., 2005. *Wireless Communications & Networks*, Prentice-Hall. 2nd edition.

Stewart, R., Xie, Q., 2001. *Stream Control Transmission Protocol (SCTP)*. Addison-Wesley.

Sun, J., Tenhunen, J., Sauvola, J., 2003. CME: A Middleware Architecture for Network-Aware Adaptive Applications, in: *Proceedings 14th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC 2003)*, Beijing, China.

Suri, N., Tortonesi, M., Arguedas, M., Breedy, M., Carvalho, M., Winkler, R., 2005. "Mockets: A Comprehensive Application-Level Communications Library", in: *Proc of Military Communications Conference (MilCom 2005)*, Atlantic City, NJ, USA.

Tian, Y., Xu, K., Ansaru, N., 2005. TCP in Wireless Environments: Problems and Solutions, in: *IEEE Radio Communications*, Vol. 43, N. 3, pp. S32-S47, March 2005.

Wong, C., Tang, C., Fung, W., Chan, G., 2005. Using TCP for Video Streaming Over Wireless Channel, in: *Proc. of 2nd International Conference on Quality of Service in Heterogeneous Wired/Wireless Networks (QShine'05)*, Orlando, FL, USA.