INFO/INGI
UCL, Belgium

Valentin Mesaros
Dec. 7, 2000

MULTICHANNEL DATA PROTOCOL

abstract specifications

Status of This Memo

This document is a draft describing an abstraction of specifications and services. It was designed within the PIRATES project at the Computer Science Department, Université catholique de Louvain, Belgium. If used as reference, it should be cited as “work in progress”.

1. Abstract

As distributed systems and their distributed protocols are developing, it is envisaged that they will need new communication primitives such as means for concurrently message send over the same connection.

The present document describes an abstraction of the specifications and services provided by the Multichannel Data Protocol (MDP), as a proposal for a new data transport protocol medium to Mozart system [5,4]. MDP provides ordered and almost-ordered data transfer, management of hosts with changeable IP addresses, reliable and unreliable priority based data transfer over the same connection and means for temporary failure detection.

2. Definitions

Before beginning the discussion, a definition of the used terms is given.

`upper level` (or `UPL`)
The upper level application.

`message`
Stream of bytes at the upper level.

`segment`
Transport protocol basic unit (i.e., datagram).

`control segment`
Segment containing control information.

`data segment` (or `data`)
Segment containing a whole message or parts of it.

`connection`

End-to-end state storage of two entities that, after a negotiation procedure, send data to each other by means of segments.

3. Introduction

Generally, distributed protocols require reliable data transport mediums in order to send their messages. Because it provides reliability, TCP [9] has been considered a good basis. However, as distributed protocols began to become more demanding, TCP's limitations became noticeable.
These limitations are the followings:

- scalability due to OS resources consumed (e.g., each TCP connection employs a pair of two file descriptors and two port numbers),
- time based decisions, without letting the user to interfere (e.g., TCP closes a connection after a certain period of peer inactivity),
- total order between segments (i.e., there is only one data transport flow)
- there are no means for concurrent and priority massage transfer
- mediocre response to lossy networks (e.g., unnecessary retransmissions).

There is a need for new communication primitives. In order to reach certain requirements, distributed protocol designers often demand that the transport protocols they employ have additional attributes. Message multiplexing, ordered and almost-ordered message delivery, concurrent message transfer and temporal and permanent peer inactivity detection, are among the most requested.

3.1 Requirements to reliable data transport protocols

Besides the well-known requirements a reliable data transport protocol has to address, we add the Mozart ones, which are imposed by the distributed protocols it employs; thus, a reliable data transport protocol must:

- provide reliable data transport;
- provide ordered and almost-ordered message transport and delivery;
- provide high throughput and be scalable (e.g., consume relatively modest system resources for a large number of connections);
- work well with long and lossy networks;
- let the user to interfere with its main decisions (e.g., connection closure, temporary peer inactivity detection);
- provide the possibility of sending messages of variable size, concurrently;
- be rather simple, but still efficient;
- keep on trying to connect, when opening a connection, even if the peer is not yet accessible (i.e., return `tempFail`)
- provide failure detection.

MDP protocol addresses all these requirements, less the last one, which is addressed only partially (i.e., means for temporary failure detection are provided).

4 Multichannel Data Protocol (MDP)

Like RUDP [1], MDP is based on the Reliable Data Protocol (RDP) [10,3,6]. It is layered over connectionless protocols (e.g., UDP [7] and IP [8] protocols) and provides reliable and unreliable data transfer, and in order and out of order delivery.

4.1 The MDP connection

TCP manages only one connection at once, employing a different pair of sockets to identify each connection. Whilst, MDP is able to manage multiple connections, employing a pair of sockets for each connection, with the local socket being common to all of them; thus, increasing scalability. MDP connections are viewed as support and service providers for the channels they contain (serve). Channels are viewed as pipes providing means for data transfer within a connection. An MDP connection allows multiple channels to be created. A connection allows maximum 255 channels of a given type (i.e., 4x255 channels, for writable/readable and reliable/unreliable). MDP channels are unidirectional, allowing send and receive of data, concurrently, within the same connection. The readable channels from a host correspond to the writable ones to the peer.

4.1.1 Opening a connection

Since a state between parties has to be kept and maintained during its lifetime, before sending or receiving any data, an MDP connection has to be issued. After connection opened, data is allowed to be exchanged between parties by the means of channels.

Although employing a three-way handshake procedure for connection open, which may take some time, the upper level is provided with means for writing data (i.e, via writable channel). The data it writes is buffered until the connection is opened and sent to the other party, afterwards. Once each party is aware about its peer, a readable channel is provided, allowing data reading.

4.1.2 Closing a connection

Normally, a connection may be closed only in two ways, either by a close request issued by the UPL, or by the peer. Note that MDP does not close a connection by itself because of timeouts; instead, it delegates to the UPL the radical decisions (e.g., connection closure). Nevertheless, there are situations when a connection closure is issued at the MDP level. These are special cases when a connection can not continue any farther with respect to its state or provided parameters (e.g., the peer does not follow the communication protocol).

The closing connection procedure is a simple one (i.e., it does not guarantee a safe closure), where the initiating party indicates its intention to its peer. When receiving a close connection indication, the host has to notify its UPL and disable the connection state. The UPL has to be notified with `dataLeft`, about unread or unsent/unacknowledged data left to readable or writable channels, respectively.

Although the connection closure is not safe, the UPL is provided with means to make it safe (see Section 4.2.2).

4.2 Unidirectional channels

Within a connection, the channels provide means for data transport and are viewed as distinct transport mediums. Thus, data is sent and received relative to a certain channel. The MDP channels are designed to be unidirectional mostly for the reasons related to channel management issues; they employ a simpler management mechanism. Unidirectional channels advantage the situations where only one-way data transfer is required.

MDP provides both, reliable and unreliable data transfer over the same connection.

4.2.1 Opening a channel

Opening a channel is a two-handshake procedure. Since channels are unidirectional, the channel open initiator is always the writing side (i.e., only writable channels may be initiated). Thereby, the writing party sends an open channel request. Based upon some criteria (e.g. maximum allowed number of reading channels at the same time), the peer accepts or rejects the request. If the open channel request is accepted, a new corresponding readable channel is created and the UPL is notified with `newChannel`.

At the writing side, the channel may be written and data sent right after the open channel request is issued. Since there is no knowledge of whether the channel request would be accepted or not, or of the peer side flow control criteria, we call it an ‘optimistic’ data send (i.e., data could be wasted if channel not granted).

4.2.2 Closing a channel

Whereas, the channel open procedure may only be initiated by the writing side, the channel close procedure may be issued both by the writing side (i.e., normal closure) and the reading side (i.e., broken channel).

The channel closure procedure is a simple one one (i.e., it does not guarantee a safe closure). When a channel close is requested by the UPL, the host will signal this to its peer right away, without waiting for sent data to be acknowledged, at the writing side, or for all data to be received, at the reading side, respectively. Upon a channel closure, if there is still data left to channel buffers, the UPL is notified with `dataLeft`.

Although the channel closure is not safe, the UPL is provided with triggers for detecting when data has been correctly transferred, thus allowing a safe channel closure (see Section 4.9).

4.2.3 Changing channel parameters

When opened, the channels are set with different proposed and negotiated parameters. The UPL has a control of the channels and their parameters. During their lifetime, the channels could change their parameters and notify this to their peer sides. Thus, both sides (i.e., writable and readable), could announce to each other their new requirements (e.g., priority level, flow control criteria).

If there is a need for parameter changes of a certain channel, the UPL is allowed to choose between opening a new channel, having the new parameters set, and changing the parameters of the current one. The former is not imminent, since it applies only to the new created channel and its corresponding data, and not also to the remaining data from the former channel.

4.3 Message multiplexing

By sending them concurrently, through different channels within the same connection, messages are multiplexed. If larger than the MDP basic data unit, they are split in segments, which would be interleaved and sent to peer. At the receiving party, the segments are delivered to the corresponding channels.

This mechanism of multiplexing, provided by MDP channels, implements almost-FIFO between segments corresponding to messages relative to different channels. Hence, there is an order only between messages relative to the same channel. The problems (e.g., data loss over the network) relative to a certain channel must not influence the other channels' functionality. For instance, by employing the channels, the messages relative to different distributed objects do not influence one another.

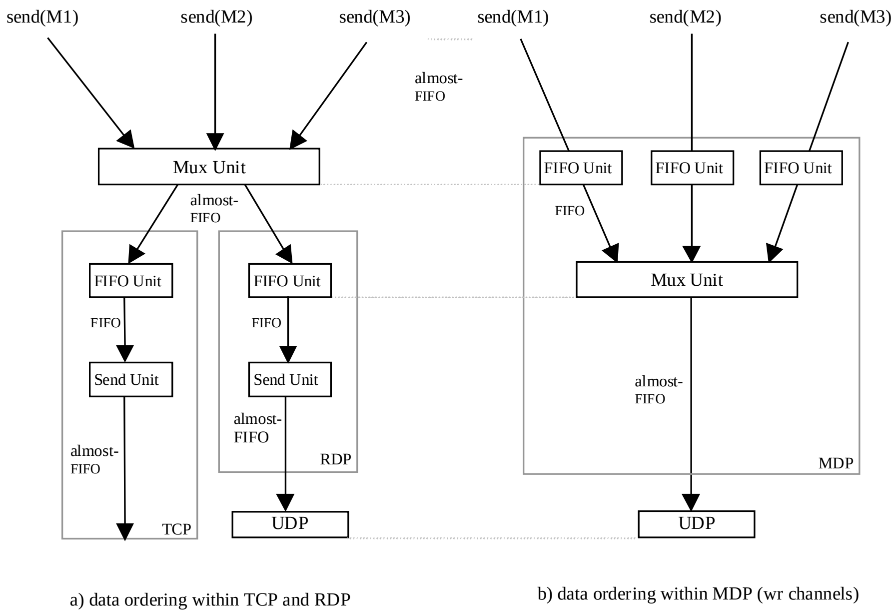


Fig. 1 Data ordering within TCP, RDP and MDP.

The most important reasons of doing the multiplexing at the transport level rather than at the upper levels are mostly related to the independence between different message flows.

Let us consider the example depicted in figure 1. By applying a multiplexing unit above a 'uniplex' one (e.g., TCP or RDP), where the latter imposes a strict sending and receiving order, dependencies between messages relative to different flows are caused, though initially independent. In figure 1, three messages belonging to three independent threads are to be sent. With respect to TCP and RDP protocols (figure 1.a), a total order is forced between these messages due to the fact that multiplexing is done before ordering.

If the multiplexing is employed at the transport protocol level, and after the message ordering, proper to each flow (e.g., MDP design), the messages from different threads remain independent of one another; thus, providing an almost-order between them (figure 1.b).

4.4 Channels with priority

The channels employed by MDP may have different priorities set by their UPL. MDP module provides the UPL with the possibility of sending data concurrently over channels with different priorities. It is the writing party who sets the channel priority; whereas, the reading party has to implement a priority based data delivery mechanism.

Once again, we consider that there are some advantages if implementing the priority scheduler at the transport level. That is, having knowledge about the channel priorities, the reading party would make a difference when addressing data to channels, advantaging the ones with higher priorities.

4.5 The flow control

The term 'flow control' defines means for avoiding the writing side flooding the reading side because of buffer or procession limitations of the latter. Thus, the reading side forces the writing side to slow down the rate it writes the data with, and in certain situations, even to stop the data transfer. In order to keep the separation between the channels, MDP employs a flow control mechanism proper to each of them. The reading party limits the data transmission at the writing party. That is, the former informs the latter about its availability, with the respect to the so-called acceptance window (i.e., the input buffer at the receiving party).

4.6 Probing the peer

Probing mechanism implies host verification, either for existence (i.e., connection state enabled and host addressable), or for resource availability (i.e., input buffer size). MDP provides the UPL with a probing means with respect to host existence. Since they do not depend on resource limitations and host availability, the probing means implemented at the transport level are more efficient than those implemented at the UPL.

The control segments `areYouAlive` and `iAmAlive` are employed for host probing and temporary failure detection. A host can be probed by sending `areYouAlive` to it. When receiving `areYouAlive`, a host must respond with `iAmAlive`.

4.7 Detecting temporary failure

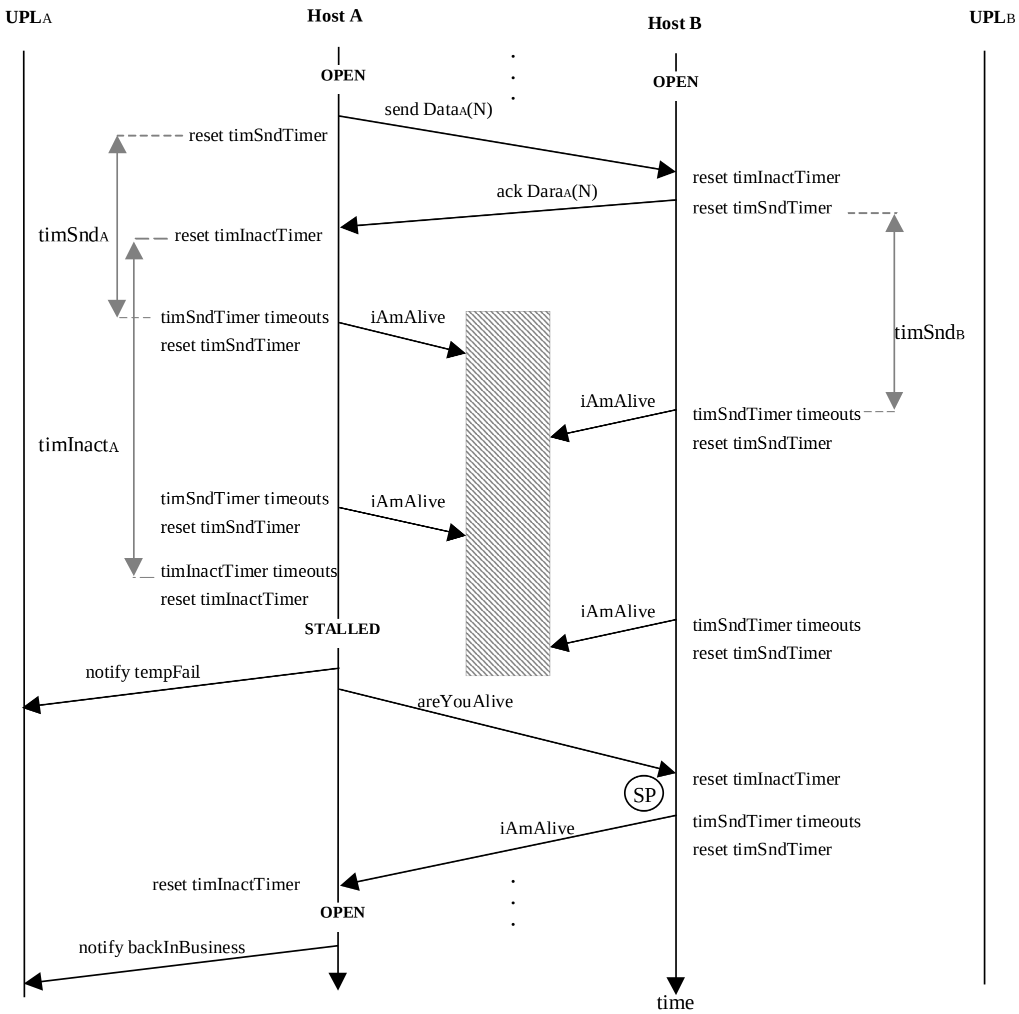
Besides probing (i.e., `pull` mechanism [2]), MDP employs a rather simple `push` mechanism [2], in order to detect temporary failures (e.g., relative network latencies). Within MDP design, temporary failures are considered with respect to connection.

The temp failure conditions are defined by the UPL; that is, each party requests periodical 'life-signals' from its peer. By life-signal we denote any segment received by a host from its peer. The period the life-signals are sent with (i.e., `TIMISND`), has to be less than the peer requested period (i.e., `TIMINACT`). Unless there is data to be sent, `iAmAlive` must be employed. `TIMINACT` is dynamically computed, function the current `SRTT` (i.e., Smooth Round Trip Time) and the constant `FDMN` (i.e., Failure Detection Multiplier Number); the latter being specified by the UPL. `TIMINACT = FDMN x SRTT`

If there is no life-signal received from the peer within `TIMINACT` period, the host must notify its UPL with `tempFail` (i.e., the peer is considered temporarily failed). In order to avoid false `tempFail` detection due to differences between the two `SRTT`s, separately computed for each party, synchronization points (i.e., `SP`) has to be provided. By synchronization point we denote a moment during the connection lifetime, when a host updates the locally computed `SRTT` with the respect to the remote one, computed at the peer side. Each of the two probing segments (i.e., `areYouAlive` and `iAmAlive`) must contain the current `SRTT` computed at the side the segment was issued.

A simple synchronization algorithm would use temp failures as `SPs` and is explained in the followings. The host detecting a `tempFail` sends `areYouAlive`, indicating its current `SRTT`. The peer must respond with `iAmAlive` and may adjust its own `SRTT`, if necessay.

After detecting a `tempFail` the host must not send any other segments but probing ones; the connection is considered `stalled` (i.e., there is a high probability of segment loss). When segments from peer are received over again, the UPL is notified with `backInBusiness`. A clarifying situation is depicted in figure 2, where the hatched region represents a problem occurring during the connection lifetime.



Not receiving any ‘signal’ from peer, host A notifies the UPL with `tempFail`. Since `timInactB` is larger then the duration of perturbation, host B doesn’t detect any network problem.

Fig. 2 A temp failure scenario.

During the connection negotiation phase, each party specifies its own `FDMN` to the other one. The use of the temporary failure detection mechanism is optional; a value of `FDMN` greater than ‘zero’ indicates the acceptance of this service.

4.8 Managing hosts with changeable IP address

Nowadays, it is more and more likely that hosts change their IP addresses while still connected and interchanging information with remote peers. MDP provides a means for maintaining the connection and addressing information between two hosts involved into a communication process, even after they change their IP addresses. It is assumed that only one host at a time changes its IP address.

During its lifetime an MDP connection is identified via its `conId` (i.e., 64-bit unique connection identifier), beforehand set, when connection opened. Thus, even if changing their IP addresses, the involved parties are still addressable, and so the connection.

4.9 Asynchronous notification

It is intended that MDP provide a non-blocking API. For this reason, besides the synchronous notification mechanism (i.e., succeeded or exception raised), an asynchronous one has to be used in order to signal the upper level application about the events occurred *a posteriori*. The followings enumerate the notifications MDP employs. When issued, each notification comes with different parameters that are not detailed here.

`tempFail`
: The peer is considered temporarily inactive.

`backInBusiness`
: After detecting `tempFail`, a 'live-signal' is received from peer.

`connectionClosed`
: The connection has been closed because one of the reasons: `remoteClose`, `remoteRefuse` or `localReset`.

`newChannel`
: A new readable channel has been opened.

`parameterChanged`
: Channel parameters have been changed.

`channelClosed`
: The channel has been closed because one of the reasons: `remoteClose`, `remoteRefuse` or `localReset`.

`dataLeft`
: There is still unread or unsent data left to readable or writable channel, respectively.

`newData`
: There is new data received into the input buffer of the readable channel.

`bufferEmpty`
: The output buffer of the writable channel has been emptied.

`bufferFull`
: The input buffer is full.

When needed, the last three notifications can be triggered by UPL. Their corresponding triggers are `newDataTrg`, `bufferEmptyTrg`, and `bufferFullTrg`, respectively.

4.10 Sequence Delivery Mode

MDP provides the possibility of delivering the segments to the UPL, either in sequence (FIFO segment delivery), or in the order they are received. In the latter case, the segments may be received out of order. This option is set by the UPL.

5. Conclusions

- In addition to the requirements imposed to a reliable data transport protocol, MDP includes the Mozart ones.
- The possibility of employing concurrent communication channels, for data transfer, could be used by Mozart distributed protocols as means for concurrently message sending.
- The fact that MDP never closes a connection without the user's agreement, allows the latter to take proper decisions with respect to the connection lifetime and network behavior.
- The probing and temporary failure detection mechanisms implemented at the transport level, provide means for detecting peer and network anomalies, more efficiently than if implemented at the upper levels.
- Since the mobility between interconnected devices is likely to increase, managing hosts with changeable IP address represents a useful functionality.
- MDP should be tested by running it over networks with different behaviors (e.g., fast networks, lossy networks), under different scenarios (e.g., sending messages of arbitrary size over different channels), and compared with the Mozart current data transport medium (i.e. TCP). If the tests would give benefits to MDP, one could envisage the possibility of employing it, as the main data transport medium within Mozart system.

6. Acknowledgements

The author kindly thanks Prof. Peter van Roy for providing the very idea of multiple channels for concurrently message transfer, and Assist. Raphaël Collet for the fruitful discussions had with, and provided remarks.

4. References

[1] T. Bova, T. Krivoruchka, "draft-ietf-sigtran-reliable-udp-00.txt: Reliable UDP Protocol", 25 February 1999 (work in progress).
[2] P. Felber et al, "Failure detectors as first class objects", 0-7695-0182-6/99, 1999 IEEE.
[3] R. Hinden, "RFC-1151: Version 2 of the Reliable Data Protocol (RDP)", April 1990.
[4] Mozart System, http://www.mozart-oz.org, February 1999.
[5] A. Neiderud, "Design and Implementation of a Network Layer for Distributed Programming Platforms", http://www.sics.se/~annan, February 2000.
[6] C. Partridge, "Implementing the Reliable Data Protocol (RDP)", IEEE/ACM Trans. on Networking, vol.1, No.4, August 1993.
[7] J. Postel, "RFC-768: User Datagram Protocol", August 1980.

[8] J. Postel "RFC-768: Internet Protocol – DARPA Internet Program Protocol Specifications", September 1981.
[9] J. Postel, "RFC-793: Transmission Control Protocol", September 1981.
[10] D. Velten, R. Hinden, "RFC-908: Reliable Data Protocol", July 1984.