Real time communications over UDP protocol

Michael Pan, 15 Nov 2011 CPOL

Introducing UDP-RT protocol for efficient and reliable communications in real time systems

Introduction

Fast and reliable communications is a basic requirement in almost all modern applications, but the real time systems take it to extreme and require real time responses from the network. In some cases, the requirements are so tough that may require special hardware to achieve desired performance. During the last decade, new technologies have emerged to support the growing demand, but their prices are high and availability is low. As a result, most of the systems are still using TCP/IP protocols stack over Ethernet backbone. Moreover, most of the programmers are still developing with well-known synchronous sockets API. This article explores usage of these dominating technologies in the real time systems and shows how to design and develop communications within such systems. In particular, it proposes reliable protocol over UDP transport layer, that gains less than one millisecond average round trip time (RTT) and processes dozens of thousands of messages every second at each and every port in network.

Background

Real time networking is a non-easy task, so before move any further, system connectivity requirements must be analyzed. The right way to do it is to map and classify traffic in the system. First classification to be made is messages sizes and their latency requirements. In many cases, these two parameters are sufficient to choose transport protocols and network topology.

Let us start with the transport protocol choice. Since this article concentrates on TCP/IP, our choice will be between TCP and UDP. The table below proposes protocol selection rules:

Table 1 (Protocol Selection Rules)

Latency / Messages Length	Few KB or less	Dozens of KB or more

1 ms (*could be less*) to dozens of ms	**UDP**	**?**
Hundreds ms or more	**TCP**	**TCP**

There is a question sign in low latency and large messages cell in the table above. In this specific case, we should consider different approaches, such as RDMA over Infiniband™, but these technologies are not in scope of this article.

TCP

Long messages may contain database tables, pictures, videos, bunches of parameters and other types of data which must be moved around the system without compromising its performance. The TCP protocol is dealing quite well with big amounts of data. Long messages allow TCP to warm up its engines and find best fit for sliding window to avoid retransmits and achieve good utilization of bandwidth. But as latency requirement become tighter, the TCP protocol exposers its weaknesses.

First, the TCP latency is not deterministic. Consider server with multiples clients. Clients are sharing the same link and thus their traffics affect each other latency. Given the fact that the TCP protocol is lacking fairness and quality of service features, the latency variation may exceed hundreds of milliseconds. Moreover, in case of multiple clients, the bandwidth utilization may significantly drop down (in legacy systems it may collapse to 50%).

Another problem that TCP suffers from is slow recovery process. Regular recovery for a lost TCP fragment may take hundreds of milliseconds. There are some tricks which allow reducing recovery time. For example, in Windows XP™, configuring of `TcpAckFrequency` and `TcpDelAckTicks` parameters in registry achieve 200ms saving in recovery time. Unfortunately, these settings are not available in all operating systems and even if they were, there are other TCP timers and algorithms which keep the recovery time high.

Short messages are special challenge for the TCP. By default, TCP uses Nagle algorithm to accumulate short messages and send them together. The algorithm may delay packets sending and, thus, affect their latency. Sockets provide an API to disable this behavior (see `setsockopt` function documentation), but legacy sockets implementations may not have this functionality.

Concluding the above, TCP is a great protocol, but it is not suitable for real time communications. As it has been stated in the table 1, it could be successfully used only in systems with latency above hundreds of milliseconds.

UDP

Short messages in real time systems are usually used for managing hardware devices, data auditing and closed-loop communications (controlling device behavior based on the data that the device produces in real time). Due to the real time nature of the data, short messages must be delivered and reach their destination within few milliseconds. The transport protocol that may stand within this latency is UDP. Indeed, observe round trip time (RTT) of message from user space of its source to the user space of destination and back. The RTT of a message over UDP protocol highly depends on hardware and on operating systems of communicating peers. Nevertheless, even 100Mb network and non-real time OS, such as Window XP™, provide average RTT of one millisecond. Usage of real time OS and more advanced network cards may reduce

it to microseconds.

Of cause, UDP is non-reliable protocol, so to use it we must add reliability features at the application level. This is complicated task, but if you have a fair big system, you should have the UDP alternative implemented. There are numerous UDP based protocols, which provide reliability feature (RDP, for example), but none of them has been found suitable for real time communications in heterogeneous systems. This article proposes new reliable protocol over UDP, especially designed for such systems.

Usually, when someone is asked about network requirements, an answer is given in terms of bandwidth. This answer is insufficient for the real time systems. As we have already seen, traffic between peers could be classified by its latency requirements and messages sizes. For short messages, there are two more important parameters that describe the traffic: number of messages and their distribution.

Consider a 1Gb link that is used for sending and receiving of short messages only. In theory, 1Gb card should be able to process over 2.000.000 packets per second (minimum Ethernet frame size is 64 bytes), but in practice, it is far from being feasible. Even if some particular network card will be able to process this amount of packets, it will be a real challenge to process them in operating system and in application. For example, 2 GHz processor will grant at most 1000 cycles for a packet, which is very tight number for the task.

Load distribution of messages should be taken into account as well. For example, if several clients are highly synchronized while sending packets to server, loosing packets at the server side becomes more likely. This, in turn, will lead to packets retransmissions and latency raise.

Network planning

Once we have finished to classify traffic in the system and chosen transport protocols, we should turn to network planning. We need to examine the existing network topology (or build one if it does not exist yet) and verify that it can handle traffics requirements. Usually real time systems have a dedicated network for real time communications. Still, this network could be shared between different modules within the system and these modules may have different performance requirements. The network backbone must be strong enough to handle communications as required by all the modules. To simplify our discussion, we assume that a proper network backbone exists in the system and our objective is resolving its utilization.

The next step in network planning is separating different types of traffics. For instance, TCP and UDP should not be messed. It is also preferable to separate UDP communications with different latency requirements. By separating the traffics we decrease density and, thus, clashes of packets. Is simplifies network analysis as well. Of cause, the best separation is achieved by creating dedicated physical connections for every traffic type (separated network cards and separated wires between peers). Nevertheless, in many cases such a separation is not possible due to budget and hardware limitations. In this case we can try to separate the traffics in time. If this is not possible as well, then we must accurately benchmark our communications and, depends on findings, consider implementing a propriety protocol that will satisfy the network requirements.

UDP-RT protocol

The UDP-RT protocol is the UDP based protocol for real time communications that allows sending short messages with low latency and provides protocol reliability features. Formally, the UDP-RT protocol should

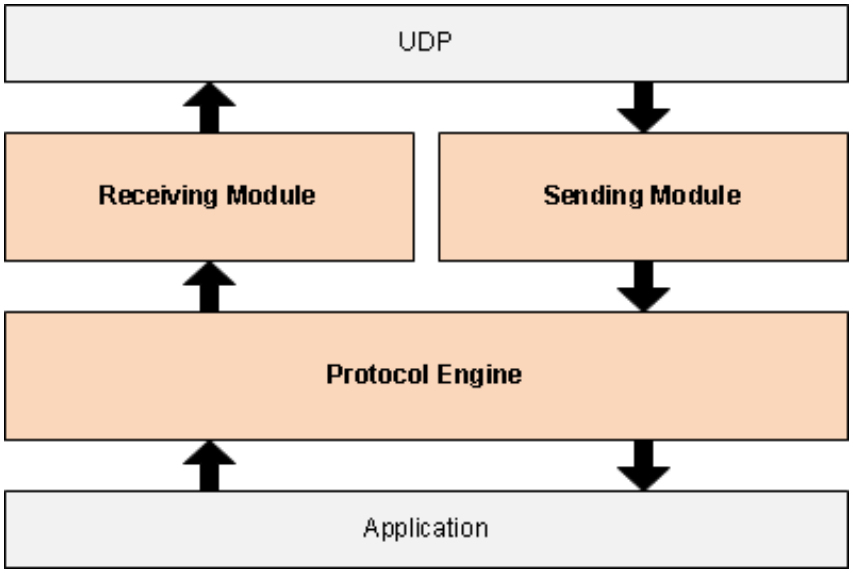
fulfill the following requirements:

1. **Low latency**
    - Send application message to its destination as soon as possible
    - Receive and deliver message to application as soon as possible
1. **Reliability**
    - Avoid losing messages
    - In case of lost message, detect and recover it (time that takes to recover a message is related to both reliability and low latency requirements, so the protocol should pay a special attention to this issue)
    - Handle messages reordering

Along with the above functional requirements, it is also important to keep protocol simple. Modules in the real time systems could run embedded software, which is hard to debug. So we gave up, on purpose, of some advanced features for the sake of design and implementation simplicity. Nevertheless, for those of you who are more interested in robustness rather than simplicity, we will discuss possible extension of the protocol in the end of the article.

The UDP-RT suite is divided into three main modules: sending, receiving and protocol engine (see figure 1). The first two are responsible for efficient sending and receiving of messages. These modules provide customization abilities to optimize protocol performance at a hosting OS. The protocol engine covers reliability requirements from the list above.

Figure 1 (UDP-RT modules)

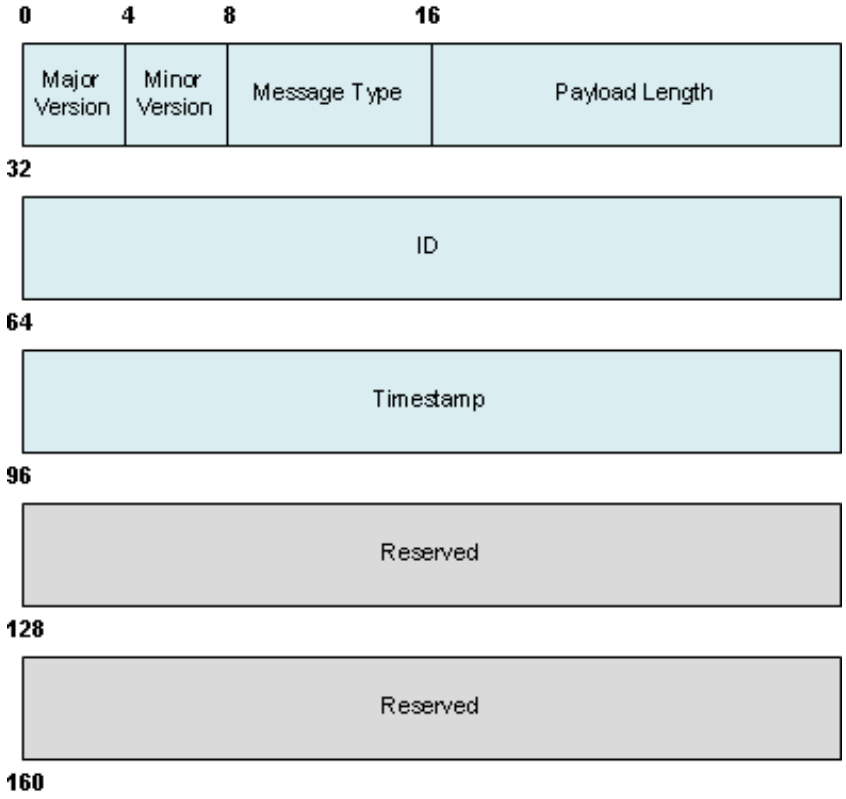


Message structure

Every UDP datagram may carry several messages of UDP-RT protocol. Messages cannot be split over UDP datagrams, so the maximum message size is limited by maximum datagram size (64KB). A UDP-RT message

consists of header and payload. Figure 2 shows message header layout.

Figure 2 (UDP-RT message header layout)



- ***Version*** fields allow the protocol to grow up. In future versions new functionalities could be added and the version fields will help peers to determine their compatibility. Recent version is 0.0.
- ***Type***: messages could be of type `DATA(0x0)`, `ACK(0x1)` and `RESET(0x2)`. The DATA messages are carrying payload. The RESET messages reset messages' counter. The ACK messages are used to acknowledge the arrival of the DATA and RESET messages. The RESET and the ACK messages carry no payload.
- ***Payload length*** defines the number of bytes in the message payload.
- ***ID*** is a message serial number that sender assigns to DATA and RESET messages. The ACK messages must copy the ID from the corresponding DATA and RESET messages.
- ***Timestamp*** contains sending time of DATA and RESET messages and filled in by message sender. The ACK messages must copy the timestamp from the corresponding DATA and RESET messages.

Channel settings

A pair of peers that uses UDP-RT protocol is considered as UDP-RT connection and called *channel*. Every channel has its own set of settings, which defines the UDP-RT behavior on the channel. Below is a list of available channel configuration settings.

- ***Initial retransmission timeout*** – Default value of retransmission timeout. See "Message retransmission timer" section for details.
- ***Retransmission timeout model*** – Algorithm to be used by protocol engine for calculating retransmission timeout value. See "Message retransmission timer" section for details.
- ***Maximum message delay*** – Maximum time that application permits for message passage. See "Message retransmission timer" section for details.

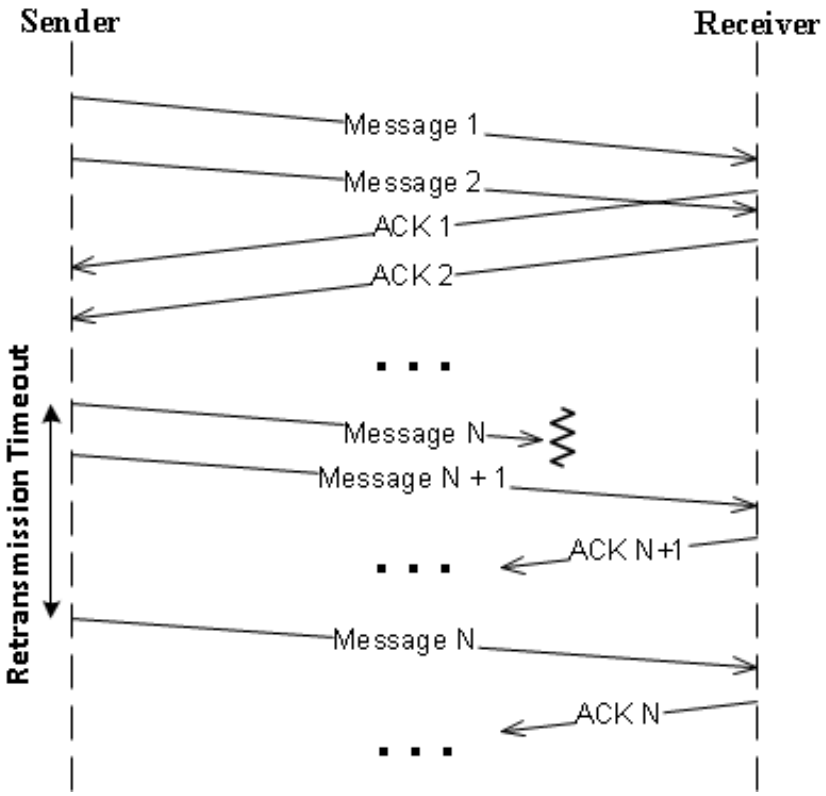
- ***Message dropping policy*** – Defines how to handle messages for which "maximum message delay" expires. See "Message retransmission timer" section for details.
- ***Out-of-order messages policy*** – Defines whether to allow out-of-order messages at the channel. See "Messages reordering" section for details.
- ***Reset waiting time*** – Time to wait during reset sequence. See "Resetting messages numbering" for details.

Protocol Engine

Lost message detection and recovery

The idea is quite simple. In order to know if message has arrived to its destination, it must be acknowledged by the receiver. The sender is storing the message in its memory as long as it is waiting for acknowledge. Once it gets acknowledge, the message can be discarded. In case the message is not acknowledged during some period of time, it must be retransmitted. Figure 3 visualizes the concept.

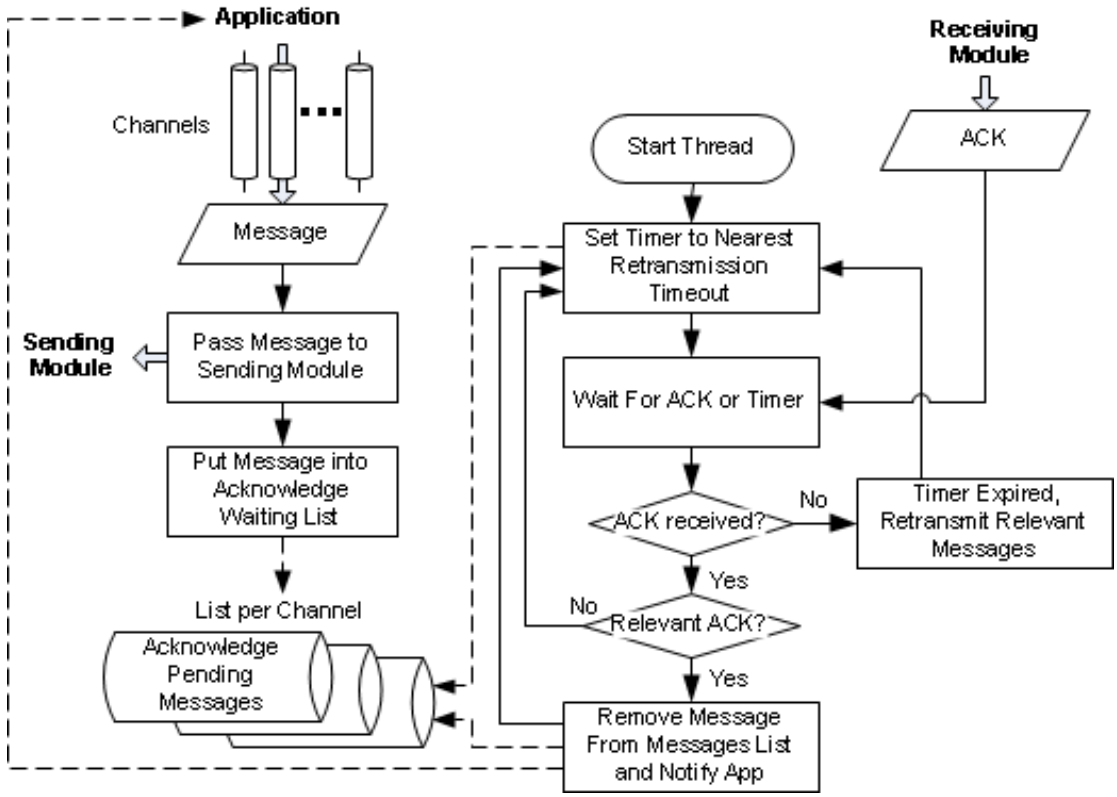
Figure 3 (Message retransmission)



When the sending operation is performed by application, the protocol engine does the following. First it assigns ID to the message and passes it to the sending module which, in turn, transmits the message to its destination. Then it adjusts retransmission timeout to the message and pushes it into queue of acknowledge pending messages (there is queue per channel). Whenever retransmission timeout of the message expires, the protocol engine retransmits the message (hands it over to the sender module) and sets a new timeout for the message. If acknowledge is received, the protocol engine is notified about it by receiver module. It examines the acknowledge pending messages and if acknowledge ID matches one of them, it removes the

message from the queue and notifies application about successful delivery of the message (if application requested such notification). Figure 4 shows flow chart of outgoing messages processing in protocol engine.

Figure 4 (Sending messages flow chart)

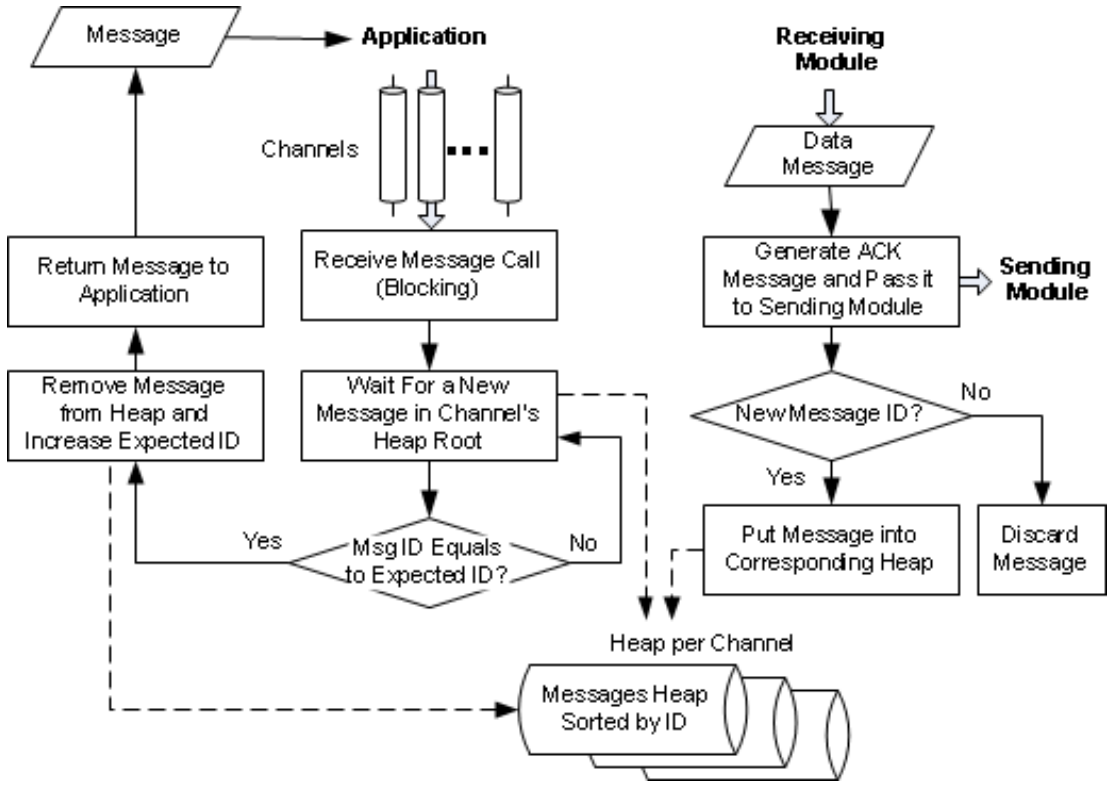


Messages reordering

The receiver module passes received messages to the protocol engine. If this is an acknowledge message then it is processed as described in the previous section. If this is a data message then the protocol engine must acknowledge it. So it generates acknowledge message and passes it to the sender module. Then it stores the received message till it is requested by application.

The protocol engine may get messages out of order because of retransmissions and routing (the last is less likely in closed real time system). So the engine put the messages into channel heap, ordered by messages ID-s. The engine knows what message is about to come next, so it should not pass out-of-order messages to application, but rather wait for the next in-order message. Of cause, some applications may do not care receiving messages out of order, so this behavior is configurable (*out-of-order messages policy* setting). The flow chart of incoming messages processing in protocol engine is shown in figure 5.

Figure 5 (Receiving messages flow chart)



Message retransmission timer

Calculating timer timeout

The UDP-RT protocol defines *maximum message delay* as a time that application permits for message delivering. Message latency is the actual time that it takes to transmit message; its value depends on network hardware and peers operating systems behavior. We assume that the maximum message delay is significantly bigger than average message latency and there is enough time for retransmissions. As a result, the timeout value of the retransmission timer becomes the most important parameter in the protocol configuration. To set a reasonable retransmission timeout, sender calculates average message RTT. It writes timestamp into message header and receiver copies the timestamp into corresponding ACK message, so the sender can calculate the RTT upon acknowledge receiving. Given the average RTT, the retransmission timeout could be set to `average(RTT)+ε`, where `ε` is a compensation for RTT jitter. Since the distribution of messages RTT-s could be considered as normal, the `ε` could be set to `2*σ`, which covers over 97% of RTT-s. While average RTT and `ε` can be calculated during protocol work, the initial retransmission timeout must be set by application (*initial retransmission timeout* setting) and its value should be determined by benchmarking or theoretical computations. We refer to the timeout model that has been described above as *average RTT retransmission model*.

The average RTT retransmission model works well as long as number of channels is relatively small and losses of messages are sporadic over time and channels. Indeed, if some random message is lost, the protocol will retransmit it after `average(RTT)+ ε` timeout. Now consider a situation where message is lost over and over again. Once it exceeds its maximum message delay value, the protocol should not attempt to retransmit the message and, instead, proceed according to the *message dropping policy* setting. The setting allows application to configure the UDP-RT behavior for expired messages; the protocol can either silently discard these messages or declare system failure.

Congestion control

The average RTT retransmission model becomes dangerous as packets' density is rising. Consider multiple clients send messages to server. There could be a spike in network activity which may lead to significant amounts of dropped and delayed messages. As a result, clients will try to retransmit the messages after expiration of retransmission timer. If the load is not over yet, the retransmitted messages will create even more load in the system, so more messages will be dropped or delayed. In the end, due to continuous load, some messages may reach their *maximum message delay*. To handle this scenario, it is crucial to understand that message timer expiration is an unlikely incident which is a result of momentary load. So, to prevent network flooding, we need to relax the network from retransmits and let the load to pass by. This could be done by increasing retransmission timeout for subsequent retransmits; for example, by increasing the timeout twice every retransmit. We refer to this timeout model as *exponential RTT retransmission model*. Pay attention, the UDP-RT protocol does not have abilities to handle continues load in the system and to adjust messages sending rate as TCP does. Instead, the protocol assumes that the system should work with no problem under normal conditions and only gives the system a chance to get back on track after an incident.

You may ask why not to use stop-and-wait approach and delay messages sends till receiving acknowledge for the recent message. Although this technique prevents network overflow, it has its own drawback. It may create unnecessary delays in messages sends and the delay may even become cumulative, unless the rate of sends is significantly smaller than average RTT time.

QoS

The UDP-RT protocol does not provide QoS mechanism. Instead, it offers a best effort to keep the average of messages delivery times below the maximum message delay. This is certainly true as long as there are no lost messages and the RTT is smaller than maximum message delay. Once messages are lost or delayed, the retransmission mechanism starts working and, if there are several channels that share the same link, it could be logical to prefer one channel over another (for example, there is a channel with stricter maximum message delay than in other channels). In other words, some channel may require faster message recovery than others. This could be done by assigning to the channel more aggressive retransmission timeout model. For instance, assigning to the channel average RTT retransmission model, while the rest of the channels are using exponential RTT retransmission model. To decide of proper timeout models, you will probably need to benchmark your system. We will discuss more strict solutions for QoS in protocol extensions section.

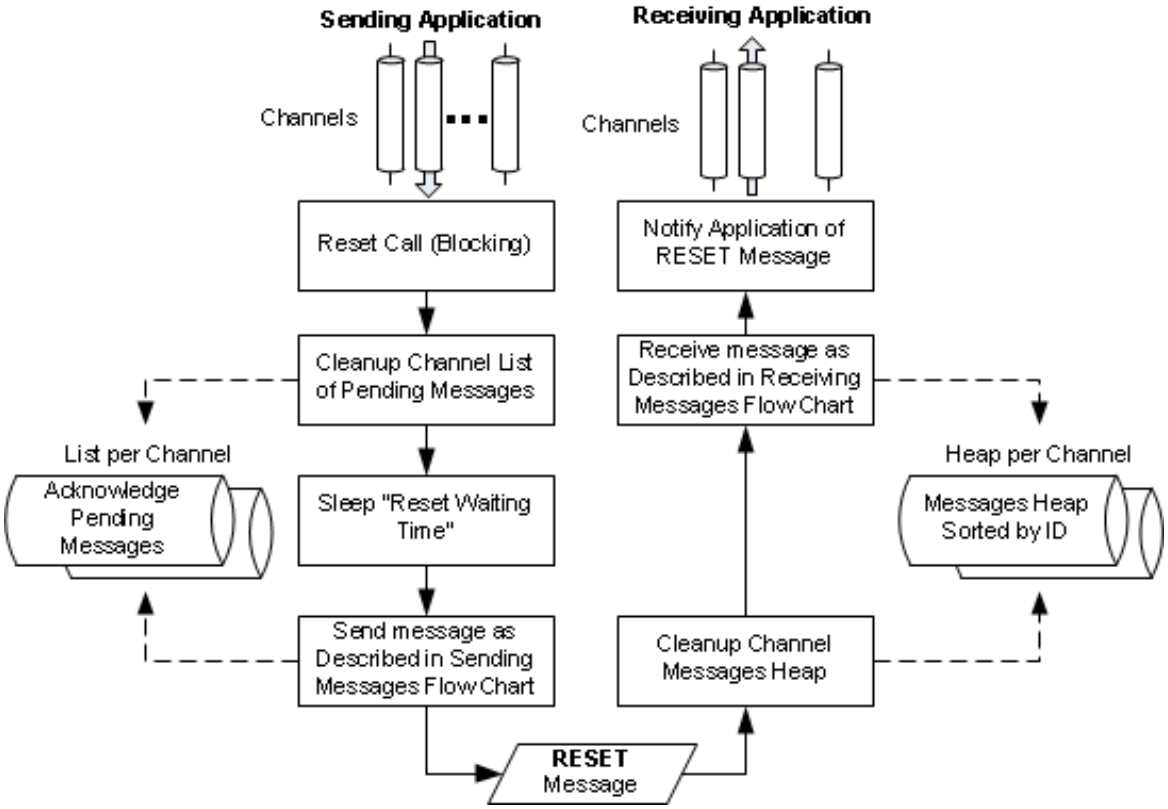
Resetting messages numbering

By default, sender is starting DATA messages ID-s from zero and assigns ID-s to the following messages in increasing order. The receiving side propagates the ID along with the message to the application. The application may reset the counter to any value any time. The reset feature is usually used by application to start new processing sequence or prevent messages ID overflow.

When application asks to reset ID counter, the protocol engine discards all acknowledge pending messages and waits enough time to allow the system to clean itself up from already sent messages. The waiting time is defined by *reset waiting time* channel setting. Then the sender issues RESET message. The receiver cleans up the heap of received messages and resets receiving ID counter to the requested value. The RESET message is acknowledged in the same way as DATA messages. The RESET command is also propagated to receiving application, so the application could properly handle the RESET notification. For example, it may request RESET from its protocol engine and, thus, reset traffic in both directions on the given channel. The flow chart

of the reset sequence is shown in figure 6.

Figure 6 (RESET flow chart)



Receiving and sending modules

The receiving and sending modules are in charge of adapting the UDP-RT protocol to the hosting OS. These modules are running in dedicated threads and operating sockets. In this section, we are going to discuss different aspects of these modules implementation and how to tune their performance.

Sockets programming

Recall that the protocol is working with synchronous sockets. Synchronous sockets are inefficient, but inefficiency is the price to pay for portability. Receiving and sending modules are mostly using three sockets API-s: `select`, `recvfrom` and `sento`. Indeed, the receiver and sender threads procedures could be described as following:

Hide Copy Code

```
Recever_Thread_Procedure() {
      while(true) {
            select(FD_SET);
            foreach(s in FD_SET) {
                  recvfrom(message, addr);
                  notify_protocol_engine(message, addr);
            }
      }
}
```

Hide Copy Code

```
Sender_Thread_Procedure() {
      while(true) {
            wait_for_msg_from_protocol_engine(message, addr);
            sendto(message, addr)
      }
}
```

Let us examine the performance of the sockets API-s with respect to the pseudo-code above:

> **`select`** – It is usually an expensive API and, in certain conditions, its execution may take dozens of milliseconds. To reduce number of calls to the API, every channel has its own socket for communications with its peer, so the `select` API could be applied to a large group of sockets at once. The `select` performance may be also affected by number of threads that simultaneously call it. In order to handle this, the receiving module allows to define number of receiving threads and assign to every thread its own group of sockets.
>
> **`recvfrom`** – After the `select` has determined sockets with datagrams, the receiver thread calls to `recvfrom` API to retrieve them. If there is datagram waiting in the socket's buffer, the `recvfrom` usually returns immediately. If this is not a case in some operating system, then `recvfrom` execution time may become a major latency limitation for the UDP-RT protocol.
>
> **`sendto`** – This API is usually fast, but under certain conditions its performance may deteriorate. For example, in Windows XP, calling `sendto` simultaneously from several threads may significantly slow down API performance (even though the calls are performed on different sockets). For this reason, the sender module provides a mechanism for defining number of sending threads and also reducing number of `sendto` calls by uniting messages into one datagram.

Another aspect to take into account is sockets configuration; for instance, sending and receiving buffers sizes. In case the sending buffer is too small to contain all outstanding packets, the next `sendto` operation on the socket will block till the buffer is freed. Similar problem exists with receiving buffer, if receiving socket buffer is filled up, the incoming packets will be dropped. As we have mentioned above, every channel has its own socket, so these situations are unlikely to happen. Nevertheless, if there is a problem with the buffers, you may call `setsockopt` API to enlarge them. This way, you will let more unprocessed packets to pile in the buffers.

Receiver threads

As it has been mention above, the receiver module allows running of multiple receiving threads. To choose an optimal number of receiving threads, many parameters should be taken into account: `select` and `recvfrom` performance, number of CPU-s and cores, number of network ports, number of channels and channels' latency requirements. Even though the right way to handle this challenge is system benchmarking, there are some simple guidelines. The receiver thread performs `select` on multiple, then processes all signaled sockets one by one and dispatches messages to protocol engine. Messages retrieving and dispatching should not create a delay, so the thread should have enough CPU power for these tasks. So, as a start point, it is recommended to create as much receiving threads as CPU cores in a given computer.

Sender threads

Similar to the receiving module, the sending module allows to tune number of sending threads. The optimal number of sending threads should be determined by benchmarking, but, since the sending thread does no processing that requires CPU, it is recommended to start with one thread only.

In addition to sending threads tuning, the module should also handle messages uniting. Every sending thread manages one main queue for sending requests and also queue per channel for outstanding messages. Protocol engine puts messages directly into channels' queues and also pushes sending requests into main thread queue. Each sending request contains channel identifier, so when sending thread pulls a request from the main queue, it also knows to pull corresponding message from channel's queue. If sending thread starts accumulating delay in sends, then the channels' queues start to grow up as well, so when the sending thread serves request for a channel with multiple messages in its queue, it unites them into one datagram. This way the sender thread reduces number of calls to expensive `sendto` API and also gets a chance to catch up after an instant delay in the system. Pay attention that the sending module unites messages only as a last resort, to overcome performance bottleneck, and not as preemptive action, like TCP Nagle algorithm does.

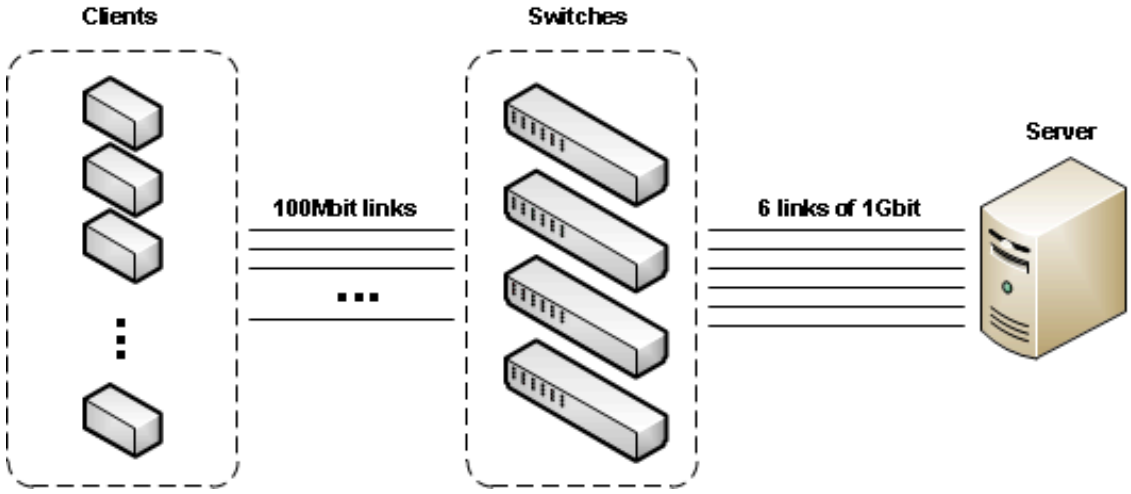
Interrupts moderation

One of the problem that UDP-RT was designed to deal with is big number of packets running in the network. Even if protocol engine, receiving and sending modules are properly configured, big number of packets may be a serious problem for hosting operating system. For example, in Windows XP™, every UDP packet fires an interrupt upon packet receiving and operating system creates deferred procedure call (DPC). The problem is that Windows XP™ processes all the DPC-s at the same core, so given a big number of incoming packets, the operating system becomes a serious bottleneck in packets processing. Luckily, modern network cards allow to moderate interrupts that they create. So if you experience a slowdown in UDP-RT, analyze interrupts performance (for example, by running Thesycon® DPC Latency Checker) and use interrupt moderation if needed.

Case study

To benchmark the UDP-RT protocol, a special setup has been built. The setup emulates client-server system and consists of over 700 clients with 100MBit Ethernet network cards, network switches and Intel® based server with 2 CPU-s (4 cores each) of 2.33GHz. The clients are connected through several cascading switches to 6 network cards of 1Gbit at the server side. The setup schema is shown in figure 7.

Figure 7 (system schema)



The UDP-RT performance could be measured by average RTT of messages. If the RTT is significantly smaller than maximum message delay, the protocol definitely serves system well. Though, there are cases when the RTT parameter is not conclusive. The RTT could be high and have a significant jitter in its values. Another parameter should be examined to measure UDP-RT performance: average length of queue of acknowledge pending messages. This parameter is particular useful in the systems with high latency and gives a good measurement for network performance stability.

The server receives messages from clients, does some calculations for every message and sends the results back, so the same amount of messages flows in both directions. This does not count the ACK messages that the UDP-RT is sending for every DATA message. In our particular setup, we recognized that there are no messages lost in server to clients direction, so the UDP-RT has been changed not to send ACK messages from clients. This way the traffic density has been decreased in clients to server direction, where messages delays and losses had frequently happened.

If all clients send messages every 3ms, then, in total, counting the ACK messages, the server receivers 130000 messages per second and sends 260000 messages back to clients. It is equivalent to receiving a message every 7.7 micro and sending one every 3.85 micro. Messages from clients are of size of Ethernet MTU (1.5KB). Messages from server are significantly smaller and do not exceed 100 bytes. The ACK messages are of 20 bytes length, so the total bandwidth required for communications is roughly estimated by 2.2 Gbit, which is less than a half of available bandwidth on the server (6 ports of 1Gbit).

The UDP-RT protocol performance has been tested with two operating systems installed on the server: Windows XP™ Professional and Windows 7™ Ultimate Edition. The Windows XP™ succeeded to process 50000 incoming and 100000 outgoing messages. The average RTT of the incoming messages was 10ms and the average length of queue of acknowledge pending messages was around 7. Clients retransmitted dozens of messages every second. The RTT and queue length measurements are not available for outgoing messages since we abolished the ACK messages for clients. The results for Windows XP™ configuration are shown in the table below.

Table 2 (UDP-RT performance on Windows XP™)

Incoming messages per second	Average RTT (ms)	Maximum RTT (ms)	Average Messages Queue Length	Maximum Messages Queue Length

50000	**10**	**50**	**7**	**33**

Increasing number of incoming messages led to significant raise in RTT times and overflow of unacknowledged messages queue, which size was limited to 50 entries. Moreover, even with 50000 incoming messages, the system was very unstable and the measurements jumped up every few seconds.

It must be mentioned that to achieve the results above, several calibration runs have been done to tune the UDP-RT receiving and sending modules. In final configuration, we used maximum allowed interrupt moderation on all network cards, the receiving module had 6 receiving threads and sending module was configured to run one thread only.

Though, Windows 7™ showed solid performance. The system succeeded to handle 130000 incoming and 260000 outgoing messages and also had only minor jitter in RTT and queue length measurements during several minutes run. No messages retransmissions have been observed. We refer these results mostly to receive-side scaling (RSS) technology of Windows 7™. The results are shown in the table below.

Table 3 (UDP-RT performance on Windows 7™)

Incoming messages per second	Average RTT (ms)	Maximum RTT (ms)	Average Messages Queue Length	Maximum Messages Queue Length
130000	**14**	**31**	**8**	**9**

The configuration of sending and receiving modules Windows 7™ was identical to the one we used in Windows XP™. Further tuning of the configuration may lead to even better results.

Possible extensions for UDP-RT

Piggyback ACK messages

The biggest challenge in UDP-RT is decreasing number of packets running in the network. In the case-study above, we determined reliable path in the network and exempt corresponding ACK messages. Another, more generic, way to reduce number of packets is to piggyback ACK messages on packets carrying DATA messages. To do so, the receiver won't send the ACK immediately upon DATA message receiving, but delay its sending. If there will be DATA message to transfer to the same peer, it will unite the messages and send them in one datagram. The receiver must know for how long it can delay sending of acknowledge. If it delays it for too long, the DATA message could be retransmitted. To overcome this issue, the sending peer should tell the receiver if it is allowed to delay the ACK message and for how long. This could be done by adding a timeout value into DATA message header. This timeout value should be calculated with respect to retransmission timeout model and maximum message delay parameters. For example, *average RTT retransmission model* does not leave much for acknowledge delay, so the sender must increase the

retransmission timer by some δ and pass the δ in the DATA message header. The receiver then will wait for, at most, δ time for piggybacking opportunity. In addition, the receiver will add the exact ACK message delay time into the ACK header, so the sender could subtract this time from the corresponding DATA message RTT. Other schemes for ACK messages piggybacking are possible.

QoS in sending and receiving modules

It has been mentioned in the previous sections, that different message recovery mechanisms may serve as a basic implementation of QoS in UDP-RT protocol. Another approach for this matter is to implement strict QoS model in receiving and sending modules. Indeed, the receiver module threads are working on sets of sockets, so specific sockets could be given higher priority by putting them into separated set and sampling it more frequently than other sets. For example, the sampling could be done by weighted round-robin algorithm. In the similar manner, sending requests queue in the sending module could be separated into several queues, which could be sampled differently.

Implementation of QoS in the sending and receiving modules does not replace the assignment of different recovery mechanisms to different channels. Since the underlying operating system and network equipment do not support QoS, any packet could be lost or delayed, so managing different recovering mechanisms is crucial for overcoming this limitation.

Summary

Real time communications are still far from “plug-and-play” approach. This article tries to systemize TCP/IP protocols usage with respect to real time communications and proposes new reliable protocol for the cases where the standard TCP/IP suite fails to provide required performance.

License

Share

About the Author

Michael Pan
Software Developer VisionMap
Israel

No Biography provided

You may also be interested in...

- IDTP, An Innovative Communication Protocol
- A Perfect Launch Every Time: 7 Steps to Avoid the Worst Day Of Your Career
- Scalable UDP Client Server
- Understanding the Need for a Single Code Base in Enterprise Mobile App Development
- RCF - Interprocess Communication for C++
- Will Mobile Application Development Follow the Path of Web Development?

Comments and Discussions

You must Sign In to use this message board.

Search Comments Go

First Prev Next

Great!
khler 3-Jul-15 20:23

My vote of 4
Member 9922649 18-Aug-14 22:33

Malloc & memcpy
dewilt 11-Mar-14 22:40

Re: Malloc & memcpy
Michael Pan 12-Mar-14 1:58

Re: Malloc & memcpy
dewilt 12-Mar-14 3:01

My vote of 5
divyang4482 13-Aug-13 23:02

Very cool...
bits&bytes 16-Dec-12 5:31

How did you handle the retransmission timer?
THerman 21-Sep-12 6:51

Re: How did you handle the retransmission timer?
Michael Pan 22-Sep-12 5:24

Help with channels
Member 9403037 3-Sep-12 12:46

Re: Help with channels
Michael Pan 5-Sep-12 21:04

Re: Help with channels
Member 9403037 6-Sep-12 15:40

Re: Help with channels
Michael Pan 22-Sep-12 5:19

My vote of 5
Member 4320844 6-Jan-12 8:18

Quick Transaction Protocol - another UDP protocol
camerony 21-Nov-11 20:16

Hello Michael
raji_raman 20-Nov-11 20:25

Re: Hello Michael
Michael Pan 20-Nov-11 22:18

My vote of 5

roman_gin 17-Nov-11 11:36

Re: My vote of 5
Michael Pan 17-Nov-11 23:15

Message Removed
beauw 17-Nov-11 8:32

Re: My vote of 5
Michael Pan 17-Nov-11 22:40

Vote
Kanasz Robert 15-Nov-11 22:23

Re: Vote
Michael Pan 15-Nov-11 23:24

Very useful article
kmamutov 14-Nov-11 21:19

Re: Very useful article
Michael Pan 14-Nov-11 23:04

Refresh 1 2 Next »

General News Suggestion Question Bug Answer Joke Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.