Wave & Wait Protocol (WWP)[1]
An Energy-Saving Transport Protocol for Mobile IP-Devices

Vassilios Tsaoussidis, Hussein Badr, Rohit Verma
Department of Computer Science
State University of New York at Stony Brook
New York 11794-4400

Abstract

This work involves the development of an experimental transport-level protocol running on top of IP for small, mobile, wireless stations. The protocol cannot and does not aim to handle real-time traffic. Its central concern is to conserve battery-powered energy used for transmission, even at the expense of slower data throughput. It attempts to conserve energy expenditure by adjusting the amount of data transmitted, trying to keep it below perceived network congestion. The higher the detected congestion risk-level in the network the less it attempts to transmit, thereby minimizing the need for duplicate data retransmission due to congested routers losing packets and so on. We outline the protocol specification, mechanisms and implementation, as well as preliminary evaluation results that clearly demonstrate the energy-saving capabilities of the protocol.

1. Introduction

Wireless communications are increasingly dominating today' s communication infrastructure. Many applications are being developed for mobile stations, which require very specific protocol support depending on the characteristics of both application (e.g. e-mail, web, multimedia) and mobile station (e.g. laptop, phone, handheld etc.). The protocol presented here is designed for battery-powered mobile devices and for non-real-time applications (e.g. e-mail). Since the market for such devices and applications is currently still evolving, there are relatively few appropriate protocols, and research in the field is being undertaken mostly by leading vendors.

Energy saving protocols take advantage of a novel approach, also used by other application-specific protocols: tradeoffs between different characteristics of QoS, and device- or application-specific characteristics. A familiar example is the Forward Error Correction (FEC), used in high-speed networks, which trades off bandwidth for reliability. Another example, AOTP [6], trades reliability for speed using a receiver-based approach to decide whether or not retransmission is required in order to conform to the user-prescribed QoS level. A similar approach, centered on a sender-based transport service is described in [2, 3]. The Wave & Wait Protocol (WWP) presented here favors energy-saving at the expense of time, two resources of varying significance depending on the application and circumstances.

3COM has designed a set of protocols to support messaging over TCP/IP for handheld devices using Palm OS [9]. The protocol attempts to save energy by limiting the transmitted data: it transmits 500 bytes during each communication phase under the assumption that the data of interest to the mobile user will be included within this limit. A user wishing to receive more data re-establishes communication for the next 500 bytes. The protocol design is fairly simple but practical: it trades off the amount of transmitted data in order to save battery power, and it does so at the expense of information. The approach takes into account the mobile user' s limitation of time as well as the fact that newer pieces of information are normally more significant than older ones.

The work we present here involves the development of an experimental transport-level protocol running on top of IP for small, mobile, wireless stations. The protocol attempts to conserve energy expenditure by adjusting the amount of data transmitted, trying to keep it below perceived network congestion: the higher the detected congestion risk-level in the network, the less it attempts to transmit, thereby minimizing the need for duplicate data retransmission due to congested routers losing packets, and so on.

2. Protocol Strategy

The idea behind the protocol is straightforward: when time is not a crucial factor we can save energy. The way to save energy is to avoid retransmissions, unnecessary headers, and redundant data. In brief, the less time expended on transmissions, the better the energy saving.

[1] 7th IEEE Conference on Network Protocols, ICNP '99, Toronto, Canada, October 1999.

For example, transmission of packets over a congested network will cause packets to be dropped and, consequently, a reliable protocol will initiate a retransmission mechanism. Instead, our protocol first "probes" the network to estimate prevailing levels of congestion risk and adjusts its transmission accordingly. This approach is quite different from that of more conventional, reliable transport protocols. TCP's congestion-control mechanism, for example, also monitors congestion and adjusts its congestion-window size in response. However, its approach to transmission at the window size set is quite aggressive. It will vigorously transmit to the limit permitted, and keep expanding its window at every opportunity, until it "overreaches" itself and is forced to adjust downwards. WWP takes a much more conservative approach. It will transmit if and only if it deems the risk of congestion to be sufficiently low, thereby reducing the need for retransmission and duplication of data. It will not attempt to transmit at all during periods when congestion conditions appear to be too high. It probes the network to locate windows of sufficiently low congestion opportunity to exploit for transmission. This results in less overall time being spent actually transmitting, though the connection time between peers might last longer than would be the case with more conventional protocols, especially if the network is, on the whole, significantly congested.

Retransmission-based error correction - in contrast to, for example, FEC - is an appropriate approach for bandwidth-limited communications (e.g. low frequency communications with handhelds). It becomes a necessary mechanism for reliable protocols, especially in the case of unreliable networks (e.g. IP networks). Including reliability features at the transport layer has two advantages for the application developer:

(i) he does not need to worry about reliable delivery of the messages sent; and

(ii) when a packet (or segment) is "corrupted", only this packet (or segment) need be retransmitted and not the entire message, which is the data unit that the application layer usually manipulates.

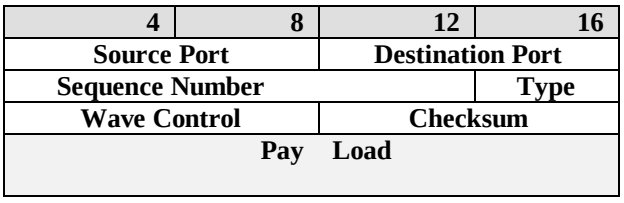
2.1 Protocol Outline and Justification

Broadly speaking, the protocol works as follows. Connection is first established using a six-way handshake. Apart from connection set-up, this six-way exchange is also used to determine current congestion conditions in the network. Using the established connection, a sender sends a "wave" to the receiver consisting of a number of fixed-sized data segments, and then waits for a response. The number of segments in a wave is set according to the current "wave level" which is determined by the receiver in line with the estimated prevailing congestion level, and is communicated to the sender. The less the perceived congestion risk in the network, the higher the "wave level" and the more segments that a wave comprises. The receiver uses the segments of an arriving wave to estimate network congestion, then sends just one Negative Selective ACK (N-S_ACK) for the entire wave, which also specifies the level for the next wave. The N-S_ACK is a NACK that identifies all lost segments that the receiver has not received up to that point. The sender has to retransmit these as part of the next wave, together with new segments, within the limitation of the wave size determined by the new wave level. This next wave starts with those segments that need retransmission, which are counted as part of this next wave. Only after these have been sent will the sender, wave size permitting, continue with new segments. If the receiver's N-S_ACK specifies zero as the next wave level, then this means that it deems the network to be too congested for energy-conserving transmission to be worthwhile at present (i.e., the receiver deems that too many segments might be lost, necessitating too many retransmissions). The sender would then periodically probe the receiver. These probes are used by the receiver to continuously monitor the network's congestion level. When conditions improve sufficiently, the sender resumes transmission at some appropriate wave level specified by the receiver.

Connection can be terminated at the initiative of either side, and involves a separate two-way handshake by each side as in TCP. The protocol provides a reliable, connection-oriented end-to-end service: on the receiver side, segments are delivered to the higher-level protocol in order, with no duplicates, and with no segments missing. The protocol first groups segments into waves on the sending side and then transmits the segments of a wave one after the other, rather than simply sending separate segments individually when it can. The reason is that, in order for the receiver to effectively estimate network congestion based on the successive segments reaching it, it needs some knowledge about the sender's pattern of transmission of these segments. While data segments are of fixed size, in any given implementation of the protocol the segment size can be set so as to optimize the average number of bytes that need retransmission, in line with the network's overall characteristics of burst errors, and so on. Similarly, the number of wave levels, and the number of segments comprising each wave level, can also be set with an eye to the application's message sizes, as well as the protocol's own internal need for wave "granularity" matching the network's range of congestion behavior (i.e. small waves containing few segments for transmission under significant congestion, through to large waves containing many segments in order to exploit opportunities when congestion is low).

2.2 Protocol Description

Our design currently has the fixed-sized segment headers set at 6 bytes, composed of the following fields:



- ***Source & Destination Port Numbers:*** Used for identifying the service access points for the higher-level (application) protocol.
- **Type:** A segment can be one of the following 12 types:

SYN

Used as the first step of the six-way connection-establishment handshake. Section 3.1 below details the connection-establishment phase, including the handling of exceptional conditions.

SYN_ACK1 & **SYN_ACK2**

Used during the connection-establishment phase. Once the connection is established, data flow between the two sides is full duplex. The level of the first wave sent is empirically determined based on the measured response times during the 6-way connection-establishment handshake (see Section 3.1 below).

D_WAIT

Used during the connection-establishment phase (see Section 3.1 below).

DATA

A data segment. Each data segment carries a fixed-sized payload, as well as Wave Control information and a Sequence Number in the header.

N-S_ACK

Negative Selective Acknowledgement. Each N-S_ACK carries Wave Control information. The payload field of a N-S_ACK segment carries an ordered list of 12-bit sequence numbers for segments that need retransmission, oldest segment first. Since this list can be of varying size, it is always terminated with the sequence number of the first *new* segment that the sender will/should be transmitting eventually. Because the number of segments at a given wave level is pre-determined, the receiver can ascertain the sequence number of that new sender segment, even if the terminating segments of the current wave or older waves have gone missing. The Sequence Number field is not used in a N-S_ACK segment.

PROBE1

When the sender receives a N-S_ACK specifying that the next wave level should be zero, it waits for an initial period of time (set to one second in our current implementation). It then starts a "*probe cycle*" by sending a PROBE1 segment carrying a unique identifier in the Sequence Number field, and initializes a SEND_T timeout (currently, one second) for that PROBE1 (see Figure 1). If, at the end of the initial waiting period, the sender has no data to transmit, it defers starting the probe cycle and continues waiting till it receives sufficient data from its higher-level protocol to at least make up a level 1 wave (which is the smallest wave in terms of number of segments). The sender expects a PROBE_ACK segment from the receiver in response to the PROBE1. If no such response reaches it by the time the SEND_T timeout triggers off, it sends a new PROBE1 segment with a new unique identifier number, and reinitializes the SEND_T timeout. It keeps doing this until it finally receives a PROBE_ACK.

PROBE2

In response to receiving a PROBE_ACK, the sender immediately sends a PROBE2 segment carrying the same identifier value as received in the PROBE_ACK, and reinitializes the SEND_T timeout, which will now be used for the PROBE2. It also ignores all PROBE_ACKs that might subsequently arrive. In response to the PROBE2, the sender expects a P-S_ACK from the receiver carrying Wave Control information for the next wave, which could again specify wave level 0. The payload field of the P-S_ACK identifies all old segments that need retransmission, if any, in exactly the same manner as in N-S_ACK segments. Receipt of that P-S_ACK marks the end of the probe cycle. If no such P-S_ACK is received within the SEND_T timeout period, the sender abandons the current probe cycle and initiates a new cycle with a PROBE1 segment carrying a new identifier as described above.

PROBE_ACK

When a receiver receives a PROBE1 segment, it immediately sends a PROBE_ACK carrying, in the Sequence Number field, the same identifier it received in the PROBE1 segment. It continues to respond to subsequent PROBE1 segments in this fashion until a DATA segment from the next wave reaches it. It thereafter ignores any further PROBE1 segments that arrive.

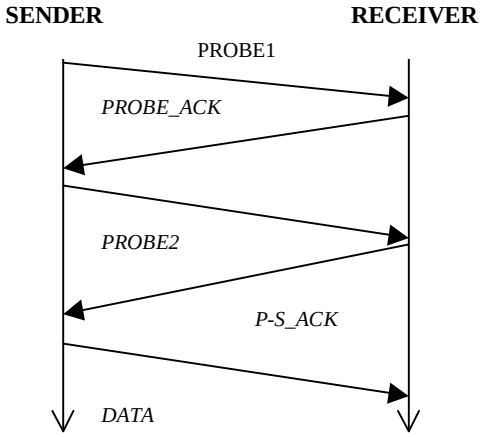


Figure 1: Probe Cycle

P-S_ACK

When the receiver receives the second probing (PROBE2) segment from the sender, it responds with a P-S_ACK. P-S_ACKs carry negative selective acknowledgements for missing segments, using the same format as N-S_ACKs. P-S_ACKs are distinguished from N-S_ACKs in order to avoid conflict in the event that the P-S_ACK probing response is delayed or lost, and the sender meanwhile receives a delayed N-S_ACK from the previous wave-connection. Such a delayed N-S_ACK would now be ignored by the sender. Note that PROBE1, PROBE2, PROBE_ACK, and P-S_ACK segments carry Wave Control information. All carry zero payloads.

FIN

Used by a side to initiate its two-way handshake connection-termination process.

FIN_ACK

Second step in the two-way handshake connection-termination process. Each side has to separately go through this two-way handshake before the connection can be gracefully closed. The connection-termination process is modeled on that of TCP, including the handling exceptional conditions.

- **Wave Control**: This 8-bit field is composed of a 2-bit *wave level* sub-field, followed by a 6-bit *wave-identifier* sub-field. The "wave level" can be 0, 1, 2 or 3: DATA segments always carry the level of the wave currently being transmitted in this sub-field; N-S_ACK and P-S_ACK segments carry the level that the receiver wishes to specify for the next wave; PROBE-type segments carry the value 0. The 6-bit wave-identifier sub-field carries an id number for the wave: DATA segments carry the id number for the current wave; N-S_ACK, P-S_ACK and PROBE-type segments carry the id number of the *next* wave to be transmitted.
- **Sequence Number**: Each DATA segment carries a sequence number. Clock-based initial sequence numbers (and wave identifier numbers) are exchanged between the two sides at connection set-up time, as in TCP. The use of the Sequence Number field in PROBE-type segments has already been described above.
- **Checksum**: This is a checksum field covering segment header, payload, and possibly also a pseudo-header as in UDP/TCP.

The receiver attempts to estimate prevailing congestion conditions by monitoring the throughput of the current wave and setting the level of the next wave accordingly. A wave at level *i* (*i*>=0) is composed of a fixed number *W(i)* of data segments. For *i*=0, *W(*0) is defined to be 0.

A DATA segment is composed of the 6-byte header and a fixed-sized data payload. Once the first segment to reach the receiver from a new wave arrives, it is easy for the receiver, given the current wave level *i*, to calculate how long it would take the rest of the wave to reach it if the network were relatively uncongested, using a "*baseline*" throughput estimate of *BT* Kbytes per second for the uncongested network. The time thus calculated is the "*baseline time*". The value of *BT* is initially determined during the connection-establishment phase, as described in Section 3.1 below. The receiver measures how long it actually takes for the remaining segments in the wave to arrive. If the throughput thus measured for the wave exceeds the baseline value *BT*, the receiver first resets *BT* to this higher throughput value and recalculates the wave's baseline time before proceeding. It then uses the baseline and measured times for the wave to set the level of the next wave. Our design currently calls for four wave levels, *i=0,1, 2, 3.* In the implementation of this design we set the number of segments in a wave

W(i) = (12 x i) for i=0, 1, 2, 3;

fixed-sized segment payload was set to 1KByte. The following simple algorithm was used by the receiver to set the next wave level:

Suppose the current wave is at level *i, i = 1, 2, 3.*

Let **T(i)** be the measured time for a level i wave.
Let **B(i)** be the baseline time for a level i wave.
For j = 1, 2, 3:
if **T(i)** is in the range **(j-1 , j)** X **B(i),**
then
next wave level is set to **(4-j);**
else
set wave-level to **0**.

The algorithm above essentially implies that the when the receiver sets the new wave level to ***k***, *k = 1, 2, 3* , it is estimating the current network throughput to be no worse than approximately a fraction *1/(4-k)* of the baseline throughput value of BT KBytes per second (and no better than approximately a fraction *1/(3-k),* for *k = 1 or 2*). The number of segments in the new wave is then adjusted proportionately. If the throughput appears to be less than 1/3 of the baseline throughput, we go to level 0, deeming it better to pause for a while than risk expending energy transmitting even a small wave that might not have a sufficiently good chance of getting through undamaged.

Since segments from the current wave might never reach the receiver, we implement a timeout, REC_T, on the receiver side so that it does not wait indefinitely for the rest of the wave to arrive. As mentioned above, if the receiver has set the current wave level at *i*, it is assuming that prevailing throughput is no worse than a fraction *1/(4-i)* of the baseline throughput. Given that the wave comprises *(12 X i)* segments, the time for the wave to be received should not be more than about 36/*BT* seconds. In our implementation we initialize REC_T to twice that value. When the first segment of the wave to reach the receiver arrives, the receiver initializes REC_T. If the wave does not fully arrive before the timeout triggers off, the receiver assumes that the missing segments are lost and proceeds sending a N-S_ACK, using the time elapsed till the timeout triggered off as the value for T(i), the measured time.

In the event the receiver sets the next wave level at 0, the sender will probe the receiver after some delay. The receiver uses the measured time between its receiving the PROBE1 and corresponding PROBE2 segments from the sender to determine the current RTT. It then uses this RTT value to estimate the prevailing network throughput, and sets the new wave level accordingly in the P-S_ACK segment it sends to the sender (see section 3.2 below).

The sender does not start transmitting segments until it has a sufficient number to make up a complete wave. When it receives a N-S_ACK or P-S_ACK setting the wave level, and if it does not have sufficient old (needing retransmission) and new data up to the specified number of segments in the wave, it will transmit the segments it has at the highest wave level for which it has enough segments, setting the wave-level sub-field in the headers accordingly. If it does not have enough segments to

make even a level 1 wave, it waits till its high-level protocol provides it with sufficient data.

Furthermore, if the sender's higher-level protocol asks it to close the connection and the sender still has a few segments left to transmit that do not come up a complete wave, the protocol transmits those segments it has. It immediately follows this up with a FIN segment that carries the sequence number of the last segment transmitted so that the receiver understands that this last and final wave is an incomplete wave, and responds accordingly.

A N-S_ACK from the receiver can be lost. Thus, when the sender finishes transmitting the last segment of the current wave, it sets a SEND_T timeout. If a N-S_ACK does not arrive before the timeout triggers off, the sender initiates a probe cycle. The Wave Control carried by the PROBE-type segments here consist of 0 for the wave-level sub-field, and the number of the next wave that sender would (eventually) commence in the wave-identifier sub-field (i.e., the same wave-identifier value that the missing N-S_ACK carried). If the N-S_ACK arrives after the probe cycle is initiated, it is ignored.

3. Connection Management

Connection Management in WWP is fairly complicated. For example, a TCP-like three-way handshake for connection establishment is not adequate since it would not permit the receiver to *both* acquire sufficient information to determine the appropriate level for the sender's first wave, *and* to communicate that information to the sender. There are, in fact, three phases during which the connection needs to be managed: for connection establishment; during the probing phase; and at connection termination. Connection termination is modeled on TCP's and will not be further dealt with here.

3.1 Connection Establishment

Figure 2 sketches the state transition diagram during connection establishment. An active open initiated by an application causes a SYN to be sent to the destination. The sender makes a transition to state SEND to wait for a SYN_ACK1. If this is not received within the set time, the sender retries once more. If, again, no SYN_ACK1 is received during the prescribed waiting time, this is taken as an indication of network congestion and, in compliance with the protocol's strategy of saving energy by trading time, connection establishment is postponed. On the other hand, if the SYN_ACK1 is received in time (sending the SYN_ACK1 causes the receiver to move to state LISTEN), the sender responds with a SYN_ACK1 of its own and transits to the REPEAT state to await a SYN response from the receiver. At the other site, the receiver responds with the SYN and moves into READY, waiting for a SYN_ACK2. If the SYN_ACK2 arrives within the receiver's waiting time, the receiver sends a D_WAIT. At this point it establishes the connection only if it determines that the appropriate wave level should be more than 0; otherwise it just returns to CLOSED. The sender receives the D_WAIT with wave control information and sends the first wave, or moves to CLOSED if the wave level is 0. If the D_WAIT is unduly delayed, the sender moves to CLOSED assuming inappropriate conditions.

Two Round-Trip-Time (RTT) measurements are taken during the connection-establishment phase at the receiver. Upon receipt of the SYN used by the sender to initialize the connection-establishment phase, the receiver sends a SYN_ACK1 and starts measuring the time for the first RTT. The measurement interval completes upon receipt of the sender's response (SYN_ACK1). The receiver initiates the second measurement interval immediately, when it sends its SYN in response to the SYN_ACK1. The second interval terminates upon receipt of SYN_ACK2.

In an error-free environment with no congestion, the two RTT measurements should be more or less equal. The receiver would then set the wave level in accordance with the experienced RTT, which would also be used to set the value of the baseline throughput *BT*. For the case in which the RTT is deemed too high to set an acceptable BT value, connection is abandoned. The application protocol can then initialize the connection establishment again, letting the application or the user decide whether, under such conditions, a connection should be established. A user may even set the desired wave level according to his preference (time or energy) although our protocol does not yet support this mechanism. In the event that one RTT is significantly different from the other, the receiver first checks that the more pessimistic (higher) RTT value does not indicate unacceptable congestion; if so, it cancels the connection. Otherwise, it would use the lower RTT value to set *BT* and set the wave level at 1. The algorithm below summarizes the situation.

---

if RTT1 = RTT2
  if *RTT1 < Threshold*
    set *BT*
    else
      abandon
else (if RTT1 <>RTT2)
  if *max[RTT1, RTT2] < Threshold*
    set Wave Level = 1
  set *BT* according to *min[RTT1, RTT2]*
    else
      abandon

There are several details about the connection-management strategy of WWP and the appropriate calibration of its mechanisms that should be discussed further. Although a detailed analysis of exceptional cases is beyond our purview, there are a couple of cases that ought to be mentioned. For example, the sender's timeout for the D_WAIT state should be quite large: at least as large as the timeout value that would cause a receiver in the READY state to make a transition to the CLOSED state. In addition, using an equally large timeout, the receiver in ESTAB mode moves back to CLOSED if no data is received (this could correspond to the case where the sender's timeout

expires because D_WAIT has not yet been received). Another exceptional case that was considered is when the receiver is forced to re-initiate his first RTT measurement because the sender timed out and repeated the SYN. Upon completion of both RTT measurements, and if they were to yield similar values below the threshold, the receiver would nevertheless set wave level 1. The logic here is that the sender's timeout might indicate higher congestion levels than the equal-valued RTTs might suggest.

3.2 Probing

The probing phase is somewhat similar to the connection-establishment phase. A 4-way handshake is used as shown in the state transition diagram of Figure 3. The receiver estimates the congestion risk-level using the Round Trip Time (RTT) measured between its transmitting a PROBE_ACK and receiving the corresponding PROBE2 segments, and then sets the next wave level. Only one RTT measurement is used here rather than two as in the connection-establishment phase because, by now, the receiver has gained some experience of the network's behavior, which could be used together with the RTT measurement in setting the next wave level.

4. Implementation & Testing

The protocol was implemented using the x-kernel [5, 8] protocol framework.
The high-level test protocol sends messages of 1024 bytes to the underlying WWP layer. These are then buffered until there are enough segments to form a wave at the level needed. The sender's buffer was set to 40 segments. Semaphore-based flow control was implemented between WWP and the test protocol at the sending side, so that the latter does not try to push new segments into a full buffer. The receiver's buffer was set to 256 segments so that it will not be forced to unnecessarily drop incoming segments during its selective-repeat mode of operation.

4.1 Testing Environment and Methodology.

We ran tests simulating a fairly low bandwidth environment. The tests were carried out in a single session, with both client and server running on one and the same host, so as to avoid unpredictable conditions with distorting effects on the protocol's performance. Congestion was simulated by dropping and delaying segments using the x-kernel protocols VDROP and VDELAY. VDROP deterministically drops segments at a fixed rate specified for the duration of a test. VDELAY holds (delays) each segment for an amount of time randomly distributed between 0 and the time value specified for the test, and independently chosen for each segment in turn. Since the action of x-kernel's VDROP is in effect for the entire duration of an execution run, we developed a second version that alternates On/Off phases during which the action of VDROP is in effect and is suspended, respectively. Thus, during a connection period, the protocol would experience phases that are error free and others with simulated congestion effects. This modification enabled us to test the protocol's behavior in response to sudden changes in the simulated environment, and its ability to rapidly re-adapt to varying congestion conditions. VDROP and VDELAY combined simulate different congestion conditions, while VDROP used just by itself may be viewed as simulating unreliable connections. Both protocols were configured above IP, with WWP configured on top of them, and our high-level testing protocol configured above WWP. We also ran TCP (Tahoe) under a similar configuration. Representative results from our tests are given in the table below.

We compare our protocol with TCP since TCP is a reliable protocol with end-to-end service similar to WWP. It is also a topic of current research interest with respect to its behavior in wireless environments. However, TCP does not distinguish well between congestion, on the one hand, and transmission burst errors, on the other, although each requires distinct actions in response to its occurrence (e.g. slow down, and feed the network, respectively). TCP's optimization capability is well known (e.g. [1], [7]), and implementation problems can be predicted or avoided altogether [4], making it a good standard of comparison.

Note that our WWP implementation makes no attempt to calibrate the various protocol parameters (data segment size, number of segments per wave at each wave level, timeout values, etc.) for optimal performance with respect to the overall characteristics of the protocol's operational environment. The segment payload was 1Kbyte; $W(i) = 12\ X\ i,\ i = 0,1,2,3$; SEND_T = 1 second; REC_T = 2 seconds; *BT* was assumed fixed at the rather conservative value of 40 Kbyte/second, and was not increased when the receiver detected higher throughput. All this probably causes WWP to understate its potential.

In order to represent the protocol overhead rate required to complete reliable transmission under different conditions, we used the formula:
`Effort Overhead = (Total - Effective) / Effective,`
where,

- `Effective` is the number of bytes delivered to the high level protocol at the receiver, and is shown in the column **Original Data** of the table;
- `Total` is the sum of the number of bytes transmitted by the transport layers at the sender and the receiver, which are given in the columns **Sender Total Bytes** and **Receiver Overhead**, respectively.

While the formula gives a measure for the overall effectiveness of a protocol, where energy saving in particular is concerned **Sender Total Bytes** gives a measure of the energy expenditure for the sender, and (**Original Data** + **Receiver Overhead**) similarly for the receiver.

Entries in the **Original Data**, **Receiver Overhead**, and **Sender Total Bytes** columns are based on 1-minute snapshots. For VDROP with On/Off phase behavior, On and Off phases were of equal duration; this phase duration is given in the column **VDROP Phase** in which the entry *Always* means that VDROP was on throughout. The

VDROP Rate reported is the dropping rate for the On phases, not the averaged overall rate across On/Off phases.

4.2 Analysis of Protocol Behavior.

WWP is designed for low-bandwidth, error-prone environments with unstable characteristics. Energy saving is a major concern when the protocol is used for battery-powered devices.

Energy: As demonstrated by test sets 1, 2, 4, and 5, the throughput achieved by WWP in a problematic environment in which congestion is the cause of packets being dropped, is 6-10 times more than the throughput achieved with TCP. For an equal amount of data to be transmitted, TCP would require up to 5 times longer. During this time, TCP transmits relatively aggressively - depending on the environment - and so increases the actual transmission time, thereby expending more energy. While comprehensive and extensive results are not being reported in this paper, the tests above clearly demonstrate the validity of our statement. In addition, it is notable that the effectiveness of the protocol is very high. The overhead needed for completion of the transmission is much lower than TCP.

Behavior under continuous congestion: In contrast to the behavior described above for situations of adequate bandwidth, the protocol becomes very conservative when problems are detected. As shown in the 3rd test set, under a consistently high dropping rate WWP delivers half as much bytes to the application as TCP, but at comparable effort overhead. This results in less overall data transmission, and so energy is saved for later, possibly better, conditions when transmission can take place at lesser energy cost. Both protocols detect congestion in an intelligent manner, and their effort overhead is similar, but the decision as to whether to transmit or back off differs. A duality exists between WWP's "relaxed" attitude to transmission in the presence of congestion, on the one hand, and the delay tolerance of low-priority traffic in the presence of higher-priority traffic, on the other. As such, WWP can be adapted for low-priority applications in other network environments (e.g. wired).

Behavior under periodic congestion: When congestion appears on a periodic basis, WWP attempts to back-off and save energy while continuously probing. As soon as available bandwidth is detected, WWP aggressively transmits data. There is benefit to both throughput and effectiveness as shown in test sets 1, 2, 4, 5.

Behavior in unreliable channels with available bandwidth: WWP possesses the important ability of distinguishing between congestion, and transmission burst errors, due to an unreliable link or a low frequency channel. When congestion is the problem, the probe does not come through, or it comes through delayed. The duration of such conditions vary. Unlike congestion, an error caused by a lossy network is transient. WWP detects this by probing: when the probe reaches the destination with acceptable RTT characteristics, the protocol's decision is to send waves accordingly. In an environment with relatively infrequent errors occurring in a stably-sustained pattern, WWP is not aggressive. TCP also adapts itself quite well. However, when the error pattern becomes unstable and varying, TCP is unable to readjust itself, and wastes enormous amounts of energy and time to transmit the required data, behaving as if the problem were congestion. WWP, on the other hand, is able to detect opportunities of error-free conditions, and takes advantage of them using appropriate wave levels. It keeps transmitting at high rates. This is demonstrated by the 6th and 7th sets of tests.

5. Conclusion and Future Work

We have presented a protocol that trades connection time for energy saving. Our results clearly demonstrate that network conditions can be effectively detected, and retransmissions and duplications can be avoided. In a sender - receiver context, the receiver passes control information to the sender which adjusts the data-wave length accordingly. The risk of congestion is estimated at the receiver and the higher this risk, the lower the wave level attempted. When the risk is too high, the sender remains idle, periodically probing the receiver for updates on congestion level. The total number of bytes transmitted is not significantly greater than the application's original message. Comparison results at different congestion levels demonstrate suitable behavior by the protocol. In contrast to TCP, totals for overall bytes transmitted are significantly less, so energy saving is proportionally higher. As mentioned above, the protocol has not yet been optimized. Results from the existing version of the protocol already clearly demonstrate that significant time/energy tradeoffs are being tapped.

Acknowledgements. We gratefully acknowledge Peter Konecny for his contribution and help in coding the x-kernel-based implementation of the protocol.

7. References

1. M. Allman, V. Paxson, W. Stevens, "TCP Congestion Control", RFC 2581, April 1999
2. R. Marasli, P. Amer, P. Conrad, "An analytic Study of Partially ordered Transport Services" *Computer Networks and ISDN Systems*, 1998
3. R. Marasli, P. Amer, P. Conrad, "Partially Reliable Transport Service" *ISCC '97*, Alexandria, Egypt, July 1997.
4. V. Paxson, et.al., "Known TCP implementation problems", RFC 2525, March 1999.
5. L. Peterson, B. Davie, "Computer Networks: A Systems Approach", Morgan Kaufman, 1996
6. Wei S. and V. Tsaoussidis, "An Application oriented Transport Protocol for multimedia communications over IP", *IEEE Conference on Computational Intelligence and Multimedia Applications*, ICCIMA '99, September 1999.
7. J. Touch, John Heidemann, Amy Hughes, "Issues in TCP Slow-Start Restart After Idle", 04/10/1998, <draft-ietf-tcpimpl-restart-00.txt>
8. The X-kernel: www.cs.arizona.edu/xkernel
9. 3COM, Technical Documents: "Palm OS 3.0 Documentation " www.3com.com, April 1998

Appendix: Figures and Tables.

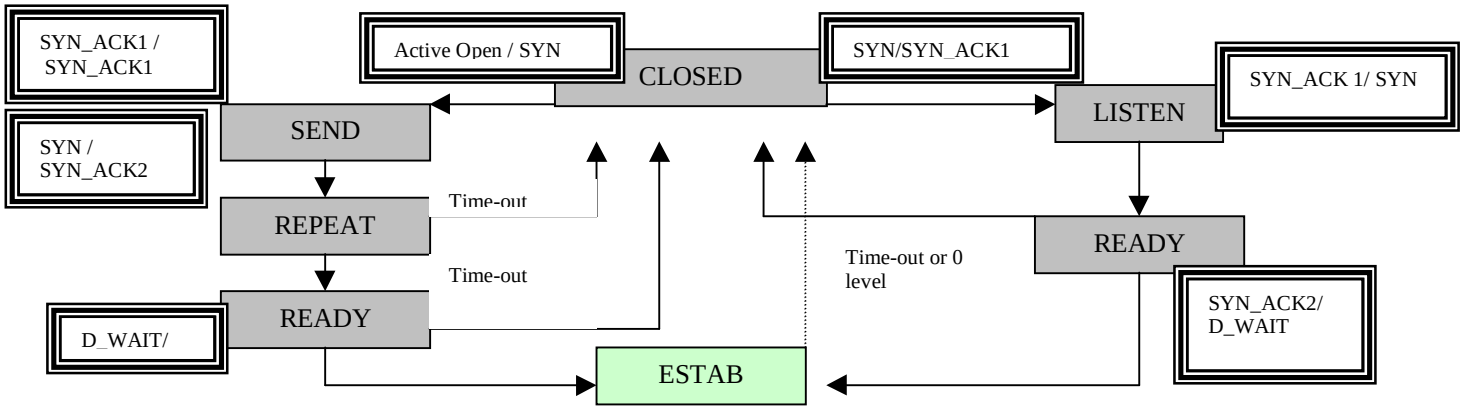


Figure. 2: Connection Establishment State Diagram

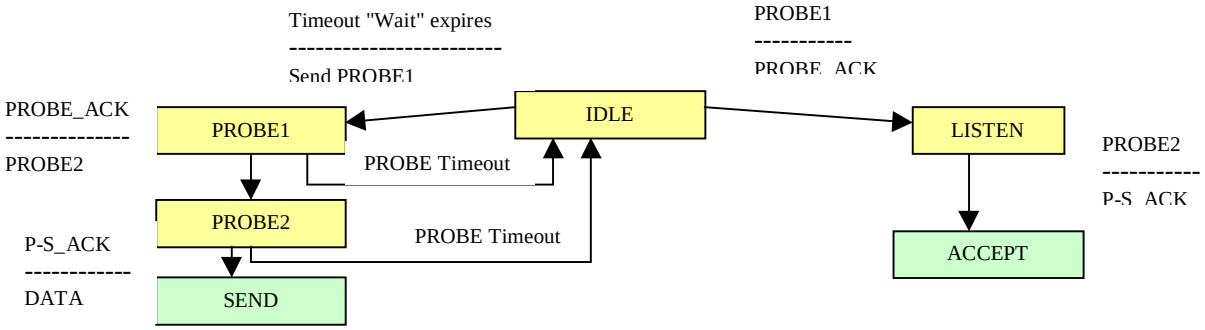


Figure 3: Probing: State Diagram

Test #	Protocol	VDELAY	VDROP Rate	VDROP Phase	Original Data	Receiver Overhead	Sender Total Bytes	Effort Overhead
1.1	TCP	50ms	20%	10sec	881,996	15,928	936,202	0.08
1.2	WWP	50ms	20%	10sec	6,387,712	1,617	6,507,656	0.019
2.1	TCP	50ms	20%	2sec	811,096	14,828	861,428	0.08
2.2	WWP	50ms	20%	2sec	5,551,104	1,662	5,803,805	0.046
3.1	TCP	50ms	20%	Always	333,230	6,068	425,712	0.296
3.2	WWP	50ms	20%	Always	143,360	590	185,645	0.299
4.1	TCP	50ms	33.33%	10sec	506,226	9,128	530,686	0.066
4.2	WWP	50ms	33.33%	10sec	5,408,768	1,459	5,556,645	0.028
5.1	TCP	50ms	33.33%	2sec	479,284	8,628	530,686	0.125
5.2	WWP	50ms	33.33%	2sec	5,581,824	1,789	5,816,213	0.042
6.1	TCP	0	5%	Always	1,024,200	14,908	1,093,168	0.081
6.2	WWP	0	5%	Always	1,119,232	763	1,185,978	0.060
7.1	TCP	0	5%	2sec	1,024,000	14,628	1,044,276	0.034
7.2	WWP	0	5%	2sec	23,012,352	4,694	23,189,859	0.007

Table 1: TCP and WWP test results