A DISSERTATION

SUBMITTED TO

THE DEPARTMENT OF COMPUTER SCIENCE,

AT THE UNIVERSITY OF CAPE TOWN

IN PARTIAL FULLFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTERS OF INFORMATION TECHNOLOGY

By

Justin Walsh

September 2003

Supervised by

Professor Ken MacGregor

Implementing the Reliable Data Protocol on a Java enabled Mobile Device

Abstract

The author examines the architecture and discusses the implementation and problems encountered during implementation of the Reliable Data Protocol on a Java enabled mobile. A preliminary evaluation of the protocol implementation is presented.

Acknowledgements

I would like to thank the the following people for providing valuable feedback during the drafting of this paper: Heinrich Brink, Andrew Roos, Noel Grandin and James Gain. Thanks also to Professor Ken MacGregor for his guidance.

Contents

1. Introduction

RDP and J2ME – Motivation and aims

The Reliable Data Protocol is a packet based, reliable data protocol that allows for sequenced or unsequenced delivery of packets as described in RFC908 and RFC1152 [1, 2]. Although somewhat similar to the Transmission Control Protocol (TCP) [3], it is intended to be a simpler protocol efficient in environments where there may be long transmission delays and loss or non-sequential delivery of message segments [1].

Although the Reliable Data Protocol has been implemented under the BSD 4.2 and 4.3 operating system [4] and Java 2 Standard Edition platform [15] to the best of the author's knowledge, no attempt has been made to implement the protocol on the Java Micro Edition (J2ME) platform. Given the type of device which this platform typically runs on, network connectivity is typically characterised by sporadic high bit error rates and intermittent connectivity. Additionally many of the applications and services that have been designed to run on such devices (such as the short message service) fit well within the model of a packet based transport protocol, as opposed to a stream based protocol provided by more complex protocols such as TCP.

The properties of the reliable data protocol (above) therefore seem well suited to an implementation in this environment.

Java is an interpreted language, running within the confines of a virtual machine meeting a standard set of specifications. Part of this specification includes automatic garbage collection: the allocation and deallocation of memory space used by objects is managed by the runtime environment during the execution of the code. Code written to the language specification is therefore simpler to write (no need to manually dereference objects and allocate memory) and portable across a wide variety of devices that provide a suitable runtime environment. The J2ME/MIDP platform specifically has been implemented on over 40 types of hand-held devices. Additionally, emulators are provided free of charge allowing developers free access to an emulated runtime environment without having to incur the expense of purchasing a compliant device.

Java is an object oriented language allowing for the implementation of the protocol using modern familiar object oriented patterns and paradigms. Code written in such a manner is therefore easily readable, extendible and flexible; an ideal candidate for a research vehicle.

Java and the J2ME platform are therefore ideal candidates for the implementation of the Reliable Data Protocol. The aim of this paper is therefore to implement and detail the implementation of the Reliable Data Protocol on the MIDP2 profile of the J2ME platform as a research vehicle. Furthermore, a preliminary evaluation of the protocol is made.

It must be stressed that the intention of this paper was not to produce a production-ready implementation.

Overview

Chapter 2 of the paper covers in more detail the specifics of RDP and introduces the Java 2 Micro Edition Platform, the Connected Limited Device Configuration (CLDC) and the Mobile Information Device Profile (MIDP), the target profile for the implementation.

Chapter 3 deals with the implementation itself, firstly from an architectural perspective and then details problems encountered during the implementation.

Chapter 4 provides an evaluation of the profile, with particular attention being paid to the memory footprint of the application and throughput.

Source code and the documentation for the source code (in Javadoc format) of the author's implementation accompany this paper in electronic format.

2. Background

RDP

RDP, as specified in Internet Request for Comments(RFC) 908 [1] and RFC1151, is a connection oriented transport protocol that differs from the well known Transmission Control Protocol TCP [3] in that it is packet based as opposed to stream based. This simplifies data handling somewhat, keeping data boundaries and allowing out of sequence acknowledgement and delivery of packets although applications that make use of the protocol need to be able to handle data in this format as opposed to the stream format embraced by TCP.

The intention of the protocol was to develop a simple transport protocol for efficient bulk data transfer, such as loading/dumping and remote debugging, to be used on networks with moderate loss rates.[1,4]. It currently has no congestion control specified.

RDP as a protocol fits into the layered Internet protocol environment, relying on lower layers for the provision of basic inter-network services such as addressing and de-multiplexing (figure 1). It makes no assumptions about the exact nature of such lower layers, and is therefore portable across different networking environments.

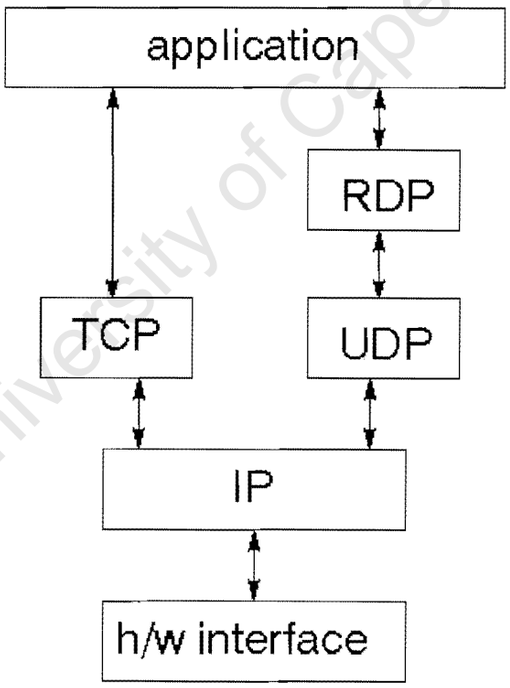


Figure 1 - The relationship between RDP and other layers

Data from higher layers, typically applications, is sent and received in contiguous segments or packets. Data segments passed to RDP are assigned sequential 32-bit sequence numbers and passed to the network (lower layers) from where they are delivered to the receiving host. Arriving segments are acknowledged (ACK) by the receiving host by sending the sequence number of the highest numbered segments received in sequence. Unless the receiving application has specified that the segments are to be delivered in sequence, segments are delivered to the application as they arrive.

Although figure 1 places the RDP protocol between the transport layer (UDP) and the application layer, it is conceivable that RDP could interface directly with IP. The implementation of RDP on a MIDP2 device necessitated the interface between RDP and UDP. This is detailed further in the section 'Architectural Goals and Constraints – Interfacing with the Network Layer'.

Flow control is managed as follows: the module in the transmitting host sends segments until it reaches the connections segment limit specified by the receiving process (as specified during connection creation process). Once this limit is reached, the transmitting RDP module may only send a new segment for each segment acknowledged (ACK or EACK). Attempting to send a segment once this limit is reached, results in an error.

RDP also employs the concept of a shifting 'window of acceptable sequence' numbers. Incoming segments are regarded as acceptable if their sequence numbers fall within the range of twice the allowed maximum number of outstanding segments. The window is shifted if the segments' sequence number that is received is equal to the left-hand edge (i.e. the next sequence number expected) of the window. Segments received out of order and acknowledged with and EACK (see below) do not therefore shift the window on.

To reduce the unnecessary retransmission of data RDP employs the concept of extended acknowledgements (EACK). A segment received out of order, but within the acceptable segment frame is acknowledged in an extended fashion. The sending peer (that receives the EACK) records the acknowledgement of the segment, removing it from the retransmission queue, but the acceptable segment frame is not advanced.

Reliability is provided by the retransmission of segments that have not been acknowledged by the receiving host within a given time frame. The specification makes no attempt to specify how this time-out should be set although the derivation of this value should take into account the round-trip time of a segment.

It is worthwhile to note that SYN segments sent during the establishment of a connection and acknowledgement segments not containing data are not sent reliably. This can result in the failure to establish a connection if a packet is lost during the three-way 'handshake' whilst setting up a connection. The specification is silent on how this issue should be handled. One implementation addresses this issue by sending the SYN segment reliably [15].

A 16 bit TCP checksum is used to provide some form of data integrity between the sending and receiving host. Segments that fail the checksum calculation are discarded and not acknowledged.

The protocol exposes a simple interface to clients that allows the opening and closing of a connection, the sending and receiving of data and the request for a status of a connection. The manner in which the protocol responds to these events (and other events such as the arrival of segments, the retransmission of segments, and the firing of the time-out event) is dictated by the state in which the protocol finds itself (Figure 2). Connections start in the CLOSED state and either use an 'active open', or 'passive open' request to reach the OPEN state. The connection established is a full duplex connection and either side of the connection may request the closing of the connection in which case the connection transitions to the CLOSE-WAIT state and ultimately back to the CLOSED state.

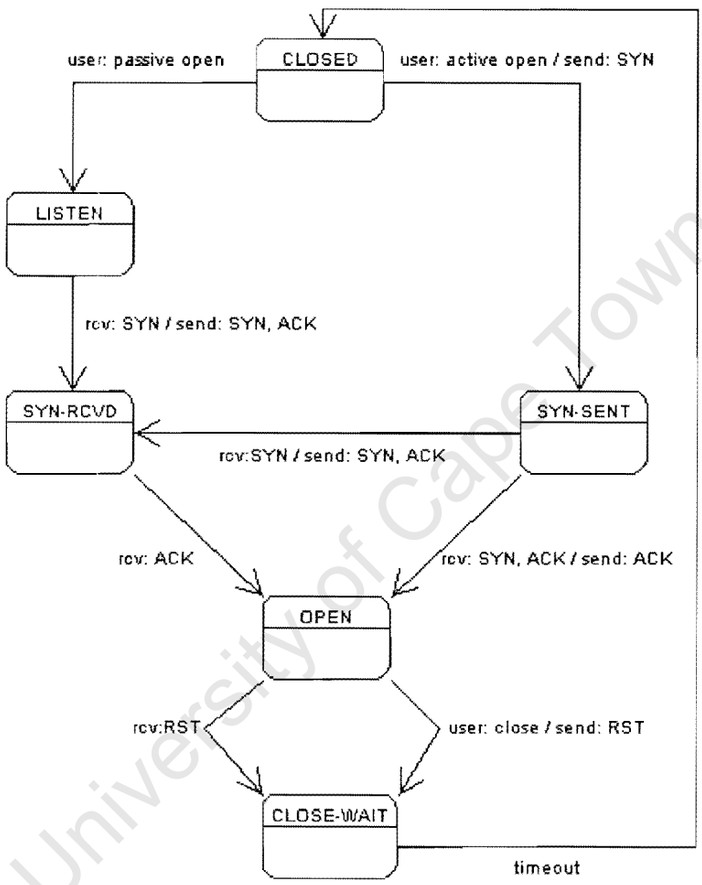


Figure 2 - Protocol State Diagram

When a RDP connection is closed, the protocol makes no further attempt to deliver packets that have been passed to the implementation, but not yet acknowledged by the remote host. These closing semantics are somewhat different to those of the TCP specification [3].

J2ME, CLDC, and MIDP2

The Java 2 Micro Edition [5] is a collection of specifications that define a set of platforms each of which is suitable for a subset of the total collection of consumer devices that fall within the J2ME's scope [6]. Each of the specifications is optimised for the memory, processing power and I/O capabilities of the target devices. The scope of J2ME is intended to include those devices with limited processing power and memory; typically those devices that that are too small to run the Java 2 Standard Edition (J2SE) platform, often hand held and portable devices.

The capabilities of devices of a "Micro-Edition" nature vary considerably [7]. To support the range of devices the concept of a "configuration" is embodied. Broadly speaking, a configuration is a specification that defines the software environment for a range of devices defined by a set of characteristics. These characteristics include the amount of memory available, the processor type and speed, and the type of network connection available to the device.

J2ME currently defines two such configurations, the Connected Limited Device Configuration (CLDC) and the Connected Device Configuration (CDC). Each configuration defines the core Java classes and minimum requirements of the Java Virtual Machine for the platform.

CLDC is aimed at the low end of the consumer electronic range designed for devices with intermittent network connections, slow processors and limited memory. These devices typically have either 16 or 32 bit CPUs and a minimum of:

- 128 kilobytes of memory for running the Java Virtual Machine and CLDC libraries (non-volatile memory).
- 32 kilobytes of memory available during application runtime for allocation (volatile memory).

CDC addresses the needs of those devices that fall between CLDC and desktop systems running J2SE, often with persistent network connectivity and therefore falls beyond the scope of this discussion.

A profile complements a configuration, providing libraries and a minimum set of hardware requirements to enable applications written to a specific API to run on a particular type of device. Currently several profiles exist, including the Mobile Information Device Profile (MIDP), the PDA Profile and the Foundation Profile. The relationship between profiles configurations and the host operating system is shown in Figure 3.

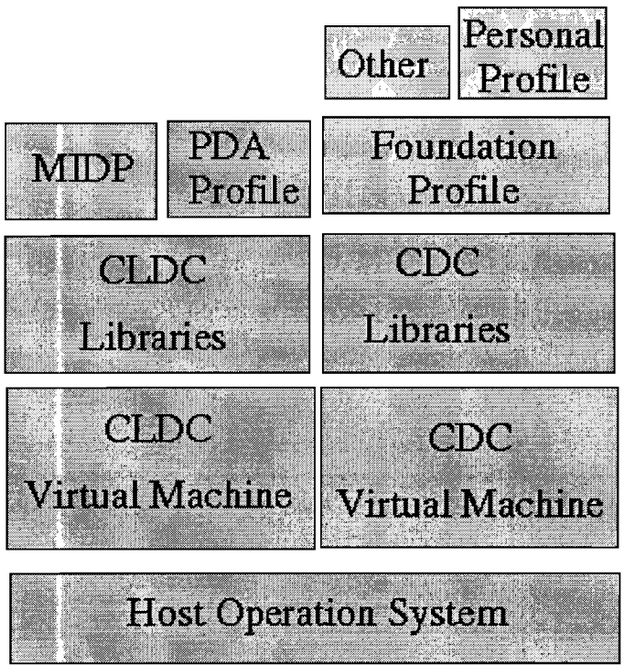


Figure 3 - J2ME Platform Architecture

The MID Profile [8] adds several features to those offered by the CLDC configuration, including support for networking. Version 2 of the MID Profile, MIDP2 [9] complements this with support for HTTPS, sockets and Datagrams. This profile is therefore intended for the vertical slice of the mobile market, including PDAs, cell phones, and two way pagers.

Java applications that run on MIDP devices are known as MIDlets. MIDlets run in an execution environment within a Java Virtual Machine (VM). The virtual machine provides well defined life-cycle method, some of which a MIDlet must implement. These methods provide life-cycle notification to the MIDlet. The author's implementation of RDP therefore runs within the confines of a MIDlet governed by the application management software.

The Mobile Information Device Profile combined with the Connected Limited Device Configuration is the Java runtime environment for today's mobile information devices [11].

The remainder of this paper therefore deals with the implementation of RDP on a MIDP2 enabled device.

3. Architecture and Implementation

Architectural goals and constraints

Although the RDP specification does detail one possible sequence for the handling of events, it does not intend to dictate a particular implementation. That said, any actual implementation should vary from the specification only in detail and not significantly in substance [1]. This gives the implementer considerable flexibility in the design and implementation of the code.

The chosen design and implementation has a direct bearing on the performance of the system. It is therefore worth considering the aims and architectural goals that were taken into cognisance when designing this implementation.

Of primary importance was the implementation of the core functionality of RDP. The specification clearly distinguishes between core functionality and functionality that could be omitted in the case of minimal implementation of the protocol. Optional features (such as the extended acknowledgement of packets) were implemented at the author's discretion.

Equally important was the manner of implementation. As the implementation was written in an object oriented language, it was decided to implement the architecture in an object oriented paradigm. This allowed for an implementation that was clean, readable, maintainable and extendible.

The implementation was intended for a specific platform, that being the J2ME's MIDP2 platform. This imposed a number of constraints on the design and implementation of the code. Memory, both program memory for the installation of an application, and heap memory for the running of an application are particularly scarce resources in the MIDP2 environment [11]. As opposed to common desktop applications which require and use megabytes of memory, program memory may be limited to as little as 30KB, as in the recently released Nokia 3410 mobile phone. In the extreme cases, heap memory may be as little as 20KB.

Given the stringent size requirements with regard to both program and heap memory allocations, one of the implementation goals would be to produce the smallest implementation both in application size and runtime footprint which satisfied the core requirements of RDP. Application size can be trimmed down within certain limits by making use of a byte code obfuscator[12]. Runtime memory use however, is more of a challenge, as realisation of the problem only occurs at application runtime as opposed to design time, with a resultant out of memory error.

Often, the consequence of a pure object oriented design is object overuse. This is particularly detrimental in environments where runtime memory is limited, such as the

MIDP2 environment. In a Java runtime environment the allocated memory space of dereferenced objects are automatically reclaimed by the garbage collector. The running of such a garbage collector itself consumes processing time and may totally (depending on the algorithm used) block execution of the code. As the rate of object allocation increases, a proportionate increase in the amount of time spent garbage collecting dereferenced objects is noted, and the execution of code slows. This allocation-dereference-garbage collection cycle can continue *ad infinitum* provided that the memory allocation of non-dereferenced objects does not exceed the limit of the heap size, in which case the application fails with a `java.lang.OutOfMemoryError`. A detailed discussion of garbage collection and the algorithms employed is beyond the scope of this paper.

Object overuse, meaning the excessive creation (and possibly dereferencing) of objects created during the runtime execution of code therefore has two consequences in an environment with limited runtime memory. Firstly the code may appear to run slowly as a large amount of time is spent garbage collecting, and secondly, if the run time's heap limit is exceeded, the application fails. Authors of such code may be tempted to refactor their code by making use of non-object oriented design patterns, such as long switch-case statements. In the interests of object oriented design and implementation, the author has chosen otherwise and made use of widely recognised design patterns, such as the singleton patten [13] to tackle these issues.

It must be noted that the primary goal of the implementation was not to write a protocol that transferred data as swiftly as possible. Implementations of such a nature, although not written for the MIDP2 platform, have already been attempted [4].

Interfacing with the network layer

Previous implementations of the Reliable Data Protocol (RDP) have interfaced directly with the Internet Protocol (IP) as the network protocol [4,14]. IP identifies incoming packets and passes them to the RDP layer. Likewise data passed to RDP is sent via the IP layer from where it is routed to the receiving host.

Unfortunately the targeted runtime environment (MIDP2) does not allow direct access to the IP layer (or even the TCP layer). Part of the reason for this is that because the Connected Limited Device Configuration (CLDC) is a specification (although a reference implementation is provided), no assumptions are made about the host's hardware and software configuration. It is therefore conceivable that many mobile devices which implement the specification do not implement a TCP/IP protocol stack. Instead the CLDC specification mandates the support for a higher level of abstraction from the transport layer, stream based sockets and datagrams.

Sending data via a stream based socket is a reliable, connection oriented method to transmit a continuous stream of data with no provision for the marking of record boundaries. Although these are useful features, the very nature of RDP as a packet based protocol that *provides* a

reliable transport service negates the use of socket based streams as an underlying transport mechanism.

The RDP implementation was therefore implemented using the CLDC's support for datagram connections and datagrams.

Architectural details

Client Interface

The RDP implementation exposes the following interface to clients via the `uct.rdp.RDPConnection` class (Table 1).

Method	*Description*
openActive	Static factory method to create an active connection. The parameters include the remote host, remote port and a uct.rdp.`DataListener`. Successful calls to this method returns a `uct.rdp.RDPConnection`.
openPassive	Static factory method to create a passive RDP connection. The parameters include the local port to listen on and a `uct.rdp.DataListener`
send	Send data over the connection. An array of bytes is taken as a parameter. The data is sent as a single packet.
status	Retrieve the status of the `uct.rdp.RDPConnection`.
close	Close the connection.

Table 1 - Client interface

The semantics of the client interface differ slightly from that of RFC 908.

RFC 908 [1] specifies that the local port (end point address) can be specified when opening a active connection. The CLDC syntax for the creation of a active `javax.microedition.io.DatagramConnection` does not allow the client to specify the local port. The local port when opening an active connection is therefore left to the CLDC implementation.

This presents somewhat of a problem during the completion of the RDP headers, as the header needs to contain the local end point address to allow for multiple connections on the same host. Fortunately the MIDP2 specification introduces the interface `javax.microedition.io.UDPDatagramConnection`, a datagram connection that is aware of it's local end point address which is used in this implementation.

The major difference in the client interface is the omission of the receive method. This method is intended to pass data from the RDP implementation to the client. This implies that data is buffered in the RDP connection, a prerequisite to allow sequenced delivery of packets to the client in cases where packets are received from the underlying transport protocol in a non-sequenced order and are acknowledged out of sequence (EACK). RDP's flow control scheme does not however allow the receiver to close the sender's window.

Apart from the increased burden on runtime memory required to buffer client data, an additional responsibility is placed on the client of the connection to receive data from the connection (or in the case of no data being present, attempt a receive of data) in a timely manner. If this responsibility is overlooked, the buffer could theoretically grow infinitely, or for practical purposes, until some resource is exhausted. The definition of timely is a very loose definition and further complicates matters. Implementations of RDP that choose to implement a receive buffer would conceivably need to agree in well defined manner, the time-frame consequences of over run buffers. This is not dealt with in RFC 908. RFC 1152 acknowledges the problem and suggests that a solution to the problem (as suggested by the members of the End-2-End research group) would be to only acknowledge a segment after it has been delivered to the application[2].

Given the memory constraints of the environment and complexity involved in implementing a receive buffer, the author decided to implement the protocols receive process in a 'push' manner, as opposed to the traditional 'pull' manner.

During connection creation (via the static factory methods) clients are required to pass as a parameter to the static factory method an object that implements the `uct.rdp.DataListener` interface. This interface defines a single method:

```
void arrive (byte[] data)
```

Data that is received by RDP is 'pushed' on to the listener object that was 'registered' during connection creation. Although this effectively negates the need for a receive buffer, there is an important trade-off to be made: data arriving out of sequence can not be delivered sequentially

The status method on the `uct.rdp.RDPConnection` object provides status information about the connection, including it's state in the form of a `uct.rdp.RDPStatus` object. This information is important for the client as although the factory methods on the `uct.rdp.RDPConnection` class return a reference to a connection, the connection may not be in a state to actively send data. This state (OPEN) is only reached once the three way handshake has been reached successfully. It is important to note that this implementation of the protocol always returns a value of 0(zero) for the 'number of segments received but not given to the user', for the reasons mentioned above.

Communications protocols such as TCP and RDP are event driven state protocols; the behaviour of the object, in this case the connection, depends on its state, and the connection is required to change it's state at run time based on the current state of the connection and the event. This makes the implementation of the protocol an ideal candidate for a behavioural pattern commonly known as the state pattern [13].

The events that the connection can respond to are grouped into 3 major types:

1) User requests (e.g. Open, Send, Close, Status)
2) Segment arrival events triggered when a packet arrives
3) Time-out events (Retransmission queue time-out and Close-Wait time-out)

The interface for the behaviour associated with a particular state is defined by the `uct.rdp.RDPState` abstract class. This class also provides default behaviour for many of the methods. Each concrete subclass of this class implements the behaviour specific to the state of the connection. A concrete subclass therefore exists for each of the allowable states of the RDP connection. A detailed definition of the connection states can be found on page 9 of RFC 908. The class diagram is illustrated in Figure 4.

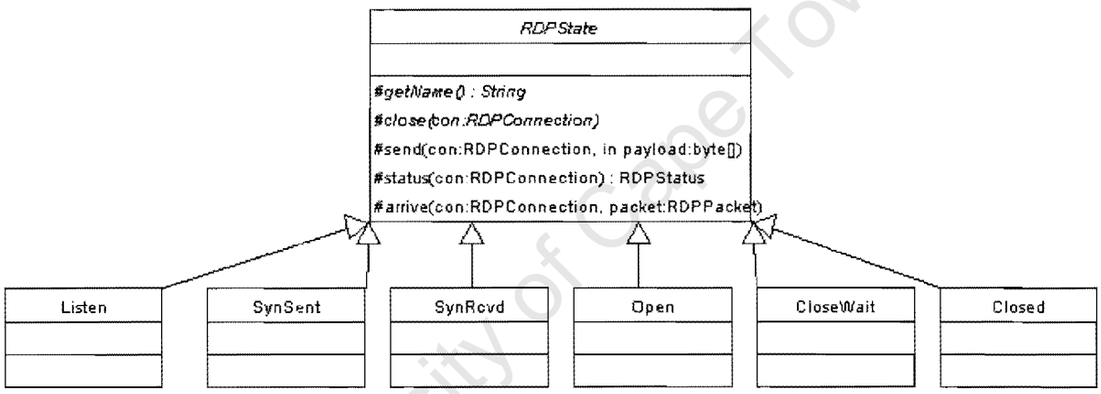


Figure 4 - Class heirachy of the concrete state subclasses

The `uct.rdp.RDPConnection` object defines the interface that is exposed to the client. The connection object is therefore responsible for delegating state specific requests in response to an event to the instance of the concrete state subclass that represents the connection's current state. The concrete subclasses are also responsible for the event based change in state of the connection object. For this reason, the connection object passes itself as a parameter to the state object handling the request.

Most of the event handling is delegated to the state subclasses by the connection instance. The exceptions are the open event and the CLOSE-WAIT time-out event. The open event,

generated when a client opens a connection in either an active or passive state, is handled by the connection class' factory method. The CLOSE-WAIT time-out event is also handled internally, merely changing the state of the connection to CLOSED.

One of the consequences of implementing the protocol using the state pattern is the excessive creation and deallocation of state objects as the connection transitions between states. This is most undesirable given the environment for which the protocol was implemented. Fortunately the state objects that are utilised have no instance variables, i.e. the state that they represent is entirely behavioural, making them candidates for the singleton pattern [13]. This ensures that the state classes only ever have one instance, and provides a single point of access for these classes. This point of access is the static instance method.

Support for sequenced delivery of segments

During connection establishment, the client has the choice of specifying the delivery mode – should data received by the connection be delivered to the client only in sequence, or whether out of sequence delivery is allowed. This parameter is valid for the life-time of the connection and cannot be altered[1].

The author's implementation of the protocol has chosen to implement the delivery of packets as a push operation onto the client (as opposed to a polling pull operation). This implies that no buffer of received data is kept. Sequenced delivery of packets is therefore simply implemented by discarding all out of sequence packets. Under these circumstances, sequenced delivery of packets occurs at the expense of the retransmission of packets arriving out of sequence.

Packet representation

Although the interface between the client and the underlying transport mechanism deals directly with byte arrays, segments of data passed from the client to the RDP and segments received from the underlying transport layer by the connection are represented internally by means of a class. This class is `uct.rdp.RDPPacket`.

The class contains convenience methods to parse an incoming byte array into a `RDPPacket`, and to convert a packet instance to a byte array for packaging in datagrams. The consequence of this is that a new object is created and discarded for each segment that is sent or is received. The benefit is a clearer and more explicit object model.

One alternative to this would have been to provide a class that defines a collection of methods used to modify and gather information from a byte array, thus leaving the byte array in its received form.

Exception handling

RFC 908 specifies that requests from the user always terminate with a return to the caller, with a possible error indication given as a character string. The author's implementation indicates erroneous conditions to the client by propagating an exception of type `uct.rdp.RDPException`. Exceptions of such a class represent application exceptions. Runtime exceptions and errors are propagated to the client unaltered.

Connections and connection records

Connection management as described in RFC 908 is intended to provide the ability to create multiple connection records for a given RDP implementation, each record bound to a distinct port on the host device. The management of these connection records would then become part of the single RDP implementation on the device.

The author's implementation of the protocol differs slightly from that described above. Although multiple connections on the same machine (but different ports) are supported, the concept of a single 'connection manager' is not realised. This is partly due to the operating platform that the implementation was targeted for. It is quite conceivable that multiple instances of the RDP Implementation could be running on a different virtual machine on the same device simultaneously. Although this is probably not the desired mode of operation, the design of this implementation provides this flexibility.

Retransmission queue and time-out

RDP provides for the retransmission of unacknowledged segments. This functionality is provided by the class `uct.rdp.RetransmissionQueue`. This class maintains a circular array of unacknowledged packets, and an internal timer that expires unacknowledged packets after a given number of milliseconds (re-transmission time-out period).

It is worth noting that the Java Language specification [16] does not provide any guarantee as to the exact time that a Thread will sleep for when calling the sleep(long milliseconds) method. The only guarantee is that the Thread will sleep for *at least* the number of milliseconds specified.

The derivation of the retransmission time-out (RTO) is not handled in the specification. To simplify the implementation of the protocol, the user is allowed to define the retransmission time-out period at application start-up. Once set, the period cannot be altered. This is not ideal. Overestimating the RTO can lead to excessive delays, underestimating can lead to excessive retransmission of packets[5].

Retransmission time-out period calculation and use has been detailed previously[5].

Problems encountered during implementation

Set up of send and receive sequence number variables

During active and passive connection creation, RFC 908 specifies that SND.MAX and RBUF.MAX should be filled in from the open parameters passed to RDP (page 21).

In the case of active connection creation, the SND.MAX variable is then sent as part of the SYN segment:

```
Send <SEQ=SND.ISS><MAX=SND.MAX><MAXBUF=RMAX.BUF><SYN>
```

and the state is set to SYN-SENT. On arrival of a SYN segment from a remote host, SND.MAX is set to SEG.MAX and RBUF.MAX is set to SEG.BMAX (page 26), effectively overwriting the values set up in the creation of the connection. This is incorrect and should read:

```
If SYN set
      Set   RCV.CUR = SEG.SEQ
            RCV.IRS = SEG.SEQ
            RCV.MAX = SEG.MAX
            SBUF.MAX = SEG.MAX
```

(... continued)

The SYN segment sent if the ACK bit is not set should read:

```
<SEQ=SND.ISS><ACK=RCV.CUR><MAX=SND.MAX><BUFMAX=RBUFMAX><SYN><ACK>
```

In the case of passive connection creation, the segment arrival event for the LISTEN state needs to be modified as follows (page 25):

```
If SYN set
      Set   RCV.CUR = SEG.SEQ
            RCV.IRS = SEG.SEQ
            RCV.MAX = SEG.MAX
            SBUF.MAX = SEG.MAX
      Send <SEQ=SND.ISS><ACK=RCV.CUR><MAX=SND.MAX><BUFMAX=RBUFMAX>
           <ACK><SYN>
```

(... continued)

Validation of maximum packet size

RDP is a packet based protocol in contrast to a stream based protocol such as TCP. The potential therefore exists to overburden the system with packets of a size that is unable to be

processed by the receiving host. Page 15 of the specification tackles the issue of packet sizes stating:

"The receiving end of each connection specifies the maximum segment size it will accept. Any attempt by the sender to transmit a larger segment is an error." "In addition, RDP will abort a connection with a RST segment if an incoming segment contains more data than the maximum acceptable segment size."

Although the specification is clear on the issue, no attempt is made to check the segment size in the pseudo-code. The problem is compounded by the fact that the maximum segment size must be at least large enough to accommodate the largest SYN segment. This is 26 bytes.

The maximum segment size acceptable to the remote host is exchanged during connection establishment as SEG.BMAX. The connection is therefore aware of this limit by the time it reaches the OPEN state and should validate the packet size against this limit when the application attempts to pass data to RDP. An alteration therefore needs to be made on page 23 of RFC908 to read:

```
Send Request
      OPEN STATE
            If (SEG.SIZE > SBUF.MAX)
                  Return 'Error - Segment size too large'
            Endif
            .... (continue as per normal)
```

Although the checking of segment size on arrival is also specified, and should be included in the pseudo-code for the sake of completeness, if both sides of the connection are proactive in checking the segment size *before* sending, this case should never occur.

The author's implementation of RDP does not make any attempt to combine packets. All outgoing data segments therefore have a minimum header length of 18 bytes [2]. The minimum value for SEG.BMAX is 24 bytes (the maximum size of a header during the SYN phase of the protocol) A 6 byte difference exists between these two values, which is therefore the largest amount of data which a client could possibly send by setting the SEG.BMAX on the remote host to it's minimum value.

Arrival of segment in the SYN-SENT state

The pseudo-code on page 25 of the specification reads:

```
If ACK set
      If RST clear and SEG.ACK != SND.ISS
            Send <SEQ=SEG.ACK + 1><RST>
      Endif
      Discard Segment; Return
Endif
```

(... continued)

The SYN-SENT state is reached after the client has issued an active open request to the RDP layer. Acknowledgement segments received in this state during the normal course of events would indicate the response of a foreign host listening on the given port. The response would typically be one of:

```
<SEQ=SND.ISS><ACK=RCV.CUR><MAX=SND.MAX><BUFMAX=RBUFMAX><ACK><SYN>
```

where the value of RCV.CUR is set by the foreign host to be that of the SEG.SEQ number of the received segment. This in turn is set by the local host to be that of the initial sequence number during the active creation of a connection (page 25):

```
<SEG=SND.ISS<MAX=SND.MAX><MAXBUF=RMAX.BUF><SYN>
```

Therefore, under the normal course of events, the RST flag will be clear, and SEG.ACK will always be equal to that of SND.ISS. The segment is thus discarded leaving the connection in the SYN-SENT state. As discussed previously, SYN segments are not sent reliably. This therefore results in a 'stale-mate' situation during the connection establishment phase.

The correct implementation would be to discard the segment only if both tests held true, and to de-allocate the connection record (as per RCF 1151). Therefore the pseudo-code should read:

```
If ACK set
      If RST clear and SEG.ACK != SND.ISS
            Send <SEQ=SEG.ACK + 1><RST>
            Discard Segment; Return
      Endif
Endif
```


Resetting of the RCV.CUR variable by a NUL segment

The NUL segment is used to determine whether the other side of the connection is still active. A connection that receives a NUL segment should acknowledge it if the sequence number falls within the acceptable sequence number window [1].

In the OPEN state, a received NUL segment sets the RCV.CUR variable (the sequence number of the last segment received correctly and in sequence) to that of the NUL segment's sequence number, effectively resetting the window of acceptable sequence number's size (page 28). The reason for resetting this flow control parameter midway through a connection is not clear and opens up a window of opportunity for misuse. A RDP implementation could conceivably 'force' packets onto a foreign host by resetting the foreign host's flow control window after every segment.

Acknowledging out of sequence packets when not supported

Page 27 (SYN-RCVD state) and page 29 (OPEN state) of RFC 908 deal with the acknowledgement of data in incoming segments. The pseudo-code reads:

```
If Data in segment
      If the received segment is in sequence
            Copy the data to user buffers
            ...
      else
            If out of sequence delivery permitted
                  Copy the data to user buffers
            Endif
            Send <SEQ-SND.NXT><ACK=RCV.CUR><ACK><EACK><RCVDSEQNO1>
                        <RCVDSEQNOn>...
      Endif
Endif
```

The problem arises when out-of-sequence acknowledgement of packets is not supported, the data is therefore not copied to the user buffers and the protocol acknowledges the packet. To correct this, the pseudo-code should read:

```
If Data in segment
      If the received segment is in sequence
            Copy the data to user buffers
            ...
      else
            If out of sequence delivery permitted
                  Copy the data to user buffers
                  Send <SEQ-SND.NXT><ACK=RCV.CUR><ACK><EACK>
                        <RCVDSEQNO1><RCVDSEQNOn>...
            Endif
      Endif
Endif
```


Management of retransmission buffers

Previous implementations[4, 15] of the protocol placed the responsibility of the retransmission expiry event on the packet itself. The author suspected that this method for the tracking of expiry times may be too heavy-weight and chose to implement the retransmission queue as a `java.util.Vector` of packets, ordered on sequence number. A thread would then scan the packets on the queue, retransmitting packets as necessary. This had several important implications.

Firstly, the entire queue needed to be scanned in order to determine which packets needed to be expired and the time to wait before scanning the queue again (the earliest expiry time of all the packets on the queue).

Secondly, if an extended acknowledgement was received, the search operation for the packet to be marked as acknowledged was an operation in the order of *O(n)* magnitude where n is the position in the queue of the packet to be acknowledged.

The benefit of using this approach was firstly simplicity, and secondly, the ease of flushing packets from the queue when a cumulative acknowledgement arrived.

Several alternative approaches were considered using ordered arrays instead of the `java.util.Vector` class as the backing queue. The MIDP2 profile however does not provide any of the array utility methods (e.g. binary searching) found in the J2SE platform and the benefit of this approach would therefore be minimal without providing custom implementations for these utilities. It was therefore decided to keep the implementation simple, possibly at the expense of performance.

4. Preliminary evaluation of the protocol

Memory profiling

Aim

To profile the runtime memory of a MIDlet during the transmission of data utilising the RDP libraries under varying:

- packet sizes.
- retransmission queue sizes.
- retransmission queue fill levels (the degree to which the retransmission queue is filled).

Methodology

A MIDlet was written that sent a packet of data using the author's RDP library to a given port every 0.5 seconds (the 'sending' MIDlet). The size of the packet sent was equal to the maximum segment size (set in the Java Application Descriptor – JAD file). 500 packets were sent. The amount of used memory (total memory – free memory) was output before sending a packet. The MIDlet code was obfuscated and optimised using the ProGuard toolkit [12].

Two 'receiving' MIDlets were then written, one which acknowledged the received packet immediately (immediate acknowledgement), and the other which only acknowledged the packet once the retransmission queue of the sending MIDlet was full (delayed acknowledgement). The delayed acknowledgement of a segment by a receiving MIDlet and subsequent flushing of the segment from the receiving MIDlet's retransmission queue needed to take place before the sending MIDlet tried to send another packet. Failure to do this would result in an "Insufficient resources to send data" error as per the specifications (RFC 908). No data was sent from the receiving MIDlet.

Since the maximum segment size (RBUF.MAX) and maximum number of outstanding segments (SND.MAX) are both variables that are specified at connection open time and are exchanged between the connecting peers during connection establishment, each peer in the connection is aware of the other peer's connection open parameters by the time the connection is established. Both of these parameters are read from the JAD file, thus negating the need to recompile the MIDlet for different scenarios.

The heap size of the sending MIDlet was set to 128Kb. The retransmission queue time-out was set to 20000 (20 seconds) so as not to play a significant role. The maximum segment

size (packet size) was alternated between 100, 200, 400 and 800 bytes. The retransmission queue size was alternated between 5, 10 and 20 packets. Both MIDlets were run on a single Pentium III 733 Mhz with 256 MB RAM.

A set of test scenarios were then run, each scenario with a different combination of parameters as specified above. The tests were executed with the following command, enabling verbose garbage collection output of the sending MIDlet, piping the results to a file:

```
C:\dev\j2me\wtk20\bin\emulator -Xheapsize:128k -Xdescriptor:
TestMemoryMIDlet.jad -Xverbose:gc > filename.txt
```

Several sequences of collected data were plotted against one another in line graph format (Appendix A).

The scenarios tested are laid out in Table 2.

Test no.	*Packet size (bytes)*	*Retransmission queue size*	*Acknowledgement Type*
1	100	5	Immediate
2	100	5	Delayed
3	100	10	Immediate
4	100	10	Delayed
5	100	20	Immediate
6	100	20	Delayed
7	200	5	Immediate
8	200	5	Delayed
9	200	10	Immediate
10	200	10	Delayed
11	200	20	Immediate
12	200	20	Delayed
13	400	5	Immediate
14	400	5	Delayed
15	400	10	Immediate
16	400	10	Delayed
17	400	20	Immediate
18	400	20	Delayed
19	800	5	Immediate
20	800	5	Delayed
21	800	10	Immediate

Test no.	*Packet size (bytes)*	*Retransmission queue size*	*Acknowledgement Type*
22	800	10	Delayed
23	800	20	Immediate
24	800	20	Delayed

Table 2 - Scenarios tested

It is worthwhile to note that the execution of this test assumed a zero loss rate of packets over the network, with all packets being delivered in sequence. Loss of a data packet in the delayed acknowledgement test scenario would result in the further delay of the acknowledgement of the packet to be acknowledged next, by an interval of at least the retransmission time-out interval (assuming that no packet loss occurred during the retransmission).

Results

Tests 13, 16, 19 and 20 terminated part way through due to a internal virtual machine error. The error output was:

```
ALERT: Heap address is not four byte aligned
```

No reference to the error type could be found in the bug report for the KVM machine. The results were able to be duplicated.

Tests 22 and 24 terminated during execution with the exception: "Insufficient resources to send data". Although the packet at which the test terminated varied slightly between runs, the error was consistent.

Analysis

Termination of tests 22 and 24 were consistent with the specification. Output revealed that at some stage after the retransmission queue on the sending peer reached its maximum size, a packet acknowledging the the first segment on the queue was not received before the sending peer sent another segment resulting in an "Insufficient resources to send data" error. Both of these scenarios occurred under delayed acknowledgement of packets with the packet size set to 800 bytes. Under these conditions, the receiving peer's time to parse the packet and respond is delayed due to the packet size and overhead required to delay the sending of acknowledgement, resulting in the error.

The rate of heap size memory allocation increases more rapidly per packet sent with larger packet sizes (graphs 1, 2, 3, 4, 5, 6). This was to be expected. The rate of heap size allocation however, is not directly proportional to the size of the packet as illustrated in the following table (tables 3 and).

Test no.	*No of packets sent*	*Packet size (bytes)*	*Total data (bytes)*	*No garbage collections (GC)*	*GC's per packet*	*GC's per Kb*
3	500	100	50000	14	0.028	0.287
9	500	200	100000	15	0.030	0.154
15	500	400	200000	17	0.034	0.087

Table 3 - Heap size utilisation (Immediate acknowledgement)

In all cases the retransmission queue size was 10 packets.

Although the number of garbage collections per packet increases as packet size increases, the number of garbage collections per Kb of data transferred decreases in an almost linear manner as the packet size increases. The transfer of data in larger packets using the author's implementation is therefore more efficient in terms of the rate of heap size allocation per Kb of data, than the transfer of the same data in smaller segments. A similar pattern was noted with the delayed acknowledgement of packets.

The average basal heap size (the average heap size after garbage collection had taken place) was noticeably different between different test scenarios using delayed acknowledgement with varying packet sizes. This is illustrated in table 4, and graphs 4, 5, and 6.

Test no.	*Packet size*	*Retransmission queue size*	*Average basal heap size*
2	100	5	44634
4	100	10	45713
6	100	20	46590
14	400	5	47167
16	400	10	50274
18	400	20	53864

Table 4 - Average basal heap size with varying packet and retransmission queue sizes.

In all cases acknowledgement was delayed.

The basal heap size increased both as the number of packets on the retransmission queue and the packet size increased. These increases were roughly linear, although more test cases would need to be run in order to qualify this.

Tests 1, 2, 5, 6, 13, 14, 17 and 18 (graphs 7, 8, 9 and 10) each plot the differences in heap memory size per packet sent for the same packet size and retransmission queue size under varying acknowledgement types. Interestingly in all of these tests, the first peak in memory use occurs at a lower number of packets with the immediate acknowledgement type in all of the graphs. This is more noticeable in the tests with larger retransmission queue sizes. This trend is then reversed as the test progresses, with a smaller number of packets between peak-to-peaks in the delayed acknowledgement type. These changes in the delayed acknowledgement tests can be attributed to firstly the higher basal heap size in the delayed acknowledgement cases, and secondly to the lower rate of heap memory usage as the retransmission queue fills initially and no acknowledgements are received. Once the

retransmission queue is filled, the amount of heap size memory utilised per packet is the same; the only difference in the profile is the level to which the heap size falls after garbage collection.

The heap size memory trough during run time profiling never fell below 41 Kb. This implementation of the RDP is therefore not suited to devices with the minimal amount of volatile memory (32 Kb).

Finally, the heap size after garbage collection in all cases returns to a level consistent with that of the previous heap size after garbage collection. This indicates the absence of memory leaks in the implementation.

Throughput analysis

Aim

To quantify and evaluate the effective throughput of the RDP implementation under varying packet loss rates.

Methodology

A MIDlet was written to send packets continuously, within the confines of the protocol viz. SND.NXT < SND.UNA + SND.MAX. The receiving MIDlet was then written to drop packets at a given rate, only acknowledging packets that were not dropped. The drop rate was varied between 1 and 12 percent. 300 packets were sent. Extended acknowledgement was enabled.

The connection timeout interval was set to 1000 ms, the size of the retransmission queue was set to 5 packets. The packet size was varied between 100, 200 and 400 bytes.

Results

Several of the tests terminated abnormally with a virtual machine error. This error was the same as that in previous tests (`ALERT: Heap address is not four byte aligned`). The results were plotted, ignoring those that terminated abnormally (see graph 11).

Analysis

The number of packets sent to derive the results was low (300). This was necessary as the virtual machine errors occurred more frequently using larger packet numbers. This however made for difficult interpretation of the results as the quantity of data was such that varying results were often collected using the same input parameters. Where possible, an average of the results were used. Small fluctuations need therefore to be interpreted with caution.

Throughput is higher with larger packet sizes. The rate of throughput increases with increasing packet size is more significant at smaller packet sizes.

With small packet sizes (100 and 400 bytes) throughput drops off in a linear manner as the rate of packets loss increases. At 800 byte packet size, no drop off of throughput occurs. This anomaly occurs due to the fact that at 800 bytes, the sheer size of the packet and consequent time incurred to parse the packet into an object representation becomes the limiting factor in throughput, and not the loss rate on the network.

Comparison with other protocols and implementations

Before comparing this implementation with other protocols and implementations we need to take cognisance of the aims of the implementation and the fact that there is no good standard for making the comparisons.

TCP is the dominating transport protocol on the Internet [18] and has been tuned for traditional networks involving stationary hosts and wired links. TCP error control is centred on congestion losses and ignores the possibility of transient random errors or temporary 'black-outs' due to hand-offs and extended burst errors that are typical in wireless networks [19]. TCP employs several methods to adapt to packet losses by dropping the transmission window size before retransmitting packets, initiating congestion control or avoidance mechanisms (e.g. slow start) and by backing off its retransmission timer [20]. Therefore, when packets are lost for reasons other than congestion, an unnecessary reduction in the end to end throughput and high interactive delays are noticed.

The shift towards a wireless environment, with the possibility of non-stationary hosts has meant that the protocol has been revisited in an attempt to improve the performance of the protocol over wireless links. These attempts have centred on two main approaches, hiding all non-congestion losses from the TCP sender and end-to-end approaches which are based on making the sender of packets aware of existing wireless hops and different propositions for new versions of TCP (e.g. TCP Westwood) [18].

In contrast, RDP is intended to be efficient in environments with long transmission delays and loss or non-sequential delivery of message segments [1]. Although the intentions (and the specification) do not address all the problems and requirements for efficient data transmission over wireless networks, it provides an alternative model from which to evaluate wireless data transport.

The RDP windowing scheme is more suited to packet loss than that of TCP's: extended acknowledgements allow for the acknowledgement of segments received out of order and the protocol allows any packet on the retransmission queue to be retransmitted, not just the first packet. Locating packets on the retransmission queue proved to be an *O(n)* access time operation where *n* is the number of packets on the retransmission queue. This was a similar performance bottleneck as that encountered in the BSD implementation [4].

TCP is able to 'manage' window sizes; RDPs window size is specified at connection initialisation and cannot be altered – dynamic flow control therefore cannot be exercised as with TCP. Because of this ability, TCP deals with peers with mismatched speeds in a better manner.

This implementation of the protocol did not make an attempt to calculate packet round trip time and adjust the retransmission queue time-out based on this calculation. This was in contrast to the BSD implementation [4].

[4] cites that the BSD 4.2 TCP system was usually incapable of sustaining a connection when the loss rate exceeded 10%. The results from the tests run on this implementations indicate that although the throughput decreased in a linear manner with an increasing packet loss rate, the connection was maintained, event at a loss rate of 12%.

Of interest is the reduction in throughput with increased error rates. [21] states that a 1.55% frame error rate clustered (as in often the case with wireless links) decreased TCP throughput by 22% below the nominal rate. This is due to TCP frequently invoking congestion avoidance mechanisms even though losses are not due to congestion. This is in contrast to a reduction of throughput of 6.7% (100 bytes) at a clustered error rate of 2% demonstrated with the author's implementation.

Overall throughput of the implementation peaked at +- 1700 bytes per second with a packet size of 400 bytes and no packet loss. Increasing the packet size beyond this amount did not result in any significant increase in throughput. Unfortunately no comparison could be drawn with other implementations of RDP [4,15] as no throughput figures were published; when compared to implementations of TCP the maximum throughput achieved was insignificant.

Use of the Java platform

The promise of portability, ubiquity and simplified programming model could conceivably be achieved by using Java as a platform. This contrasts with the implementation of of the protocol under the BSD 4.2 and 4.3 system which only offers reasonable portability across different machines running the BSD system [4].

Several drawbacks to using the J2ME as a platform were noticed however.

Java is an interpreted language. This adds another layer of indirection, allowing for the

realisation of the portability promise of the language. This however comes at a trade-off, slower speed of interpretation (execution) and the standardised accessibility to low level APIs. Programmers are thus unable to take advantage of enhanced features on certain platforms without resorting to native code calls. Unfortunately, the J2ME platform does not allow the use of native code.

Garbage collection algorithms, although advanced in the J2SE platform, are limited by the sheer size restrictions of the virtual machine (Kilobyte Virtual Machine – KVM) used. Less complex algorithms such as the mark and sweep algorithm typically halt the execution of code during collection. The platform is thus still maturing in this respect.

5. Conclusion

Although the J2ME platform is still maturing in many regards, the RDP implementation on a MIDP2 enabled device provided a suitable model for experimentation on a platform with limited resources.

The runtime environment however was limiting in several aspects. Firstly, the kilobyte virtual machine (KVM) proved to be unstable and crashed regularly with a suspected memory allocation error. Secondly, the interfaces (APIs) exposed by the profile were often a limiting factor in the implementation resulting in performance trade-offs. Lastly, the runtime execution speed of the code was found to be a limiting factor in the throughput of the implementation at packet sizes above 400 bytes.

The task of comparing this implementation with other protocols and implementations was therefore a difficult one from a high level perspective. The implementation did however prove useful in highlighting specific advantages of the protocol over some of other transport protocols, notably TCP, when used over wireless links.

References

1. David Velten, Robert Hinden and Jack Sax, "Reliable Data Protocol", RFC908, In ARPANET Working Group Request For Comments, no 908, SRI International, Menlo Park, California. 1984.

2. Craig Partridge and Robert Hinden, "Version 2 of the Reliable Data Protocol (RDP)", RFC1151, April 1990

3. Jon Postel, ed, "Transmission Control Protocol", RFC793, In ARPANET Working Group Request For Comments, no 793, SRI International, Menlo Park, California, 1982.

4. Craig Partridge, "Implementing the Reliable Data Protocol (RDP)",

5. The Java 2 Platform, Micro Edition, http://java.sun.com/j2me

6. Kim Topley, "J2ME in a Nutshell, A Desktop Quick Reference", O'Reilly and Associates, Inc. 2002

7. John Muchow, "Core J2ME Technology and MIDP", The Sun Microsystems Press, 2002.

8. MIDP Version 1.0, Java Specification Request JSR37, http://jcp.org/jsr/detail/37.jsp

9. MIDP Version 2.0, Java Specification Request JSR118, http://jcp.org/jsr/detail/118.jsp

10. Mobile Information Device Profile Data sheet, http://java.sun.com/products/midp/midp-ds.pdf

11. Jonathan Knudsen, "Understanding MIDlet memory", http://wireless.java.sun.com/midp/ttips/memory, June 2002

12. Eric LaFortune, Proguard Obfuscator and Shrinker, http://proguard.sourceforge.net/

13. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns, Elements of Reusable Object-Oriented Software", Addisson-Wesley, 1994.

14. Jon Postel, "Internet Control Message Protocol", RFC792, In ARPANET Working Group Request For Comments, no 792, SRI International, Menlo Park, California, 1981.

15. Faber Ted, "An Implementation of the Reliable Data Protocol for Active Networking", USC/ISI.

16. James Gosling, Bill Joy, Guy L. Steel Jr, Gilad Bracha, "The Java Language Specification, Second Edition", http://java.sun.com/docs/books/jls/index.htm, Sun MicroSystems, 2000.

17. The Java Bug Parade, http://developer.java.sun.com/developer/bugParade/ (login required)

18. Anna Ewerlid, "Reliable Communication over Wireless Links", Signals and Systems, Uppsala University.

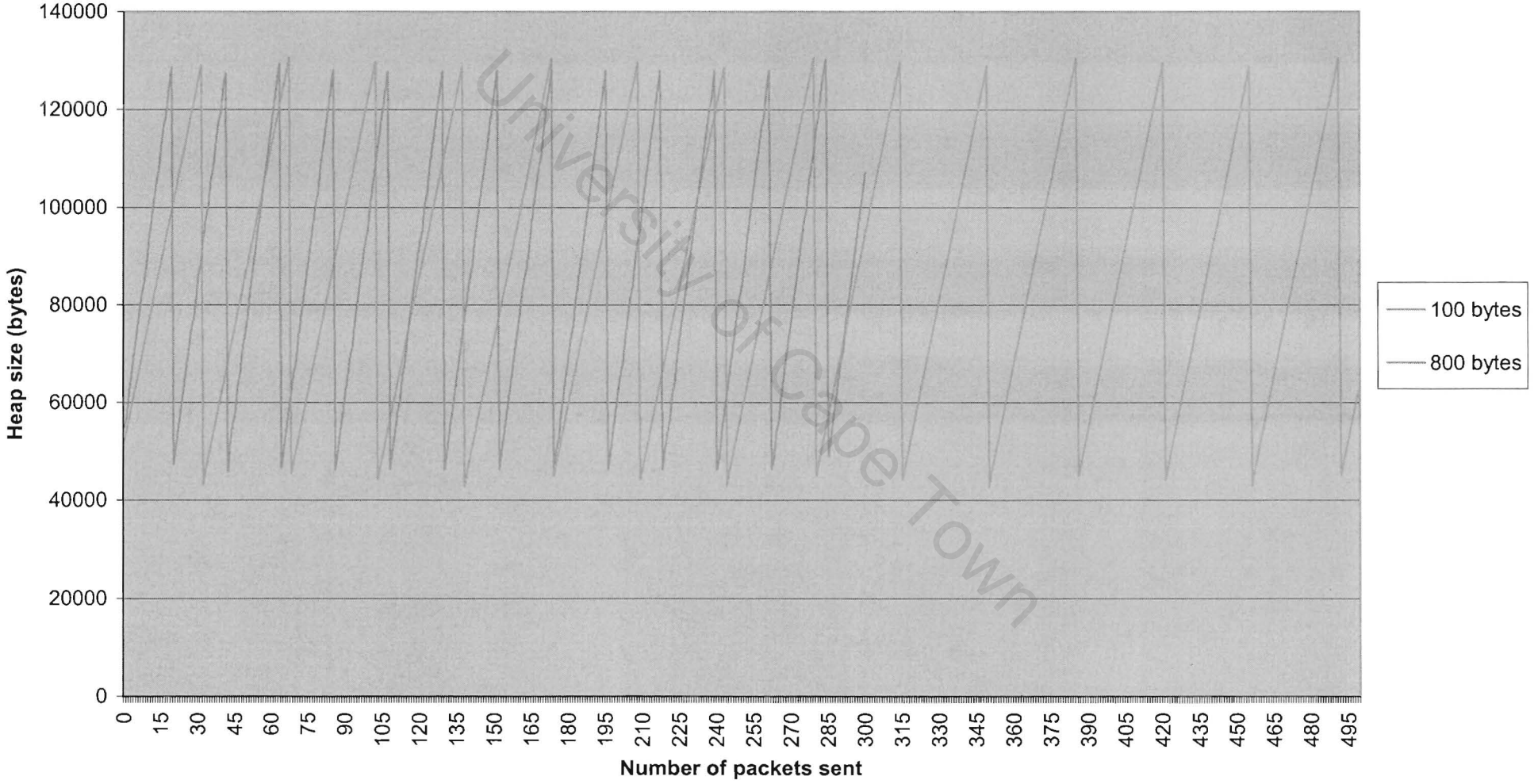
19. Vassilis Tsaoussidis, Ibrahim Matta, “Open Issues on TCP for Mobile Computing”, Northeatern University and Boston University.

20. Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Sheshan and Randy H Katz, “A Comparison of Mechanisms for Improving TCP Performance over Wireless Links”, Computer Science Division, University of California at Berkeley.

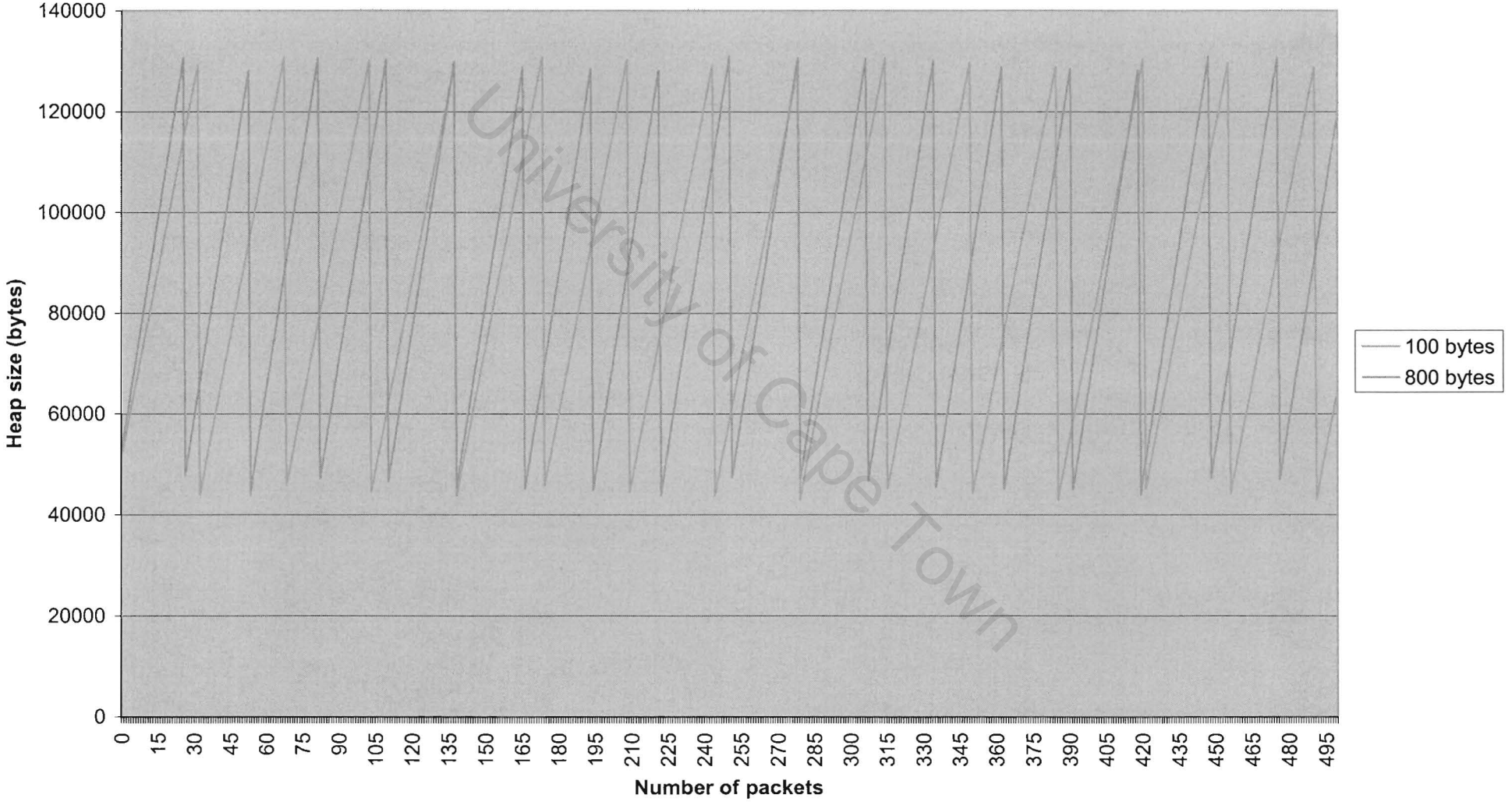
21. George Xylomenos and George C. Polyzos, "Internet Protocol Performance over Networks with Wireless Links", Center for Wireless Communications and Computer Systems Laboratory, Department of Computer Science & Engineering, University of California, San Diego

Appendix A – Memory profiling graphs

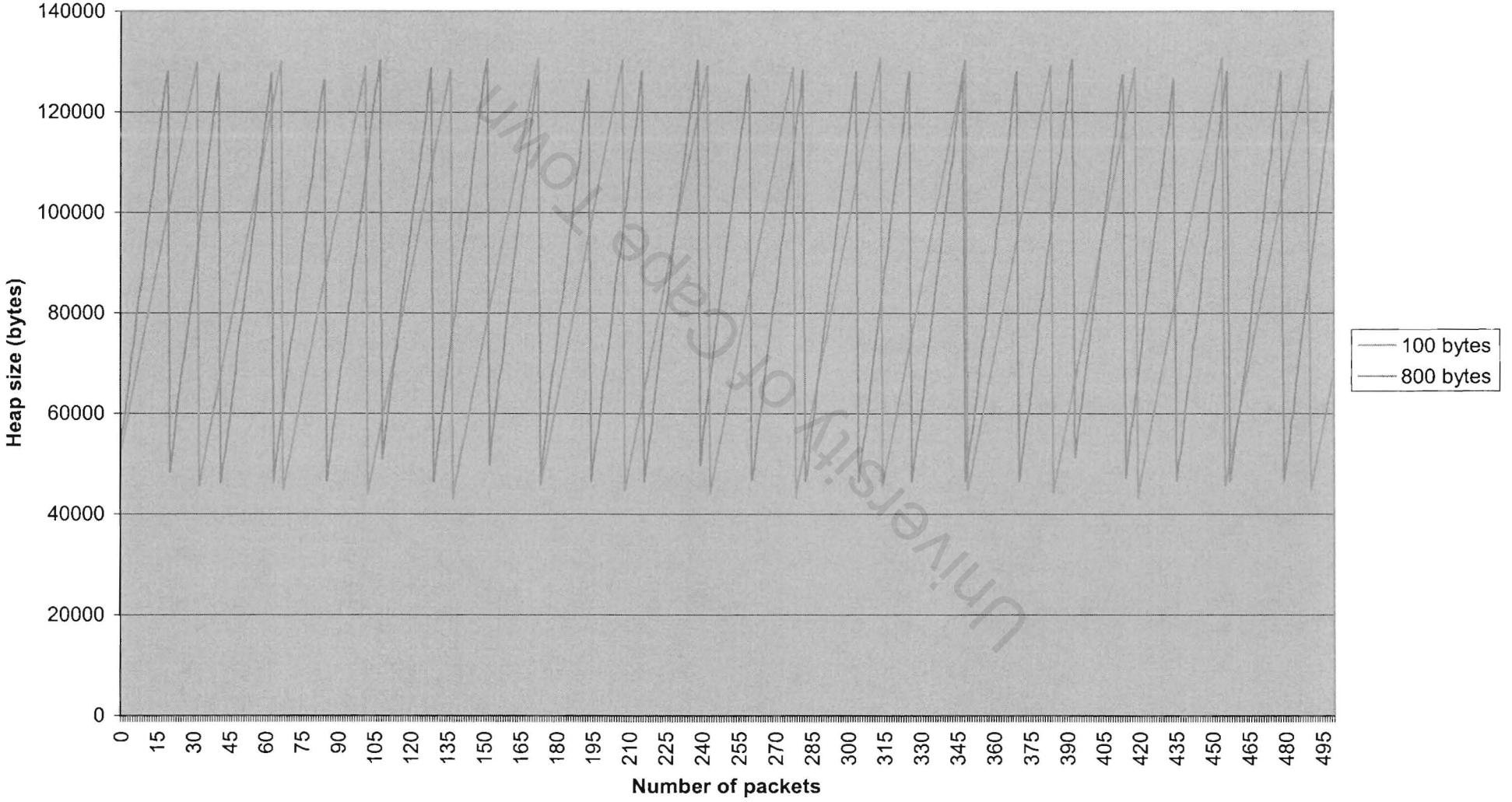
Graph 1. Heap size fluctuations with varying packet sizes.
Retransmission queue size: 5. Immediate acknowledgment



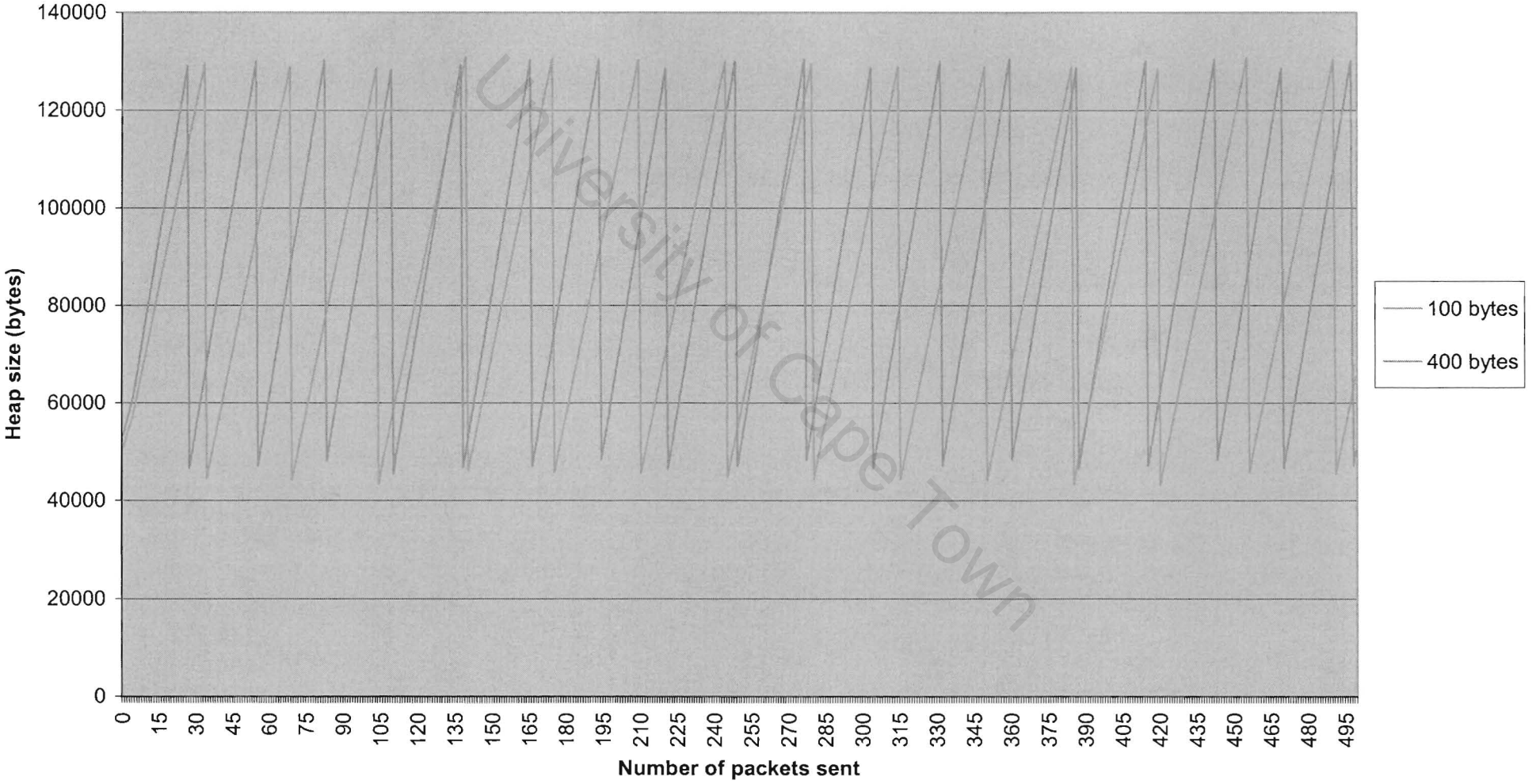
Graph 2. Heap size fluctuations with varying packet sizes.
Retransmission queue size: 10. Immediate acknowledgment



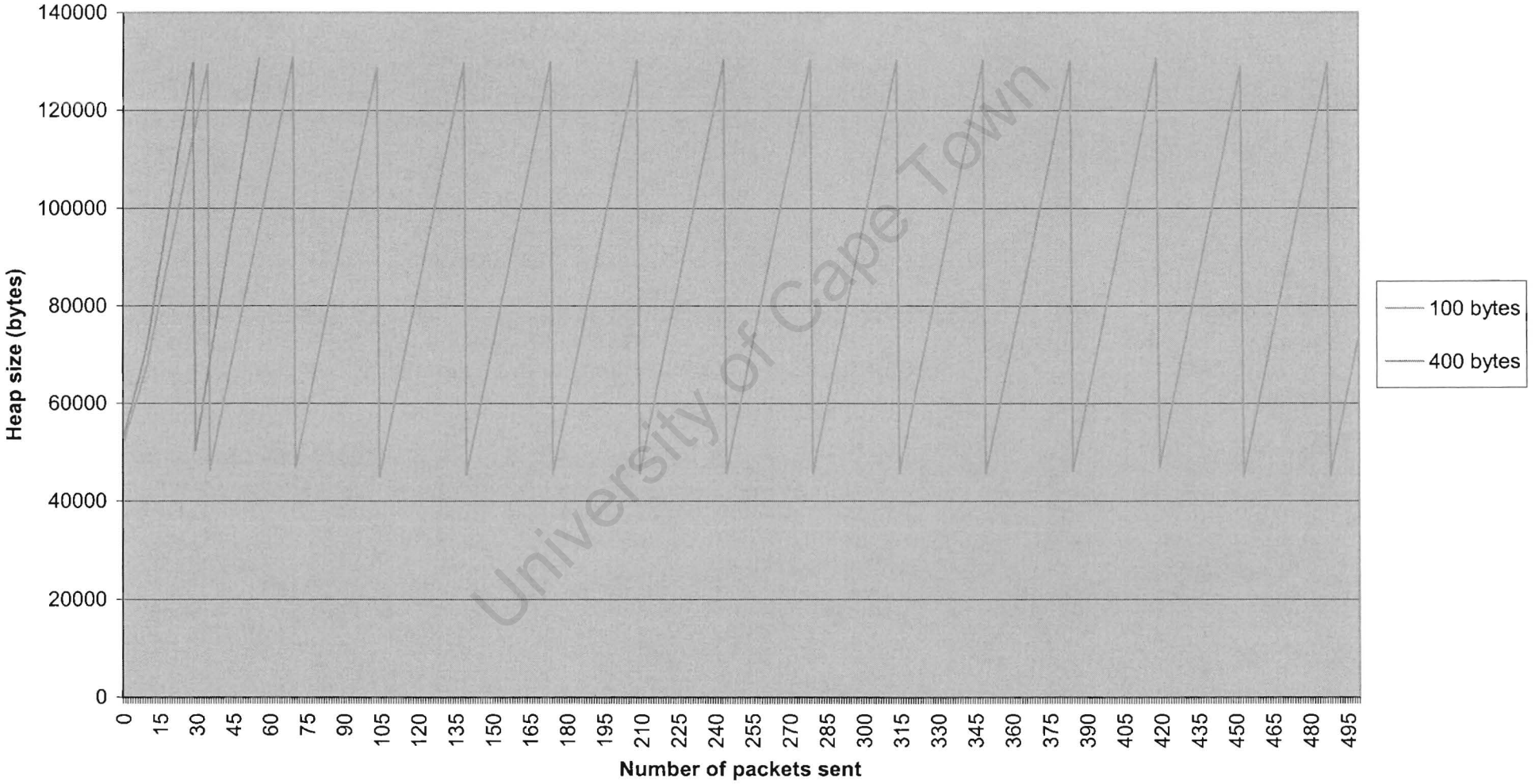
Graph 3. Heap size fluctuations with varying packet sizes.
Retransmission queue size: 20. Immediate acknowledgment



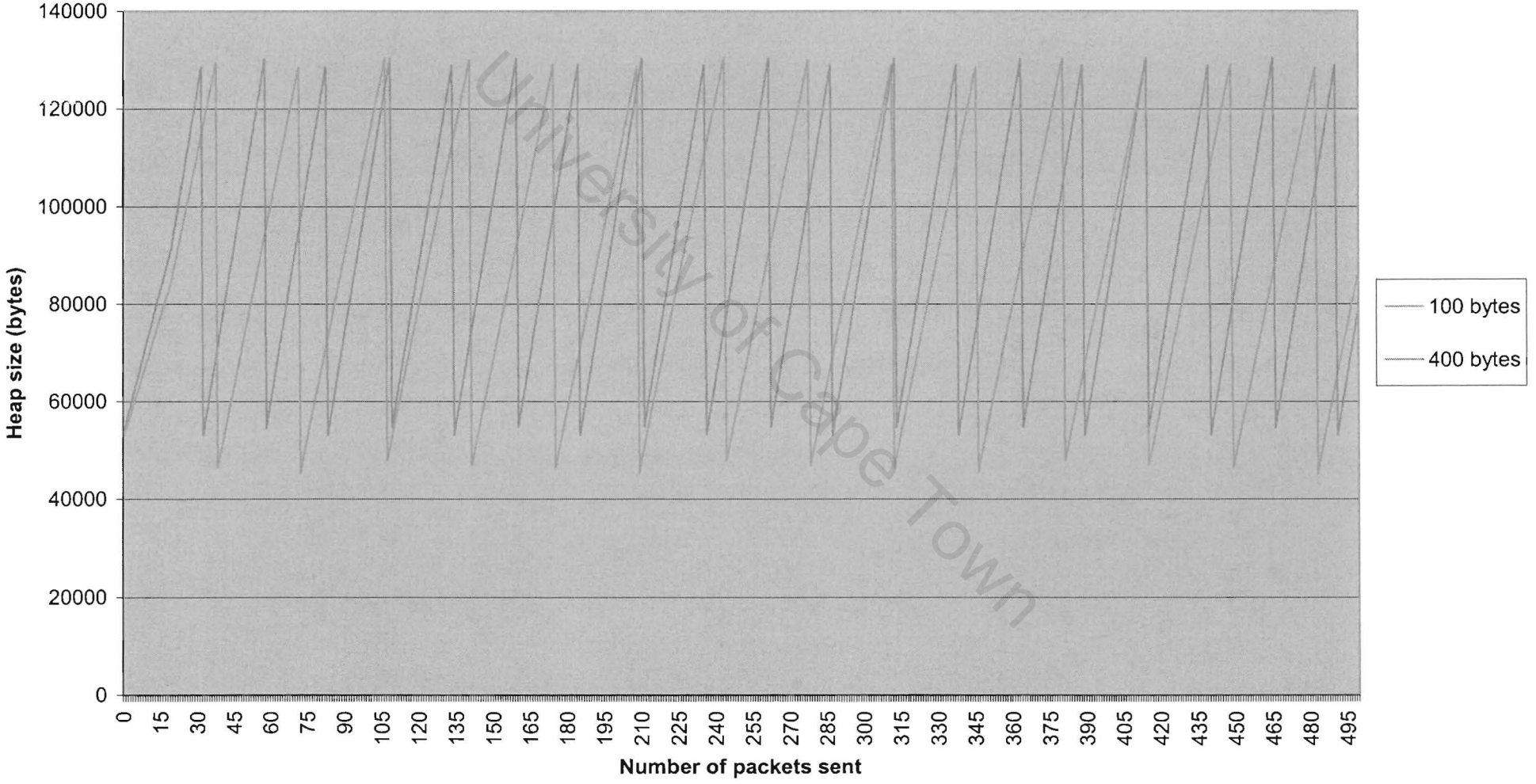
Graph 4. Heap size fluctuations with varying packet sizes.
Retransmission queue size: 5. Delayed acknowledgment



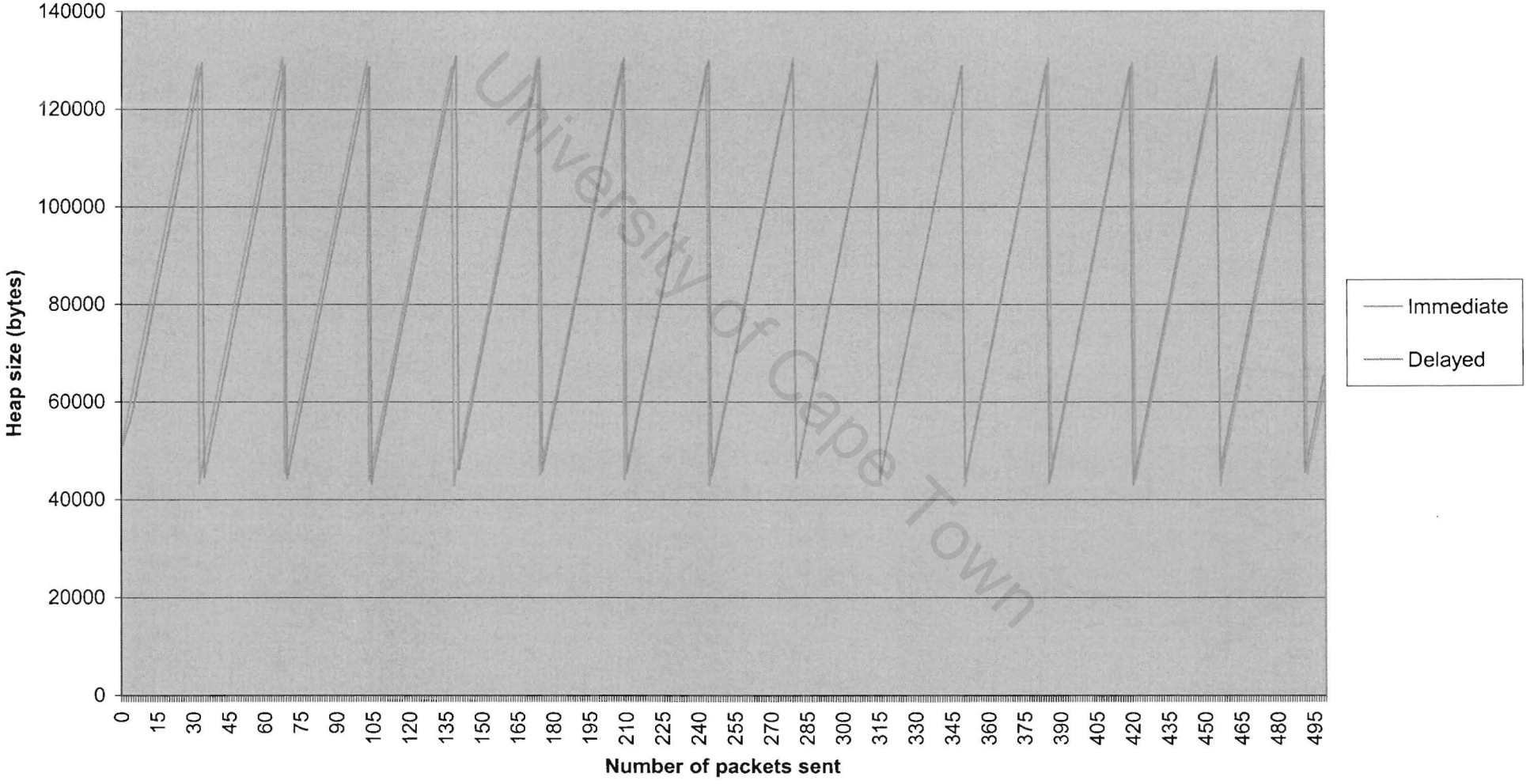
Graph 5. Heap size fluctuations with varying packet sizes.
Retransmission queue size: 10. Delayed acknowledgment



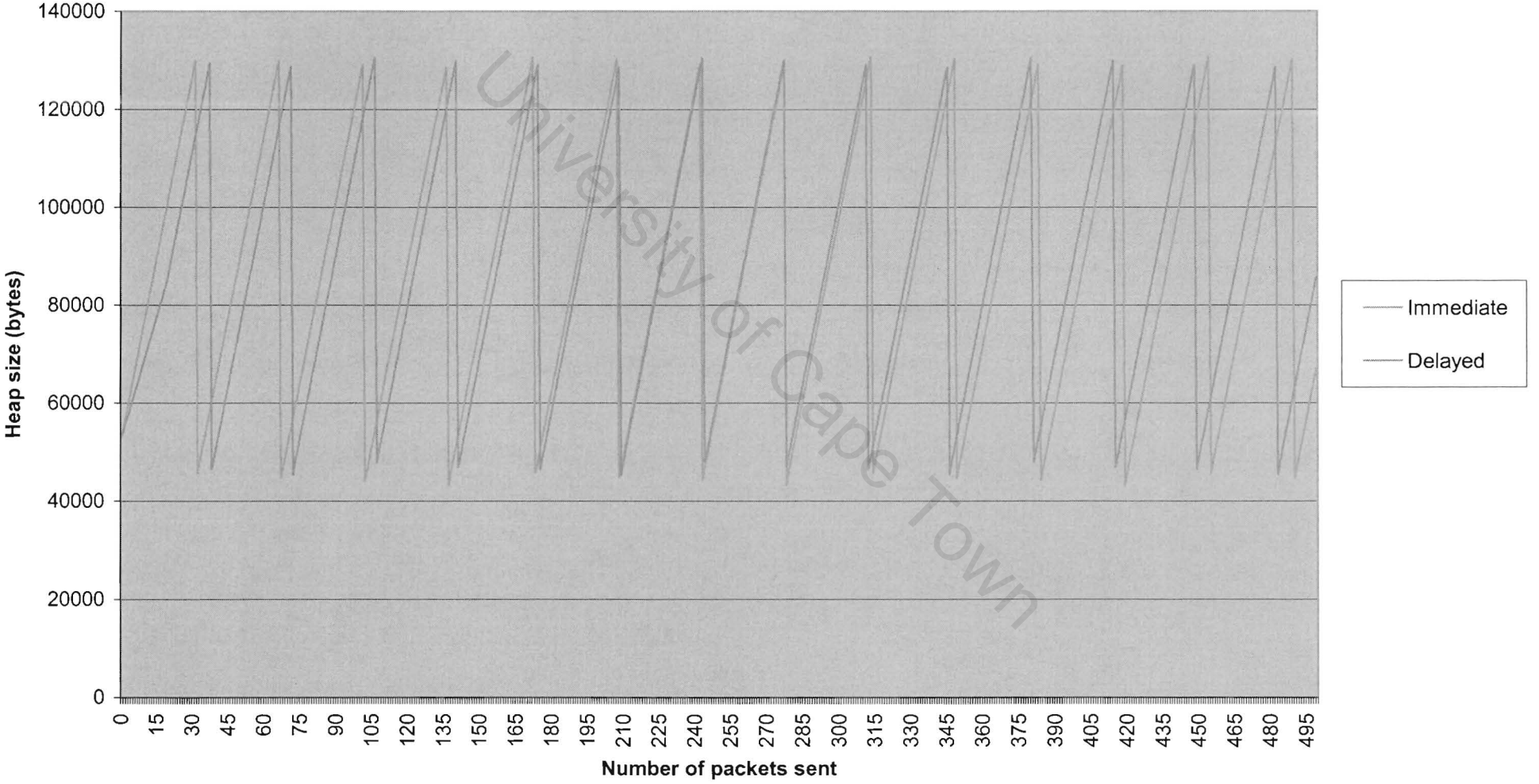
Graph 6. Heap size fluctuations with varying packet sizes.
Retransmission queue size: 20. Delayed acknowledgment



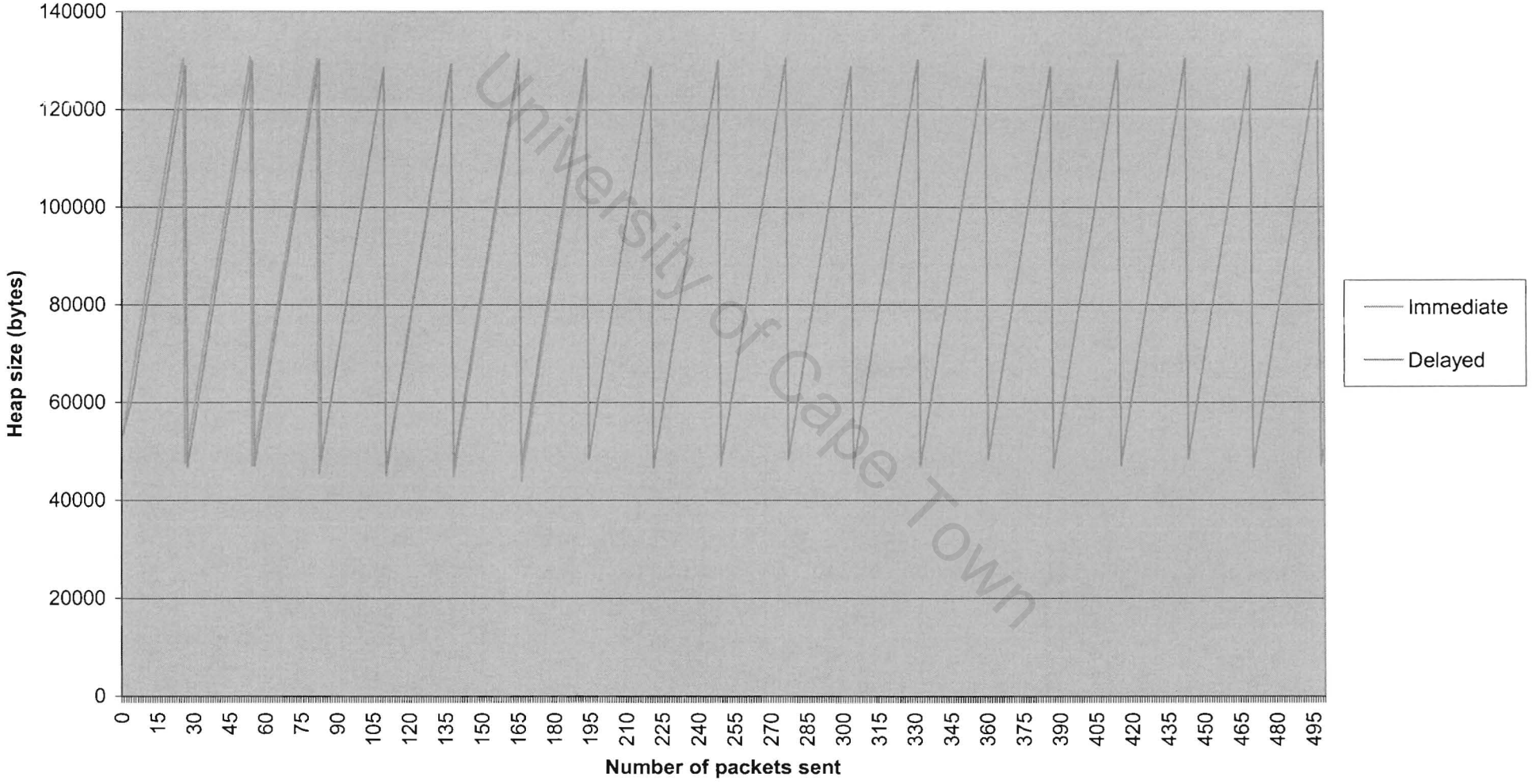
Graph 7. Heap size fluctuations with different acknowledgment protocols. Retransmission buffer maximum size: 5. Packet size: 100 bytes.



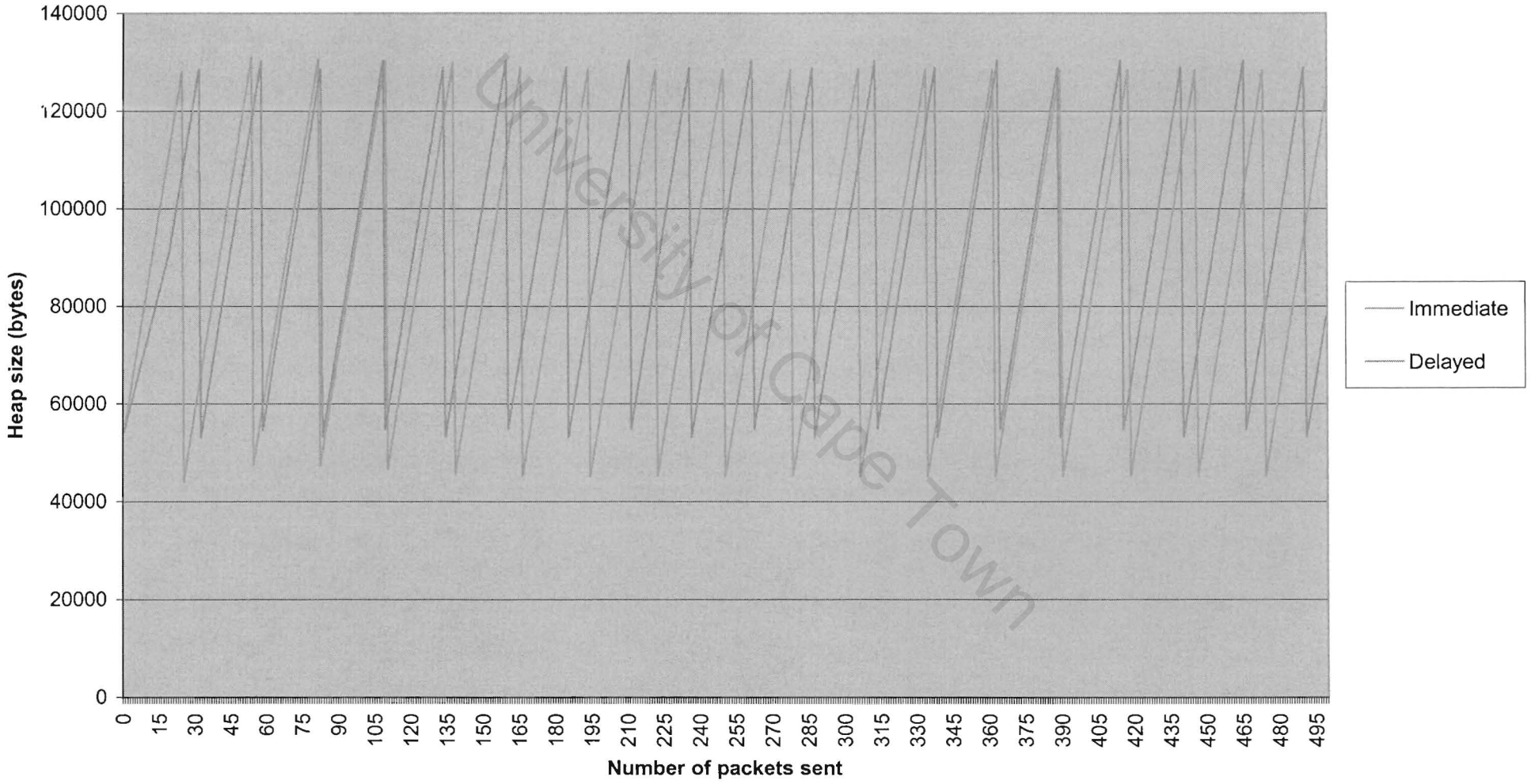
Graph 8. Heap size fluctuations with different acknowledgment protocols. Retransmission buffer maximum size: 20. Packet size: 100 bytes.



Graph 9. Heap size fluctuations with different acknowledgment protocols. Retransmission buffer maximum size: 5. Packet size: 400 bytes.



Graph 10. Heap size fluctuations with different acknowledgment protocols. Retransmission buffer maximum size: 20. Packet size: 400 bytes.



Appendix B

Usage instructions for the RDPMIDlet example

The RDPMIDlet example, demonstrating the use of RDP running as a MIDlet on a mobile device, consists of two files:

1. RDPMIDlet.jad - The Java Application descriptor for the application.
2. RDPMIDlet.jar - The Java Archive containing the application

The application can be run on a mobile device supporting the MIDP2 specification, or an MIDP2 emulator. This appendix describes the installation and execution on a MIDP2 emulator.

Prerequisites

Download and install a MIDP2 emulator. Sun Microsystems provides a MIDP2 emulator based on the reference implementation of the MIDP2 .0 and CLDC 1.0 specifications called 'The Wireless Toolkit" (WTK). It can be downloaded free of charge from http://java.sun.com/products/j2mewtoolkit/download-2_0.html. The author used version 2.0 of the WTK.

The minimum system requirements for the WTK version 2.0 are as follows:

- Operating system: Windows XP, Windows 2000, Windows 98, Windows NT, Solaris or Linux
- Java Runtime environment (JRE) 1.4 or later. This can be downloaded from http://java.sun.com/j2se/downloads.html .
- 50 MB of hard disk space
- 64 MB RAM
- 166 MHZ CPU

Running the application

1. Copy the two files to the same directory on your hard disk.
2. Change to this directory.
3. Execute the following command:

   Windows: `[path to WTK base]\bin\emulator -Xdescriptor:RDPMIDlet.jad`

   Unix: `[path to WTK base]/bin/emulator.sh -Xdescriptor:RDPMIDlet.jad`

You should be presented with an application similar to that of Figure 5. Note that the exact

skin of the phone may vary depending on your settings

Figure 5 - Initial screen shot

Figure 6 - Connection type

Figure 7 - Connection Details

Launch the application by selecting the 'Launch' button. Note that keypad mappings and control toggles vary widely between devices being emulated.

The application will then prompt you for the RDP connection type, Active or Passive (Figure 6). Select the connection type by navigating and selecting using the keypad and toggles provided and select 'OK' from the context menu.

Depending on the type of connection selected, the next screen will either prompt you for a remote host and port (active open) or just a local port (passive open) (Figure 7). Fill in the details as required and select 'OK' from the context menu.

During connection establishment the device may prompt you for permission to utilise airtime (Figure 8). Select either of the first two options. A passive connection will listen on the selected port until a connection is established (Figure 9), or the application is terminated by selecting 'Exit' from the context menu. An active connection will try to establish a

connection at the chosen host and the selected port until either a connection is established or the connection time-out expires, whichever is the sooner.

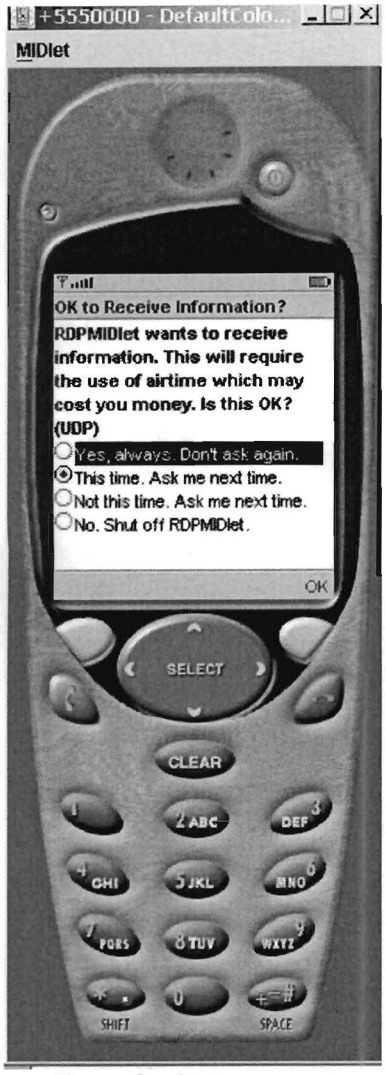


Figure 8 - Airtime prompt

Figure 9 - Listening for a connection

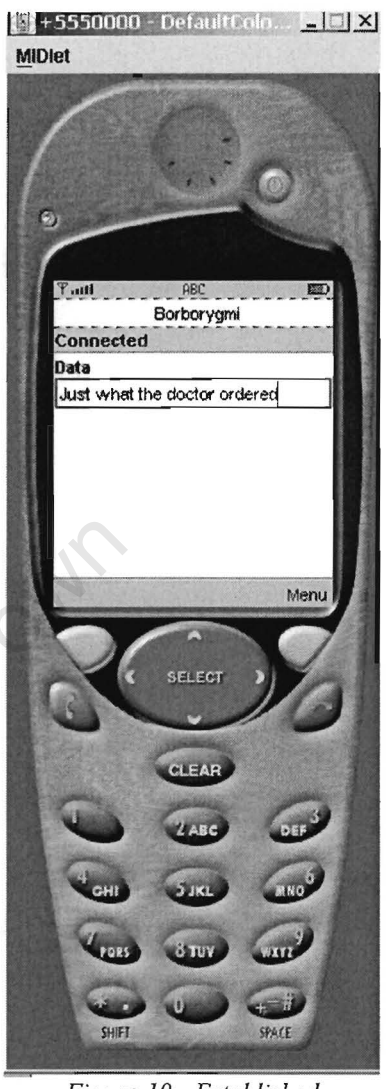


Figure 10 - Established

Once a connection has been established between two peers, a screen as in Figure 10 will be displayed. Incoming messages are displayed at the top of the screen, whilst text to be sent can be typed into the text area provided and sent by selecting send from the context menu.

Customising the RDPMIDlet

The Java Application Descriptor (JAD) file is a text file containing MIDlet properties including customisable RDP properties. The following table briefly describes the properties that can be changed between executions to allow MIDlet customisation.

Property name	*Description*
`RDP-MaxSegSize`	The maximum segment size (in bytes including the header size) that can be handled by this connection. Must be > 26.
`RDP-MaxUnackSegs`	The maximum number of unacknowledged segments that can be sent on this connection.
`RDP-RTO`	The retransmission queue time-out value, i.e. the number of milliseconds that a packet remains on the queue before being sent again.
`RDP-ConnTimeout`	The connection time-out value in milliseconds for an active open request. When issuing an open request the protocol will wait at most this number of milliseconds before aborting the connection request.