Institutionen för datavetenskap
Department of Computer and Information Science

Examensarbete

Arm-P

Almost Reliable Multicast Protocol

av

Fredrik Jonsson

LIU-IDA/LITH-EX-G—08/003--SE

2008-06-18

Linköpings universitet

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Abstract

Distribution of information across IP based networks is today part of our everyday life. IP is the backbone of the Internet and most office networks. We use IP to access web pages, listen to radio, and to create computation clusters. All these examples use bandwidth, and bandwidth is a limited resource.

Many applications distribute the same information to multiple receivers, but in many cases the same information is sent to a single receiver at a time, thus multiple copies of the same information is sent, thus consuming bandwidth.

What if the information could be broadcasted to all the clients at the same time, similar to a television broadcast. TCP/IP provides some means to do that. For example UDP supports broadcasting; the problem faced when using UDP is that it's not reliable. There is no guarantee that the information actually reaches the clients.

This Bachelor thesis in Computer Science aims to investigate the problems and solutions of how to achieve reliable distribution of fixed size data sets using a non reliable multicast communication channel, like UDP, in a LAN environment.

The thesis defines a protocol (Almost Reliable Multicast Protocol – Arm-P) that provides maximum scalability for delivery of versioned data sets that are designed to work in a LAN-environment. A proof-of-concept application is implemented for testing purposes.

The thesis concludes that multicasting is a promising technology for minimizing bandwidth usage. However it also presents significant problems that need to be solved before a multicast service can be said to be reliable. Several methods on how to detect data loss, maintain scalability and error recovery are discussed. Which methods to choose are dependent on the type of application, the maximum number of clients, type of data, available bandwidth, and more.

This page has been intentionally left blank

1 Acknowledgments

Petru Eles for being an inspiring lecturer and for supporting my work with this paper.

Mike Fahl for his input on designing communication protocols.

Dataton Utvecklings AB for financing my work on this thesis and for providing the equipment I needed.

Daniel Bergström for all his knowledge in the intricate details of C++.

Carl-Johan Uhlin for reviewing drafts of this thesis, and providing loads of valuable input on how to improve the thesis.

My family, friends and class mates that have been providing a “supporting act” throughout the past four years.

Joakim Berg, Martin L Gore, Roger Waters and Dave Gilmour for writing all that wonderful music that’s been a source of inspiration during the many hours writing this thesis.

2 Introduction

Network bandwidth in TCP/IP based networks is a limited resource and the number of applications that utilize the TCP/IP is on a constant rise. Most of these applications require reliable data delivery, and an ever increasing number of applications require that an application provides the same *data* reliable to multiple *clients*; this type of application is often referred to as a o*ne-to-many* application. In this thesis we examine how to achieve reliable data delivery while minimizing the bandwidth requirement for *one-to-many* applications using multicasting technology available via the User Datagram Protocol (UDP).

2.1 Outline of this document

This thesis is organized into three (3) sections; the first section provides some background and reviews the research in the area. The next section presents a suggestion to a reliable multicast protocol and tests preformed on the protocol. The third section presents conclusions and suggestions to further work within the area.

2.2 Readers

The intended readers are developers and researches with an interest in networked *one-to-many* applications.

It's assumed that the reader has a fair understanding of networking and common TCP/IP networking protocols, like UDP, IGMP and TCP, nevertheless section 3.1.1 provides some background to TCP/IP protocols.

2.3 Purpose

The purpose of this thesis is to define a protocol that achieves reliable-data-delivery on an IP based Local-Area-Network using currently available multicasting technology and providing a non linear increase of bandwidth usage with every new *client*.

2.4 Scope

This thesis focuses on using currently available IPv4 UDP multicast technology for achieving reliable data delivery in a LAN-environment.

The term reliable refers to the reliable delivery of data. That is a *client* is to receive all the data in a data-set or not at all.

2.5 Terms and Abbreviations

Term	***Explanation***
Protocol	A set of rules that two, or more, parties must adhere too in order to communicate with each other. In this document the parties are nodes connected to a network.
ACK	A signal used in the protocol to indicate that an operation has been performed successfully.
NACK	A signal used in the protocol to indicate that an operation has failed.
Multicast	Communication between one *sender* and multiple *clients*.
Unicast	A communication between two parties, one *sender* and one *client*.
PGM	Pragmatic General Multicast
IETF	Internet Engineering Task Force (http://www.ietf.org/), an organization that among other things standardizes protocols used for communication on the Internet.
API	Application Programming Interface
Dataset	A collection of a data with a finite size, i.e. a file, a table in a data base, etc.
Packet	An atomic element of data in a communication channel, i.e. an Ethernet network. The term packet is casually used in this thesis, the context defines the type of packet, UDP, IP, etc.
Network congestion	A condition that occurs when more data are fed into the network than there is available bandwidth.
MTU	Maximum Transfer Unit, the maximum allowed size of packets on a given leg in a network path. For an Ethernet network the MTU is 1518 bytes.
Reliability	Refers in this document to reliable delivery of data.
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
Field	An atomic entity of information in a data structure.
IGMP	Internet Group Management Protocol
Multicasting	The ability for a sender to send a message designated for multiple receivers.
PoC	Proof of concept implementation
BSD	Berkeley Software Distribution.
IPTV	A system for delivering digital television using a protocol layered on top of IP (Internet Protocol).
ATM	Asynchronous Transfer Mode. A virtual circuit high speed communications standard, commonly used in the telecom sector.
IEEE 1394	Also known as FireWire. A high speed data bus used to connect hard drives and other peripherals to personal computers.
MAC-Address	Media Access Control address. Used in Ethernet and is a world unique address that identifies the network adaptor.

3 Pre-study and Research

Section 3.1-3.1.2 provides some insights to TCP/IP and the need for a reliable multicast protocol. Section 3.2-3.3.3 breaks down the problem of how to achieve reliability in the broad sense, and presents mechanism that allows one to achieve it.
Finally in section 3.4 desirable properties of a reliable protocol are presented.

3.1 Background

3.1.1 The TCP/IP protocol suite

Before discussing how to achieve reliable data delivery using multicast UDP it's suitable to briefly study some of the protocols included in the TCP/IP protocol suite.

TCP/IP is commonly used as a collective name for a set of protocols defined by the Internet Engineering Task Force (IETF). The set includes a wide variety of protocols with very different purposes. For example TCP **[20]** defines how to achieve reliable data transfer between two nodes (*point-to-point*), RIP [**17]** defines how routers exchange routing information and FTP **[18]** defines how to transfer files.

The protocols within the TCP/IP suite form a hierarchy. Hierarchies of network protocols that are dependent on each other are often described as *layers*, the series of *layers* often referred to as the network stack. Each *layer* adds new functionality. Usually the layers at the top of the stack are application domain specific, whereas the layers at the bottom are more generic.

A commonly used model for describing a network stack is the OSI-model (Figure 1). The OSI-model defines seven (7) layers ranging from physical at the bottom to the application at the top. The physical layer and the link layer are hardware dependent, and define the electrical attributes and base level communication protocol. Ethernet and ATM are defined in these two layers.

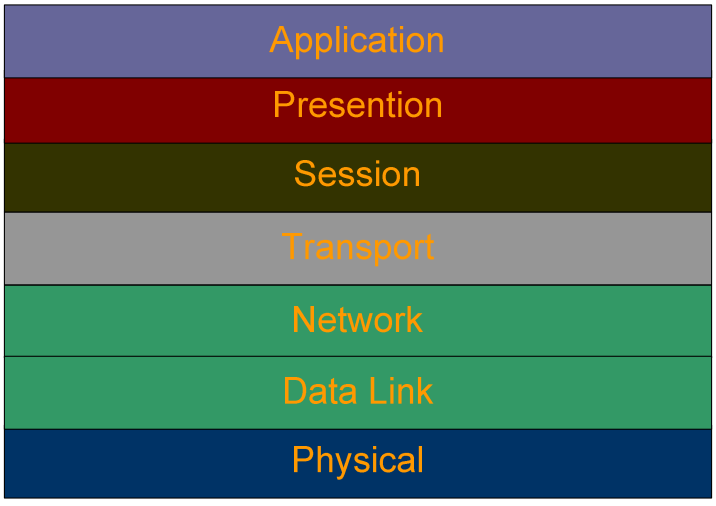


Figure 1 The OSI network model

When the OSI model is used to describe TCP/IP the first TCP/IP specific layer is the network layer (Layer 2 from the bottom), RIP, ARP and IP are TCP/IP protocols implemented at the network layer. IP **[22]** is the foundation of many other protocols, including the one that is the basis of work presented in this thesis. For this reason we begin with a brief look at IP.

The basic element of IP is the IP-packet. The IP-packet is a payload carrier with size, sender and destination address attached.

IP-packets are fed into the network from one node and are transported to the next hop node. It's likely that the next hop node is a router. The router investigates the destination address of the IP-packet and forwards the IP-packet on the link that is part of the shortest path to the destination.

IP is connectionless and non reliable, this means that any node can send a packet without any prior negotiation and that a packet is not guaranteed to reach the *receiver*. IP does not provide any feedback to the sender when a packet is not delivered.

One of the most commonly used protocols in the TCP/IP suite is TCP **[20]**; TCP is used by a multitude of services, such as FTP **[18]**, SMTP, and HTTP **[19]**. TCP adds reliability and is connection-oriented. This means that data are reliably transferred from the sender to the client. If delivery fails TCP will notify the sender. Connection-oriented refers to that the parties must negotiate a *connection* before they can exchange data.

TCP extends the IP-packet with several pieces of information; an important part of that information is a sequence number. The sequence number is increased by one for every IP-packet that is sent. When a receiver detects a gap in the sequence number it indirectly (via the feedback loop, see below) sends a request to the sender asking the sender to resend the missed IP-packet(s).

The sender also maintains a feedback loop for every IP-packet it sends, this way the sender knows if an IP-packet has been received, if not it can resend the IP-packet.

TCP also provides mechanisms for regulating the transmit rate. When the network is loaded, the time before the sender receives the feedback increases. Eventually the feedback response time will be larger than the response timeout period, which indicates to TCP that the transmit rate shall be reduced, and that the timeout period should be adapted to the new transmit rate. This helps to reduce network congestion and to provide fair sharing of bandwidth between TCP based services.

A second widely used protocol based on IP, and the one that the work in this thesis is based on, is UDP **[21]**. UDP is a low overhead protocol. UDP shares many of the properties of IP, for instance it is non reliable and connection less. UDP adds an important piece of information to the IP packet called a *port*. A *port* is just a 16 bit number, but it allows for multiple UDP based services to run on the same host since the *port* can be used to dispatch each received UDP packet to the correct service handler.

Unlike TCP, UDP does not provide any mechanism for fair sharing of bandwidth, thus a single UDP service can use up all available bandwidth in a network without respect to other services in the same network. A UDP based service can thus starve other services.

UDP provides multicasting support; support for multicast is implemented at the network layer via the IP protocol, but since many systems do not provide access to the IP level and because of the addition of *ports* in UDP, UDP is often the choice for application developers that whish to utilize multicasting in an IP network environment.

Multicasting is achieved by the use of reserved IP address ranges; there are two types of multicasting: Broadcasts that are received by anyone that listens, and group-casting that is received only by the nodes that have joined the group.

To join a group a node must issue a join request. The join request is handled by the router and it's the router's job to forward group addressed packets to any link that leads to a member of the group.

Broadcasts are also handled by the router, but in that case the broadcasted IP-packets are forwarded on every link that shares the same broadcast address as the sender.

For a more detailed look into TCP/IP the reader is encouraged to read Kurose et al **[10]** or Jeremeu Bentham **[26]**.

3.1.2 The need for a reliable multicast protocol

The main driving force behind researching multicasting is to minimize bandwidth use. Bandwidth is a limited resource and its use can be translated into a monetary value, thus multicasting can save money.

Multicasting is already in use in many areas. Routers use the RIP multicast protocol to exchange routing table information, and IPTV services uses multicasting to distribute the video data. None of these services require TCP style reliability; best effort reliability with some packet loss is acceptable.

Reliability on a protocol level is a desirable attribute since it allows a *sender* to know that the data it sent has been delivered. TCP would do a good job in achieving reliability; it would however come with a high price tag in terms of resource use, bandwidth, processing and memory. Thus a reliable multicasting protocol would save resources for *one-to-many* applications.

A large research effort has been put into devising reliable multi cast protocols. The research has led to several suggestions to *a standard protocol*, NORM **[1,2]** and PGM **[3]** are just two examples.

The US Navy Naval Research Laboratory provide an experimental implementation of NORM **[14]** and PGM is included in the Microsoft Windows™ Socket API **[15]** in Windows XP™.

Most of the research has been aimed at devising protocols that are suitable for a wide area network (WAN) environment like the Internet. Some of them, like NORM, rely on reliability to be implemented in the end nodes. Others like PGM require some level of network infra-structural support, for example in the routers.

The focus on the WAN-environment implies, long end-to-end networks paths, the number of routers that each packet must pass through is fairly large, the transfer time can be in the order of several hundred milliseconds and bandwidth of the link layer are likely to vary along the network path. The scalability target is usually in the order of ten or a hundred thousands of clients.

In contrast, this thesis is aimed at defining a protocol for use in the LAN-environment. In such environment the number of routers between any nodes on the network is small, packet transfer time is in the order of a few milliseconds, the link layer bandwidth of each leg in the network graph are likely to be the same and the requirement for scalability is in the order of a one to a hundred clients.

Even though the Internet and the LAN are two very different environments the issues that need to be addressed are very similar.

3.2 Issues to address by every multicasting protocol

Recall from the scope (Section 2.4) that reliability means that *a guarantee* is given that the data sent actually reaches that *receiver* while the network is operational.

To achieve this there is a couple of issues that need to be addressed. These are error detection, error recovery, congestion avoidance, and congestion control.

These issues are discussed in the following sections. The risk of experiencing any of the above mentioned issues increases with the number of clients. This is because a larger set of clients requires more communication and presents more locations where these issues may arise. Thus it's important to consider scalability when designing a reliable multicast protocol.

3.2.1 Error Detection and Error Recovery

Error recovery is straight forward; the detecting party simple reissues the *damaged* or *lost data*. In a case where the *receiver* is the detecting party it will have to request the data from the sender. In the case where the *sender* is the detecting party the sender can resend the data. The question is how many times a *client* should re-request the data and what the *server* should do when the requested data is no longer available?

Error detections are more complex, there are several types of errors that can occur. From checksum errors, via lost packets to applications that generate wrongly formatted messages. Errors in link or network layer are normally never reported to the upper layers when using API's like BSD Sockets, thus from the *clients* point of view these errors appear to be lost packets.

Let's focus on lost packets as they are likely to be responsible for a majority of the errors. There are many causes of packet-loss, the most common one is receive buffer overflow. A receiver buffer overflow occurs when a node receives more packets than it can process, the affected node enters a congested state. Receive buffer overflow can occur in both routers and receivers. A packet loss in a router is more serious than a packet loss in a receiver since the packet loss may impact *n* receivers.

According to Milner **[7]** and Sunahara **[8]** packet loss comes in bursts so that multiple consecutive packets are lost. The reason behind the bursty behavior is that the *senders* in the network need time to detect that the congestion has occurred and thus enforce their congestion avoidance algorithms. During this period the *senders* will continue to feed packets into the network which cause the node to remain congested. Note that multiple nodes may be in a congested state at the same time.

How can a service detect lost packets? One way is to *tag* each packet with a sequence number. The receiver can then look for gaps in the sequence numbers in the stream of packets. A more detailed explanation follows below.

Figure 2 A sequence with a lost packet

Figure 2 illustrates a sequence of network traffic where a packet is lost. At T0 a packet with sequence number 1 is received; upon reception of P1 a timer is started. At T1 another packet, P3, with sequence number 3 is received, this packet is not the expected one (P2), the packet is put on hold. At T2 the timer started at T0 expires and triggers a request of P2 from the *server*.

Even though the gap has been detected earlier, at T1, no action is taken until T2. This is because IP does not guarantee the in order delivery of packets, thus P2 may arrive after P3 even though it where sent before P3.

What should the *sender* do when there is no more data to send? With the approach described above the *receiver* will start requesting packets at T2 even if P1 was the only packet that was sent. One approach is to let the *sender* send NULL-packets, these are data packets like P1 and P3 but contain no data thus, they allow a *client* to detect that there is no more data. Another approach is to include a *total packet count* with each packet, this way the *receiver* will know how many packets to expect, as long as it has received one.

If a *positive acknowledgment scheme* is used, then the *client* can include information about the last in-order packet it received with every ACK. If the packet presented in the ACK mismatches the *servers'* opinion then the *server* can resend the mismatching packets.

3.2.2 Congestion control / avoidance

As described in section 3.2.1 a major cause of packet loss is congestion. Congestion is a result of a buffer overflow at one or more nodes. Each node (router, switch, desktop computer etc) on the network have a receive buffer, whose size may vary from very small to very large. If the receive buffer becomes full it indicates that the reception rate is higher than the outbound processing rate. In such case there isn't much the node can do except to drop any packet it receives while the buffer is full.

Let's look at why congestion occurs, in Figure 3, we assume that links A, B and C all run at 100 mega bits per seconds. Now let's assume that node X tries to talk to node Y with a rate of 100 MBPS and that node Z tries to send data to node Y with a rate of 30 MBPS,. Since the link from the switch to node Y is 100 MBPS it won't be able to carry the combined transmit rate of X and Z. The receive buffers in the switch will therefore start filling up, and once they get filled packets will be dropped.

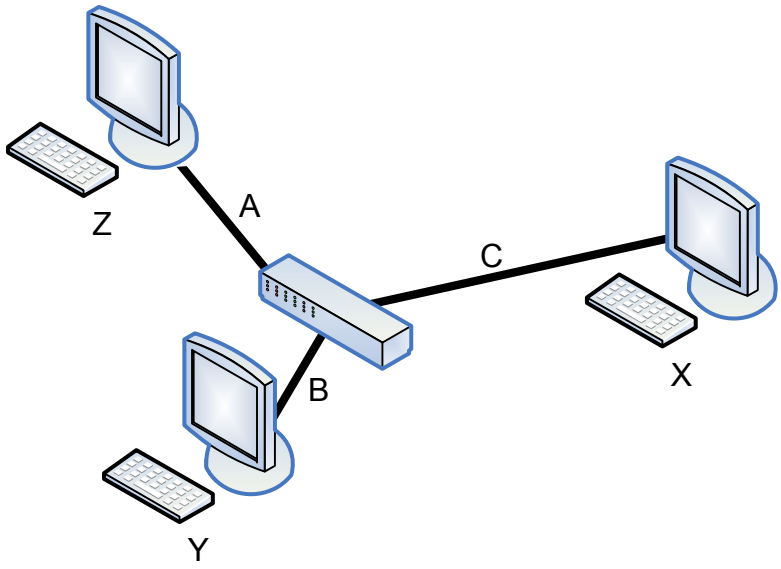


Figure 3 A example LAN

Since the network doesn't provide its *current state* to the nodes it's up to the active services on each node to detect when the network is congested and enforce a mechanism that reduces the effects on the service and that allows the congestion to dissolve.

TCP does this by reducing the bandwidth of a connection upon detection of packet loss. This means that the packets sent on the connection are more spread in temporal space, allowing more time for the network nodes to process buffered packets. If TCP detects another packet loss it decreases the rate further, the process is repeated until the congestion ceases. At this time the transmission rate is slowly increased.

This type of congestion control scheme allows a *sender* to transmit at the maximum available rate until the congestion occurs at which point the bandwidth is reduced. It also

provides fairness among competing services, since all services are likely to detect roughly the same amount of congestion.

This works well for TCP since it is point to point and maintains a feedback loop that allows both nodes to detect the sequence number of the last IP-packet that the other end received. In a one-to-many multicast situation with several hundreds of *clients* a TCP style scheme would lead to an amount of feedback that could cause congestion or worsen an existing congestion. In addition it would require a lot of resources at the *server* in terms of memory and processing capacity.

For the above mentioned reason it's important to minimize the feedback. TCP uses an ACK based feedback scheme. However, a NACK based scheme that provides no feedback until an error (i.e. a packet drop) occurs is more suitable for largely scalable multicast protocols **[4]**. With a NACK scheme the *client* is responsible for detecting the congestion which takes some of the workload away from the *server*. NACK based schemes are discussed further in section 3.3

Then, what should be done when congestion is detected? The TCP scheme of a quick reduction of transmit rate is a good solution as it gives each node more time to process each packet.

How can congestion be avoided? Unless the full state of the network is known at all times this cannot be achieved. There are communication models (ATM and IEEE 1394) where communicating parties must negotiate the bandwidth requirement at all nodes along the path between the parties. If such a scheme is employed in the example above (See Figure 3) the switch wouldn't agree to allow the 30 MBPS connection from Z to Y since all available bandwidth of the link from the switch to Y is used by X.

There is research on how to change the effects of congestion. As noted by Floyd and Fall **[25]** congestion caused by UDP traffic can cause TCP services to starve. Floyd and Fall suggest that the routers, or other network infrastructure devices, should monitor the traffic and if a UDP based service or other non responsive service[1] is using significantly more bandwidth than the competing TCP services at times of a high traffic load then the router should limit the bandwidth of the non responsive service so that the bandwidth is shared fairly. Such scheme would cause the non responsive services to get high bandwidth when TCP traffic is low, and cause the non responsive service bandwidth to be limited at times of high traffic load. Such a scheme would cause the non responsive service to loose packets at times of high traffic load, in favor of not starving TCP based services, thus a non responsive service must employ error recovery to maintain reliability.

If routers would employ the scheme described above then congestion management would be simple in a one-to-many multicast, in fact the service would not need to enforce any congestion control, the word would be shifted over to error recovery. Routers that use the scheme suggested by Floyd and Fall are not commonly available, thus the service must provide some sort of congestion control at the *server* and the *clients*. At the moment a

[1] A service that does not enforce a TCP friendly congestion control algorithm.

NACK based scheme for detecting the congestion is a good candidate. This is further discussed in section 3.3.

3.3 Schemes for Reliable Multicast Data Transfer

The following three sections describe three different approaches to how to achieve *reliable data delivery* using multicasting; each scheme has its specific set of properties.

3.3.1 Positive acknowledgement scheme

The first approach suggests that each *client* uses two communication channels for communication with the *server*. The first is a non reliable multicast channel; this channel is used to receive data from the *server*. The second channel is a reliable channel, like TCP, that is used to acknowledge the packets that the *client* has received correctly. Kurose et al **[4]** call this type of protocol *sender-initiated*; with such a protocol the *server* detects the errors and initiates error recovery.

This type of protocol puts all the workload on the *server*. For example the *server* must keep track of what packets have been acknowledged. Though there are many ways to implement such an acknowledge tracking system. It adds extra processing work to the *server* and thus impacts scalability.

Kurose et al define the factors that impact scalability as a function of the verbosity (direct) and processing requirement (indirect) for a protocol. The writer would also like to add the memory use, thread requirements, and the use of other resources as indirect impacting factors.

For example, assume that TCP is used as the reliable channel. TCP is connection based and most implementations use a fairly large amount of memory for buffering. In addition each TCP channel might use timers and threads. Thus, even with a *server* with a fast CPU and large quantities of memory there is a limit to the maximum number of *clients* connections the *server* can maintain.

This scheme can lead to a situation called *feedback-implosion* when the number of *clients* is high. *Feedback-implosion* is when the amount of *feedback data* is larger than the amount of *data sent* by the *server*. Let's illustrate with an example: Let's assume that we have 1000 *clients* and, the size of each packet we send is 1000 bytes and the size of an acknowledge packet is 2 bytes. This means that for every 1000 bytes the *server* sends it must receive 2000 bytes. Thus only 1/3 of the data handled by the *server* is data sent by the *serve*r. The problem grows linearly with the number *clients*.

Though the scalability is somewhat limited, a protocol scheme as described above has advantages too.

- The *client* implementation is simple.
- The positive acknowledgement scheme allows the *server* to know the data reception state of each *client* and that can be a big advantage when detecting congestion.
- The use of a reliable feedback channel guarantees that the feedback reaches the *server*.

3.3.2 Negative acknowledgement scheme with a reliable return communication channel

The second type of protocol scheme described by Kurose et al **[4]** also uses two communication channels, one non reliable and one reliable. The difference is that the reliable channel is not used for acknowledgment of each sent packet; instead the *client* sends a *negative acknowledgement* via the reliable channel when the *client* detects that a error occur.

This type of scheme is sometimes referred to as *receiver-initiated,* this refers to that it is a responsibility of the client to detect lost packets and initiate the recovery sequence. The idea behind this scheme is that packet loss only occurs under certain conditions like network congestion or broken links.

The intention of this scheme is to minimize the risk of *feedback-implosion*. The scheme will do a god job in the case where the *clients* are distributed widely so that they span a large network, like a WAN. With such distribution and with packet loss at random nodes the packet loss will only affect a subset of the *clients*, thus the transmission of NACK:s is not likely to cause long periods of feedback implosion. Feedback implosion can still occur, for example when the *router* closest to the *server* drops the packet. In such a case every *client* will detect a packet loss and initiate error recovery.

A LAN-environment is quite different to the WAN-environment; usually only a few routers/switches are involved, thus it's likely that when a packet loss occur, it does so in a node that affect many or all *clients* and thereby increases the risk of feedback implosion.

It might be argued that the LAN is a more controlled environment and that the available bandwidth is generally higher, packet loss are rare and thus this scheme might still be a good candidate for a LAN-environment.

This scheme provides enhancements to scalability, compared to the scheme described in section 3.3.1. This is due to that part of the processing now takes place in the *client*. Nevertheless, he *server* must still maintain a large number of connections, and the question is how much of an improvement to scalability this scheme is since each connection will still use up the resources. Bandwidth utilization is also improved as less bandwidth is used for feedback.

3.3.3 Negative acknowledgement scheme with a non reliable return communication channel

The third scheme, also discussed by Kurose et al in **[4]** was originally suggested by Ramakrishnan et al in **[5].**It is also *receiver-initiated* and uses negative acknowledgement. The new aspect of this scheme, compared to the one presented in section 3.3.2, is that it uses a single non reliable multicast communication channel for both data transmission and feedback.

An effect of this scheme is that every *client* receives the feedback from every other *client*. That is because the *clients* use the same multicast channel for feedback as the *server* use for transmission to the group. Another effect of this scheme is that the feedback may experience packet loss.

Is there any advantage in using a common return channel? The use of a single shared return communication channel significantly reduces the resource use at the *server*. Only buffers, threads, timers, etc need to be allocated for a single communication channel rather than one per *client*. Combined with the fact that the processing requirement is balanced between *server* and *clients* this scheme provides a high level of scalability. However, in order to utilize the scalability, the *feedback-implosion* issue must be addressed.

Feedback-implosion can be minimized through using a randomized packet loss detection period. Recall from Figure 2 that a the period from T0 to T2 is the time before a packet is assumed to be lost, now what if this period is randomized and a packet loss occurs? This means that different *clients* will detect the packet loss at different times and thus send their NACK at different times. Since the NACK is received by every other *client*, other *clients* that are waiting for the same packet that where NACK:ed can suppress their NACK and wait for the *servers* response. If a response to the NACK isn't received within a period then *clients* can reissue the NACK.

There will be times when two or more *clients* will detect a packet loss before they have received the NACK for the lost packet from the other *client(s)*. In such case the *server* will receive multiple NACK:s and generate multiple responses.

Kurose et al suggest in **[4]** that this type of scheme is the most suitable for one-to-many applications in terms of resource use and scalability.

3.4 Summary

In this section we summarize the discussions presented in section 3.2-3.3.3.

To provide reliable data transfer over a multicast communication channel (like multicast UDP) error detection, error recovery, congestion control and congestion avoidance must be addressed **[4]**.

Error recovery can be based on resending information. Error detection can be implemented using sequence numbers and timers. The main bulk of errors are assumed to be caused by lost packets. It's common that multiple consecutive packets are lost **[7].**

A protocol that supports a high degree of scalability is likely to experience conditions that require error recovery more often than a protocol that only supports a modest level of scalability; that is because the number of *clients* increase the number of possible points of error and thus increase the risk of a packet loss (or other error) with every transmitted packet.

Congestion control and congestion avoidance are important to address, otherwise a single UDP multicast service can *starve* all other services running on the same network. Some researchers **[25]** suggest that congestion control and congestion avoidance for UDP services should be handled by the network infrastructural devices, such as routers, rather than in the end nodes.

The type of scheme to use in a multicast protocol depends on the scalability requirement. For the highest degree of scalability the protocol must minimize its resource use (memory, CPU, etc). The processing workload should be balanced between *server* and *clients*. To provide maximum scalability the protocol should use a single multicast communication channel for data transmission and feedback. A NACK based feedback scheme should be used, as it minimizes the risk of feedback implosion **[4, 5]**.

If the scalability requirement is modest then *clients* can use a reliable feedback communication channel, such as *TCP*. This way a *client* is guaranteed that the feedback reaches the *server*. Depending on the scalability requirement ACK or NACK based schemes can be used. A NACK based scheme balances the workload more even between *client* and *server* thus provides a higher degree of scalability. NACK based schemes also utilize bandwidth more efficiently. For ACK based schemes very little workload is put on the *client*, thus making *client* implementation simple.

If every *client* receives feedback sent by other *clients* in a NACK based scheme, then the feedback can be used to avoid multiple *clients* from sending NACK for the same packets **[5]**.

At present no single protocol provides an *one-fit-all* solution to reliable data transfer via multicast. PGM and NORM are attempts but so far they have not been widely spread and PGM infrastructural support is so far limited in low cost network infrastructural devices.

4 Arm-P – The Almost Reliable Multicast Protocol

Arm-P is a multicast protocol which provides a high degree of reliability; Arm-P is designed to be implemented as part of the application layer using UDP as the transport protocol. Arm-P does not require any network layer support other than required by IP and IGMP, thus off the shelf network infrastructure components can be used to build an Arm-P compatible network. Figure 4 illustrates the relation between Arm-P and the OSI model and provides a comparison to another commonly used TCP/IP protocol.

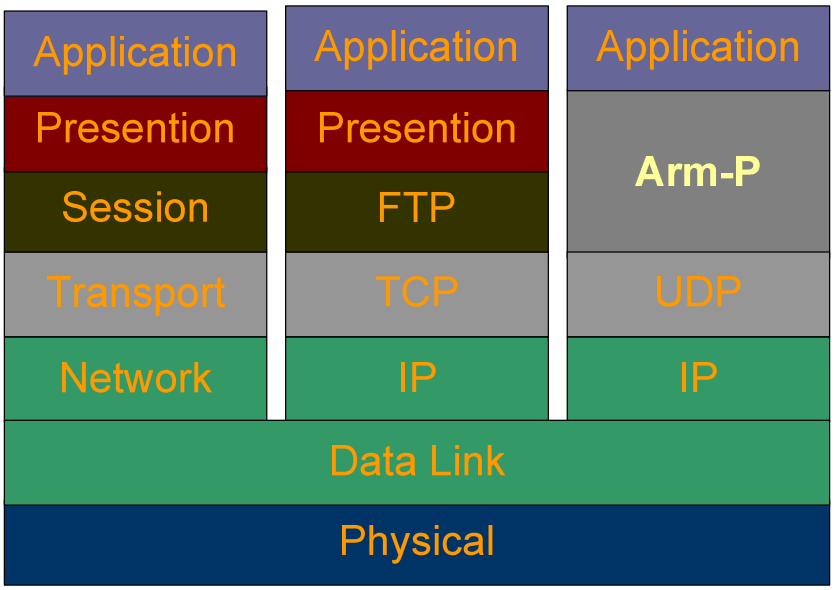


Figure 4 Arm-P and the OSI network model

Arm-P as designed with the LAN-environment in mind and one of its main goals is to achieve a high degree of scalability. Arm-P is designed to use few resources and to utilize bandwidth efficiently. For these reasons Arm-P is based on the scheme described in section 3.3.3. This means that Arm-P uses a single multicast communication channel for outbound and inbound communication and that the *receiver is* responsible for error detection and recovery. Thus, Arm-P is *receiver-initiated*.

Arm-P is designed for transferring finitely sized data sets (An image, file, etc). It supports partial update of the data sets, meaning that the changed part of the data set needs to be included in a partial update. The data set is versioned, meaning that a change to the data set causes it's version to change. Multiple data sets may share the same Arm-P communication channel.

An example application can be used to illustrate the above mentioned features. Let's assume an application that distributes images. The images change from time to time, but usually only a part of the image changes. In this example the image is the data set. When the image changes its version number is increased. If the entire image has changed the entire image needs to be sent. If only the upper left corner has changed a partial update of that corner need to be sent. Multiple images can be distributed over a shared Arm-P communication channel.

Note: Arm-P does not define how the part of the data set to update with a partial update is defined, Arm-P only provides the means to be able to do partial updates.

The basic element of Arm-P is the *message*: a *message* is a container that provides identifying information that a *receiver* can use to verify that a message is correctly formatted. A *message* can contain two types of information, a *segment* or a *command*. A *segment* carries a part of the data content of a data set; *segments* are sent from the *server* to the *clients. Commands* are a request or a reply to a request, i.e. a *ping* or an ACK-request reply. *Commands* are contained in a single *message*. Commands can be sent in booth directions.

The figure below illustrates the relation between an IP-packet, UDP-packet and an Arm-P message. The line from bottom up indicates *a contained-in* relation, i.e. an Arm-P message is contained in a UDP-packet.

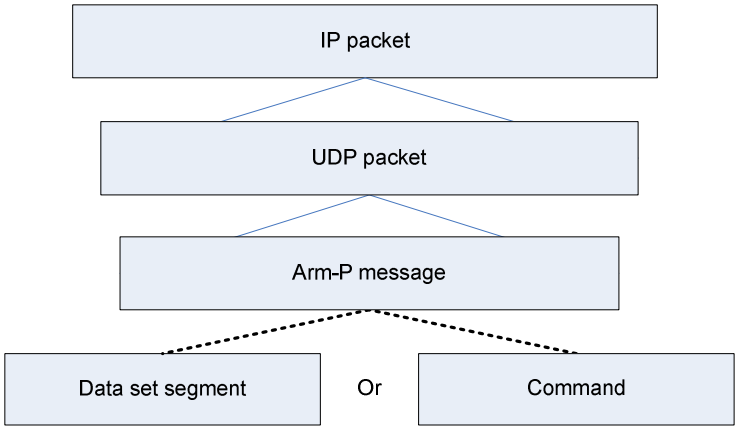


Figure 5 Relationship between IP and Arm-P

Segments (*DSS*) are grouped into data set updates (*DSU*). A data set update can be either *full* or *partial* (See above for an explanation.). A receiver may not use a *DSU* before all the *DSS* in a DSU have been received, and a receiver must not use a partial *DSU* unless the receiver has access to the data set that the partial *DSU* is based on. Recall the image example: if a *client* has a copy of an image with version 15, it must not apply partial update for image version 17 onto it as it would lead to an erroneous image.

It's recommended to keep the size of a *DSS message* small enough to fit into a single network topology maximum transfer unit (MTU) (Data link layer packet). As Arm-P runs over UDP, Arm-P will support larger *DSS*:s. They will simply be segmented as they are being sent. The UDP-receiver will then reassemble them into a packet of the original size. But segmentation increases the risk of packet loss, since a missing piece at reassembles will cause the entire UDP-packet to be lost.

The *client* keeps track of two *DSU*:s, the *pending* and the *active*. The *pending DSU* is the *DSU* currently being received. The *active DSU* is the last fully received *DSU*. Once an application has used the data in the *active DSU*, the data content of the *active DSU* can be

thrown away. However, the information, such as version, must still be kept on record in order to be able to handle partial *DSU*:s.

In the following sections data structures used by Arm-P, and Arm-P:s methods for achieving reliable multicast is presented. Also refer to Appendix A & B for Arm-P state flow charts.

4.1 The Methods

4.1.1 Client side DSU managment

Initially the *client* has no *active* or *pending* DSU. The *client* should accept any *full* DSU as the *pending*. If the client receives a *DSS* that is part of a partial DSU it shall request a full update of the version specified in the partial *DSS* DSU from the *server*.

Once a client has received a complete *DSU* it makes that DSU *active* and begins waiting for the next *DSU*.

After the first *full* DSU is received the *client* can accept *partial* DSU:s. However, only *partial DSU*:s of the version that is next after the *active* DSU shall be accepted as *pending*. For example, let's assume that the *active* DSU has version 14. Then a *client* can accept a *partial DSU* with version15, but not with 16. Since a DSS is usually fit into a single IP-packet they can be delivered out of order. Thus, the *client* should put the *DSS* of versions later than the expected on hold as they may just be out of order deliveries. Figure 6 below illustrates the scenario of out of order delivery of *DSS*:s.

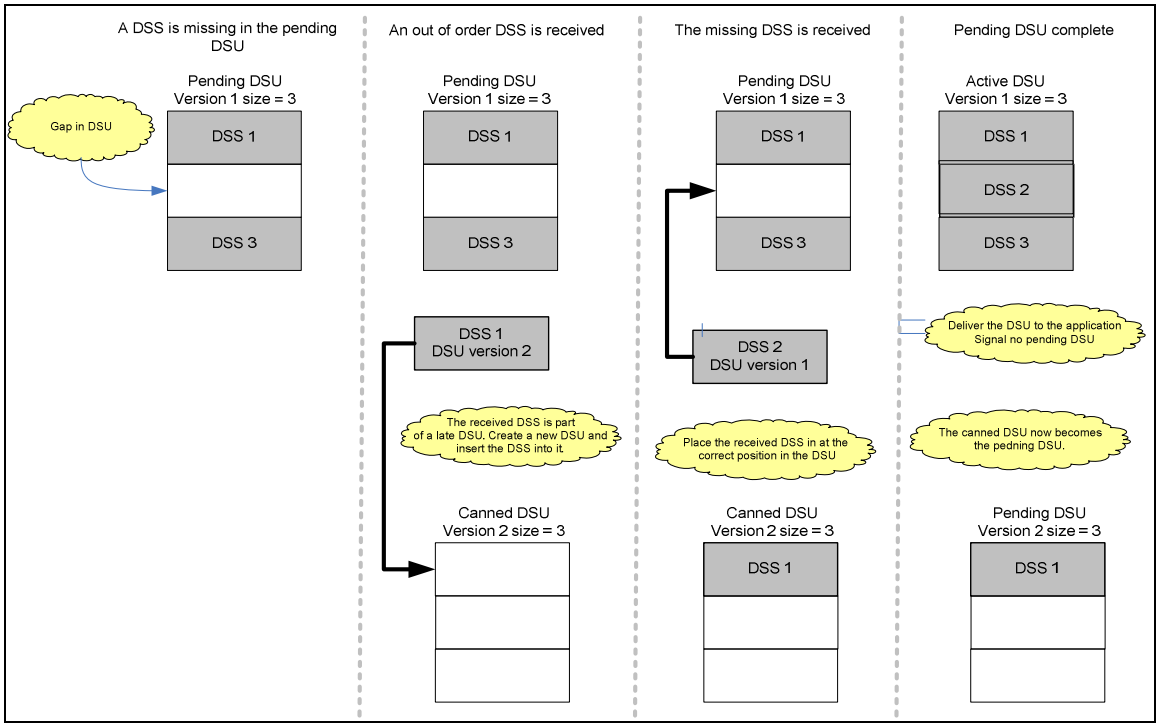


Figure 6 Out of order delivery of DSU DSS's

If a DSS is received that is part of a *full* DSU that is newer than the *pending* DSU then the *pending* DSU should be dropped and the newer (*full*) DSU becomes the *pending* instead.

This can cause a *client* to change *pending* DSU even though all the DSS in a DSU are en-route. That occurs when packets are delivered out of order. For example, lets assume that the *pending* DSU is *partial* and with a version of 4, Then a DSS from DSU of version 5 that is a *full* DSU is delivered before the last DSS of DSU of version 4 is received, in this case the *client* will drop the *pending* DSU and, instead, make the DSU for data set version 5 the *pending*.

Arm-P is based on the assumption that packet reordering is rare in a LAN-environment, and that there is some temporal space between *DSU*:s that will further reduce the risk of *newer* DSU DSS:s to be received before the all of *DSS*:s of the *older DSU*, making the above situation unlikely.

4.1.2 Lost packet detection strategy

The *receiver* is responsible for detecting lost packets. The packet loss detection algorithm is based on inter-*DSS* timing; this is based on the assumption that the packets in a *DSU* arrive at regular intervals.

When a *client* receives a *DSS* for the *pending DSU* a timer, called T, is started. If no further *DSS* of the *pending DSU* is received before the expiration of T, the *client* should re-request the missing DSS's of the pending *DSU*. The period of T shall be set in accordance with the strategy outlined in section 4.1.3. Upon reception of a *DSS* of the *pending DSU*, T is restarted. Figure 7 below outlines the strategy.

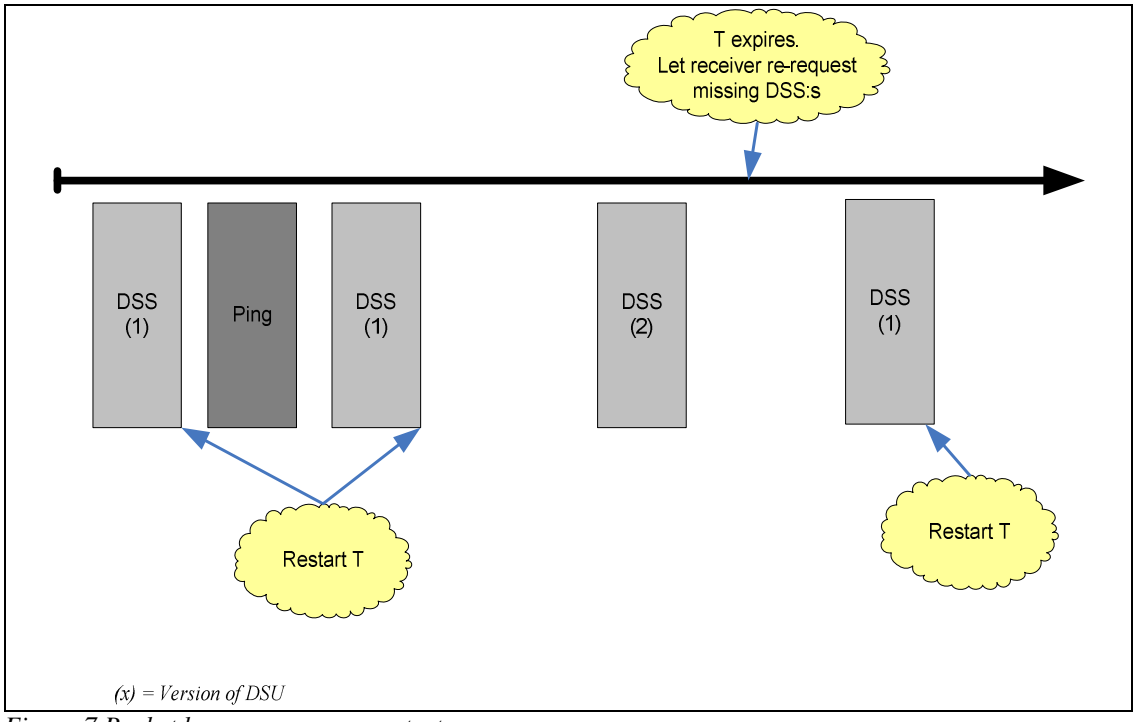


Figure 7 Packet loss error recovery strategy

This packet loss detection strategy is not flawless. If all DSS:s in a DSU are lost then, the *client* will not know that an entire DSU has been lost. For this reason Arm-P is only Almost Reliable!

Once the *client* receives a DSS again the *client* will detect the lost *DSU* due to the change in version number, and can re-request the missed *DSU*:s from the *server*. If the requested DSU isn't available at the *server,* then the *server* should respond with a *full* update of it's current DSU. This way the *client* is given a chance to resynchronize with the data set.

4.1.3 Randomized re-request

As purposed in section 3.3.3, *clients* should randomize the point-in-time when they re-request packets. In Arm-P the method to achieve this is to randomize the inter DSS timer timeout (i.e. the expiration period of T in the example given in section 4.1.2. To calculate the timeout period the below formula is used.

T_{per} = **max**(*minimum-timeout, ratefactor / rate*) + *randomvalue* **mod** *maxrandomness*

```
          rate  The current transmit rate
    ratefactor  A value that scales the rate to a suitable value timeout
                for a given rate.
   Randomvalue  An unbounded randomized value
 maxrandomness  The value of the largest randomness span allowed
```

The formula is based on the assumption that the average delivery time for a packet is linearly scalable with the transmit rate.

4.1.4 Congestion control and avoidance strategy

The Arm-P congestion control management is shared between the *clients* and the *server*. The basic idea is to reduce the *server* transmission rate when congestion is detected. No action is take at the *client* end, which is based on the assumption that the *server* responsible for the bulk of the transmitted data.

The *client* detects lost packets. As described in section 4.1.2, a lost packet is assumed to be an indication of network congestion. The *client* response to a lost packet is a NACK that is sent to the *server* re-requesting the missing segment(s). This behavior is effective when the packet loss is caused by buffer overflow in the client, a broken link or a buffer overflow in a router.

If the cause of the lost packet is a heavily congested network then the NACK will add to the network load and cause the network to remain congested. In such a situation the NACK may not reach the *server*. Thus, the *server* will not be able detect that the network is congested. To cope with such a situation Arm-P requires that the server requests ACK from randomly chosen *clients periodically*. If the network is congested then the ACK-request will not reach the selected *client(s)* or the congestion will cause the ACK response to be delayed beyond the timeout of the ACK-request.

This strategy is potentially slow. If the *server* would request ACK from every *client* with every DSS, the result would be *feedback-implosion* that would impair scalability. So there must be a balance between being able to detect congestion and minimize the feedback.

The suggestion is that the server requests an ACK from at least one *client* with every DSS, this means that if the *DSS* frequency is low and the number of *clients* is high then it can pass several seconds before an ACK has been requested from every *client*. Thus every implementer must find a scheme those suites the needs of the service that Arm-P is used for.

The *servers* response to the detection of congestion is to cut the transmit rate by 50%. If a consecutive congestion condition is detected the rate is cut by 50% again, and so on, until the minimum rate is reached.

Once the congestion releases, the server transmit rate is raised with 10% of the maximum transmit rate every 100 *ms.* As illustrated in Figure 8 the transmit rate should be cut by 50% if a new congestion is detected while recovering the transmit rate.

Note that maximum transmit rate not necessarily is the maximum rate of the network over which Arm-P is running. The maximum transmit rate is the maximum bandwidth allocated to Arm-P.

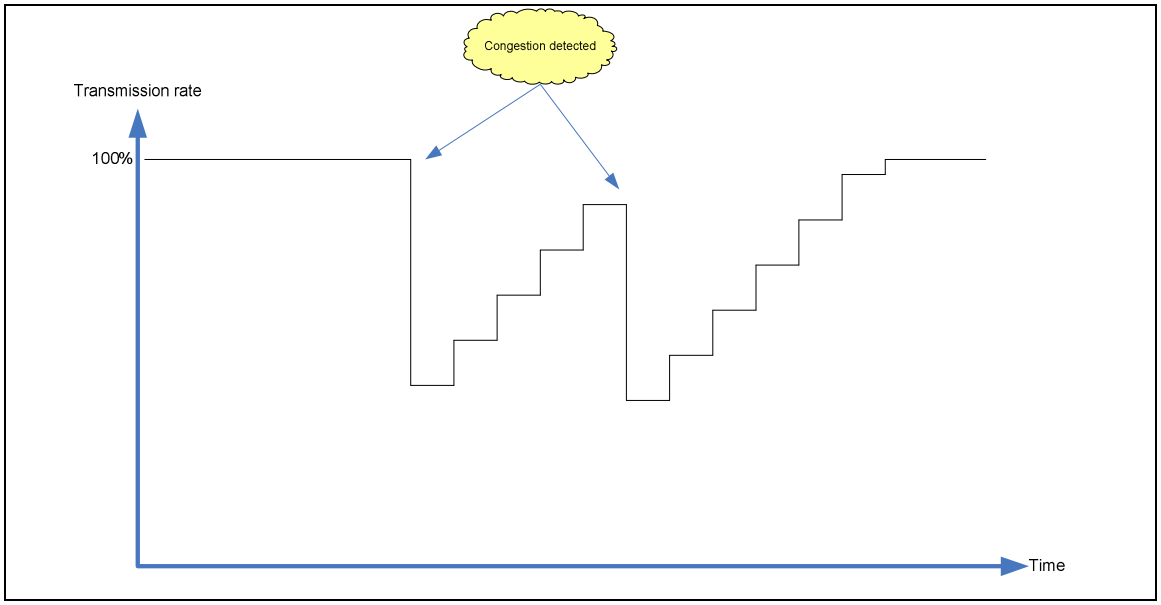


Figure 8 Server congestion response

Note that Arm-P doesn't enforce the *server* to resend packets lost to congestion detected via ACK's. There are two reasons to that: (1) the *server* can't know which packets have been lost; (2) resending the packets could cause the congestion to be prolonged.

4.2 Echo-Suppression strategy

As a result of using a single multicast communication channel for all communication each *sender* (Clients and server) receives every *message* they transmit (loop back.). For this reason it's important to be able to distinguish messages from *self* early in order to be able to discard them without consuming too much resource. One way to do so would to filter on the *senders* IP-address. The disadvantage of that is that the *server* and *client* can not run on the same host. So, instead of filtering on the IP-address each *message* is tagged with a sender reference. Once the header of the *message* has been found it's easy to locate the sender reference and use it as filter. The sender reference is single 16-bit value and must be unique to the server. How to negotiate a sender reference is outside the scope of this thesis.

4.3 Error recovery strategy

When a *client* detects a missed *DSS* it should send a request (NACK) for the missed *DSS* to the *server*. The missed *DSS* is identified by its *DSU* type, its *DSU* version and *DSU* index (See section 4.4.3).

If the *server* doesn't respond within a period, then the *client* should re-request the missed DSS again. If the *server* continues to not respond to *DSS* requests, then the *client* should assume the *server* to be lost/dead.

If the *server* isn't able to respond with requested *DSS*, the *server* should start sending a full update of its current data set. The cause of this behavior is that if a *server* can't access the requested *DSS,* then the most likely cause is that the requested *DSS* is part of a *DSU* that has gone out of scope, in other words the data set has changed before the *DSS* request was handled.

4.4 Data structures

In the following sections the data structures used by Arm-P are described.

4.4.1 Data types

Arm-P uses a single data type with a variable bit width, called U*n;* U means unsigned integer and *n* is the width of that integer in bits. Thus, U8 is the same as byte, U2 is a two bit wide unsigned integer, and so on.

4.4.2 Message

Name	***Data type***	***Comment***
syncHeader	U8[4]	Used by client to verify that the message is correctly formatted.
protVersion	U4	Version of protocol used, currently 1.
mSize	U12	Size of message content.
senderRef	U16	Source of the message.
contentType	U8	0 = Command 1 = Segment
content	U8[size]	The content data
syncTrailer	U8[4]	Used by client to verify that the message is correctly formatted.

The message is a container for Arm-P *commands* and *segments.* The purpose is to provide an *identity* field that a *receiver* can use to perform basic sanity check of the message allowing for early discarding.

If the message is transported via a stream, like TCP, then the *syncHeader* and *syncTrailer* allows a *receiver* to detect start and end of a message.

A *receiver* should verify *syncHeader, protVersion, contentType* and *syncTrailer* before doing any further decoding of the content.

The *senderRef* is used to allow the *client* to suppress echoed messages, that are messages that originated from the *client* and where echoed back by the router/switcher.

4.4.3 Data set segment

Name	***Data type***	***Comment***
dsVersion	U12	Version of the data set, changes every time the source data set changes.
type	U2	0 = DSS is part of a full DSU. 1 = DSS is part of partial DSU.
ackRequest	U2	0 = From none 1 = From all 2 = From selection
dsID	U16	ID of the data set.
dsuIndex	U16	Index of the segment within the data set update.
dsuCount	U16	Number of segments in the data set update.
segmentSize	U16	Size of the DSS data.
segmentData	U8[segmentSize]	Data content
ackID(x)	U16	The ID to use when acking, if *receiver* is on the ACK list.
ackListCount (o)	U16	The number of clients to request acknowledge from.
ackList(o)	U16[ackListCount]	A list of client ID

(o) Marked fields are sent if ackRequest = 3 *(x) Marked fields are sent if ackRequest = 1 or 2*

Data sets segments (*DSS*) are sent from the *server* to the *clients*. *DSS*:s is the carrier of the data set data. A collection of *DSS* relating to the same data set (Same version of the data set and same data set ID) is a Data Set Update (*DSU*).

dsID defines which data set the *DSS* relates to, a *receiver* can dispatch on dsID to allow multiple data set to share the same Arm-P data channel.

Each *DSS* contains the total numer of *DSS:s* in the he *DSU*(*dsuCount*), and the location of the *DSS* with in the *DSU*(*dsuIndex*).

dsVersion is the current version of the data set. Every time the data set changes the version is increased by one. Note that 0 is not a valid version, and should not be used.

A *DSU* can be a *full* or *partial*. A *full DSU* contains all data for a data set.

The *ackRequest* field is used by the *server* to ask *clients* for acknowledgement of a *DSS*. Acknowledgements can be requested from *all* or a *selection* of *clients*. If the acknowledgments are requested from a *selection*, then the *ackList* contains the *sender reference* of those *clients* that the *sender* is requesting acknowledge from.
The *ackListCount* and *ackList* fields do not need to be included in the *DSS* if *ackRequest* type is *all* or *none*.

4.4.4 Ping command

Name	*Data type*	*Comment*
commandID	U8	= 1

Sent from *server* to *clients* at periodic intervals; allows *clients* to detect that the server is alive.

4.4.5 ACK reply command

Name	*Data type*	*Comment*
commandID	U8	= 2
ackID	U16	From state packet,

Sent from *client* to *server* as replay to a acknowledge request in a DSS.

4.4.6 Request missing DSS command.

Name	*Data type*	*Comment*
commandID	U8	= 3
dsID	U16	Datset to request DSS's from.
dsVersion		Version of the data set that DSS's are requested from.
requestType	U8	0 = Full 1 = Partial
reqDSSCount	U16	Number of DSS to request.
reqDSSList	U16[reqDSSCount]	Which DSS to request.

Sent from *client* to *server* when a *client* detects a missing DSS. The *server* views a request as a NACK (see section 3.3.2).

4.5 The Proof-of-concept implementation of Arm-P

As a part of this exam work an Arm-P *proof of concept* (PoC) implementation has been developed. The PoC implements most of the features of Arm-P, but ACK support was left out due to time limitations.

The PoC has shown that implementing the *server* side is fairly straight forward. The *client* side is more complex. This is due to the need to handle *partial* and *full* an keeping multiple DSU:s up to date(*pending, active* and *out-of-order*).

5 Testing and verification

In this chapter tests and results for certain aspects of Arm-are is presented. The tests have been preformed on a small number of *clients* but should still present results that are applicable to a larger set of *clients*.

The purpose of these tests is to provide data that can be used to provide setup details for the PoC, for example for the re-request randomization presented in section 4.1.3. The tests also aim to provide tests that verify functionality of the *PoC* implementation.

5.1 Test setup

In the test setup the *server* application distributes a 400 by 400 pixels image with a color depth of 32 bits to the *clients*. The image changes periodically.

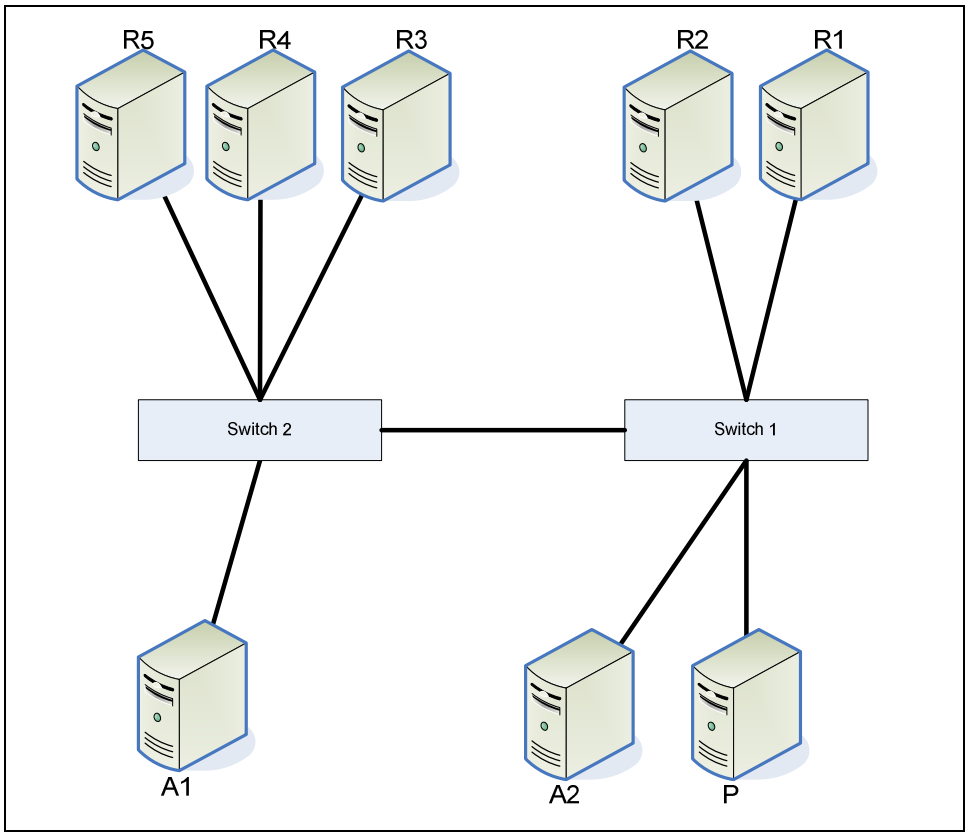


Figure 9 Test network.

In Figure 9 above *Rn* represents the *clients* used. P represents the *server*. Two switches are used for interconnecting the *server* and *clients*. The network does not include a router. *An* represents a host that can feed *"disturbing"* data into the network.

5.2 Test cases

5.2.1 Request reply time

Measure the round trip time (RTT) from the transmission of a *DSS* request to that the *client* actually receives the requested *DSS*. Test booth with loaded and unloaded network.

Expected result: The RTT is assumed to increase with increased network load.

Result:

	Average (ms)	***Max (ms)***	***Min (ms)***
Unloaded	4.5	6.8	3.7
Single TCP load	10.2	12.0	9.5
Multiple TCP load	10.2	12.3	9.4
Single UDP load (2 Mb data/s)	8.8	15.3	4.5

The purpose of this test is to estimate an expected response time for when a client is requesting a *DSS* and till the client receives the *DSS*.

5.2.2 Scalability

Connect *n clients* to single *server*; compare network utilization to a setup with only a single *client*.

The *n clients'* setup is expected to produce a network utilization that is only marginally higher or the same as the single *client* setup.

In the current test environment only a small number of *clients* could be tested simultaneously, no significant increase in bandwidth was detected.

5.2.3 Congestion Avoidance

Trigger the server congestion avoidance by stimulating the *recivers* to re-requests *DSS*:s the *clients* have already gotten.

Expected result: The server should respond by reducing the transmit rate.

Result: Network bandwidth analyzer indicates that bandwidth use is reduced.

5.2.4 Congestion detection

Create network congestion by overflowing the network every 10 seconds with UDP traffic for 4 seconds at a time.

Expected result: The congestion should be detected either by lost ACK reply or the reception of a NACK.

Result: The *clients* detect the congestion within an average of 220 ms, only 20% of the NACKs they generate are received by the *server*.

5.2.5 Error recovery

Stimulate the *server* to drop random *DSS*:s.

The test is preformed with 1, 3 and 5 clients.

Expected result: The clients should detect the missed *DSS* and re-request it.

Result :The *clients* where able to detect and recover the missing *DSSs*.

6 Conclusions

The purpose of this thesis was to define a protocol and to implement the prototype of a protocol handler that realizes a reliable multicast communication across a LAN using UDP as the transport protocol. The study of the current research on the subject (Chapter 3) led to the conclusion that there are four issues that need to be addressed in order to achieve reliable multicast: Error detection, error recovery, congestion control and congestion avoidance.

Tests show that the error recovery strategy is sufficient, currently *DSS*:s of a data set that is *active* on the *server* can be requested. A strategy that allows for the recovery of older DSS:s might be required, which can easily be added using a sliding window in the *server*.

Reliability in Arm-P can be improved. There are situations where Arm-P is not one hundred percent reliable, refer to section 4.1.2 for an example. Nevertheless, if there is a more or less constant flow of *DSU:*s the purposed lost packet detection algorithm could be sufficient. However, improving reliability is the focus of further development of Arm-P.

Tests indicate that scalability is good. However, the test system was too small to provide a definite result. Testing a large system of 50 or more *clients* would provide a more definite answer to the scalability properties of Arm-P.

Congestion control needs to be improved. The current algorithm is primarily based on the feedback of NACKs. Since the network is probably congested, the NACK may not reach the *server*, or do so at the second or third try. This is causing the congestion detection to be slow.

The congestion avoidance works. Once the congestion is detected the Arm-P congestion avoidance strategy is quite offensive and thus provides the network with a good chance of recovering from the congestion.

The work on this thesis has given insights into the problems that need to be overcome in order to achieve reliable multicast delivery of data. In addition it has provided a testbed / prototype that can be used for further development in the area. Arm-P is a step closer to reliable multicast, but more work needs to be done to provide full reliability, a high degree of scalability and congestion handling that are responsive.

7 Future work

In the following sections a number of possible improvements to Arm-P are presented.

7.1 Using multiple multicast communication channels

The use of a single multicast communication channel minimizes the resource use, but there are disadvantages too. In order to receive feedback, the *server* must be a member of the multicast group it transmits through. For this reason every packet the *server* sends will be echoed back using up bandwidth and an increased work load for handling and discarding the echoed packets.

A way to solve this is to use two multicast channels. This does not remove the problem entirely (See note below), but it can reduce it. The *server* only joins one multicast group, but is aware of the other and can send packets to it. Every *client* joins two multicast groups.

One of the multicast groups is used for communication from the *server* to the *clients*; the *server* is not a member of this group. The other multicast group is used in communication from the *clients* to the *server*; every *client* and the *server* join this group. With such a configuration the packets sent from the *server* (Which will be the bulk) are not echoed back to the *server.* The *server* will receive all packets from the *clients*, and each *client* will receive packets from *itself*, other *clients* and the *server.*

Note: Multicast group management is handled in the router, multicasting will however work even if the network is built using switchers, that is because if a switch doesn't know the MAC-address of the destination of a packet it will forward the message on all of it's links.

7.2 Server side suppression of packet retransmission

Arm-P defines a strategy for minimizing the amount of NACK:s[2] sent; even so, multiple requests may reach the *server*. This occurs when multiple *clients* detect a lost packet only a few milliseconds apart and, thus, send a NACK before they have seen the NACK for the same packet from the other *clients*. The *server* will therefore receive multiple NACK:s for the same packet within a short time span. In such a situation the *server* can discard NACK:s for the same packet if they have responded to a NACK for that packet within the past 20 ms or so.

[2] Request for a segment

7.3 Adding stream support

In its current form Arm-P does not support the distribution of streaming data. This is a limitation, and in many cases (When the size is not known) it is more feasible to transmit a stream rather than a finitely sized data set. The problems raised by distributing streams are the much the same as presented in chapter 3, if focus is on reliable delivery rather than real time delivery of data. Arm-P defines strategies for solving many of those problems. Adding stream support to Arm-P would be beneficial and probably a requirement in order to support certain type of applications.

7.4 Improving reliability

A way of improving reliability is to tag every DSS with a consecutive number that increases with every sent DSS, thus providing each DSS with a unique sequence number. When idle, the *server* can distribute the last sent DSS sequence number with every ping. This would allow the *client* to detect a mismatch between the segments it has received, and the ones the *server* has sent allowing the *client* to resynchronize. It's unlikely that a *client* would lose the ping commands for an infinite amount of time. Thus it's likely that the *client* would realize that it has missed a DSS relatively quickly.

References

[1] Negative-acknowledgment (NACK)-Oriented Reliable Multicast Protocol
http://www.ietf.org/rfc/rfc3940.txt, 2007-11-01

[2] Negative-Acknowledgment (NACK)-Oriented Reliable Multicast (NORM) Building Blocks
http://www.ietf.org/rfc/rfc3941.txt, 2007-11-01

[3] PGM Reliable Transport Protocol
http://www.ietf.org/rfc/rfc3208.txt, 2007-11-01

[4] "A comparison of sender initiated and client-initiated multicast protocols.", Kurose et al.
Joint International Conference on Measurement and modelling of Computer Systems. Proceedings of the 1994 ACM SIGMETRICS
http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=183043, 2007-11-15

[5] "A Negative Acknowledgement with Periodic Polling Protocol for Multicast over LANs", S. Ramakrishnan and B. N. Jain
Proc. IEEE Infocom'87, pp 502–511, Mar-Apr 1987.

[6] "Multicast Packet Loss Measurement and Analysis over Unidirectional Satellite Network", Mohammad Abdul Awal et all
http://www-sop.inria.fr/rodeo/personnel/Abdul.Awal/Multicast_Packet_Loss_UDL_AINTEC05.pdf, 2007-11-15

[7] "Robust Speech Recognition in burst-like packet loss", B. Milner,
Acoustics, Speech and Signal Processing, ICASSP, 2001 IEEE International Conference, Volume 1, 7-11 May 2001 Page(s):261 - 264 vol.1
http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/7486/20365/00940817.pdf?tp=&arnumber=940817&isnumber=20365, 2007-11-18

[8] "Characteristics of UDP Packet Loss: Effect of TCP Traffic", Hideki Sunahara et al
http://www.isoc.org/INET97/proceedings/F3/F3_1.HTM, 2007-11-20

[9] "Fcast Multicast file Distribution", Jim Gemmell and Jim Gray
http://research.microsoft.com/research/pubs/view.aspx?msr_tr_id=MSR-TR-99-14, 2007-11-20

[10] "Computer Networking: A top down approach featuring the Internet"
J.Kurose and K.W Ross, 2nd edition, Addison Wesley
ISBN: 0201976994

[11] ACMA
http://citeseer.ist.psu.edu/cache/papers/cs2/119/http:zSzzSzwww.univ-pau.frzSz~cphamzSzPaperzSzISCC03.pdf/maimour03amca.pdf

[12] ”Use Forward Error Correction To Improve Data Communications”
Electronic Design, August 21, 2000
http://electronicdesign.com/Articles/Index.cfm?AD=1&ArticleID=4648

[13] “Video loss recovery with FEC and stream replication”
Chan, S.-H.G.; Xing Zheng; Qian Zhang; Wen-Wu Zhu; Ya-Qin Zhang;
Multimedia, IEEE Transactions on
Volume 8, Issue 2, April 2006 Page(s):370 - 381

[14] “NACK-Oriented Reliable Multicast”, Naval Research Laboratory, US Navy
http://cs.itd.nrl.navy.mil/work/norm/index.php, 2007-11-21

[15] “Reliable Multicast Programming (PGM)”, MSDN, Microsoft
http://msdn2.microsoft.com/en-us/library/ms740125(VS.85).aspx, 2007-11-21

[16] “Multicast Tech FAQs”, Multicast Technologies Inc
http://www.multicasttech.com/faq/, 2007-11-21

[17] “RFC 1058 - Routing Information Protocol”, C. Hedrick, Network Working Group
http://www.faqs.org/rfcs/rfc1058.html, 2007-11-25

[18] “RFC 959 - File Transfer Protocol”, J.Postel and J.Reynolds, ISI
http://www.faqs.org/rfcs/rfc959.html, 2007-11-25

[19] “Hypertext Transfer Protocol -- HTTP/1.1”, R.Fielding et al, Network Working Group
http://www.faqs.org/rfcs/rfc2616.html, 2007-11-25

[20] “RFC 793 - Transmission Control Protocol”, Defense Advanced Research Projects Agency (ARPA)
http://www.faqs.org/rfcs/rfc793.html, 2007-11-25

[21] “RFC 768 - User Datagram Protocol”, J.Postel, ISI
http://www.faqs.org/rfcs/rfc768.html, 2007-11-25

[22] “RFC 791 - Internet Protocol”, Defense Advanced Research Projects Agency (ARPA)
http://www.faqs.org/rfcs/rfc791.html, 2007-11-25

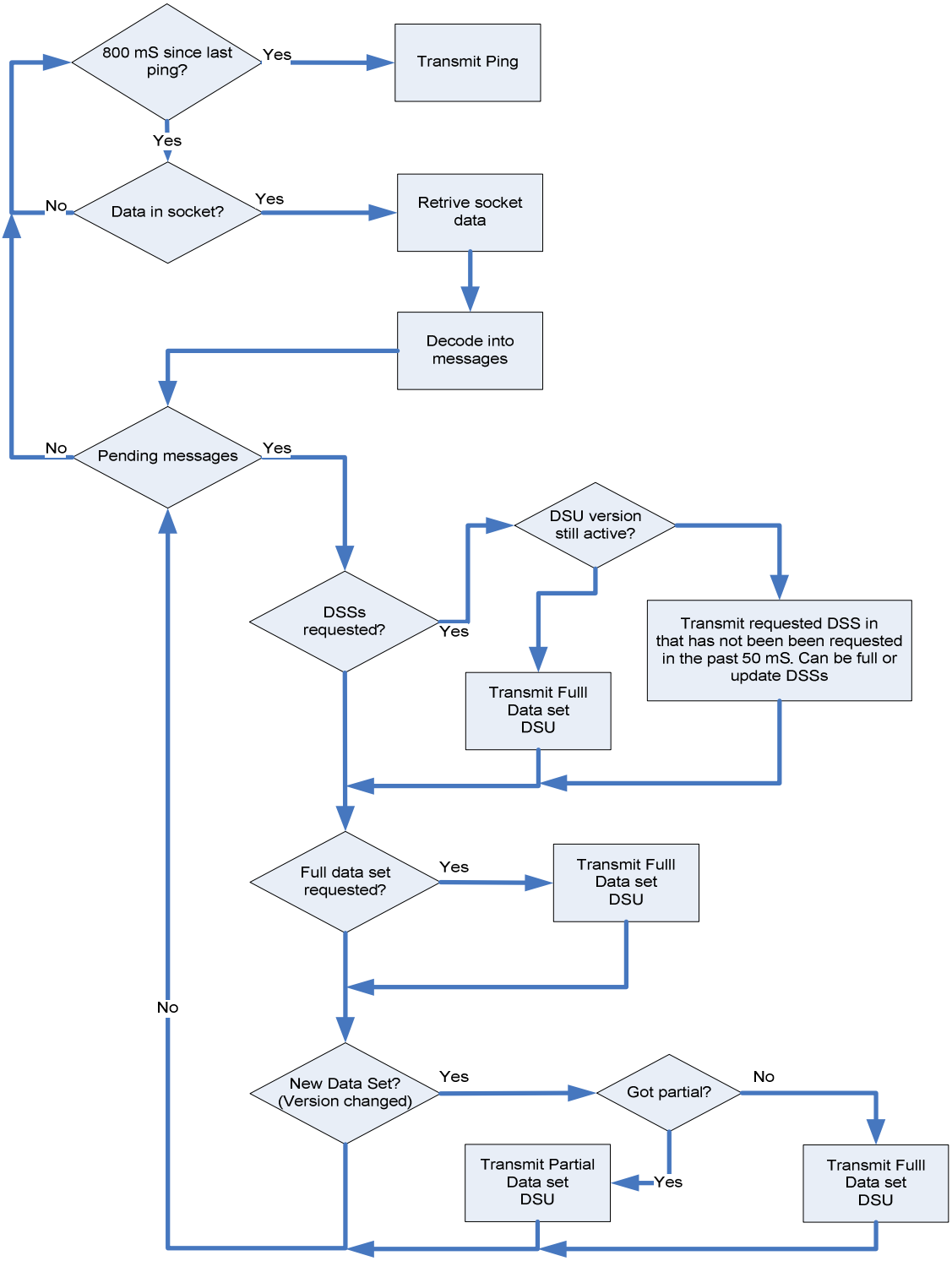
[23] “Berkeley sockets”, Wikipedia
http://en.wikipedia.org/wiki/Berkeley_sockets, 2007-12-20

[24] TCP/IP primer
http://internetfixes1.tripod.com/tcp.htm

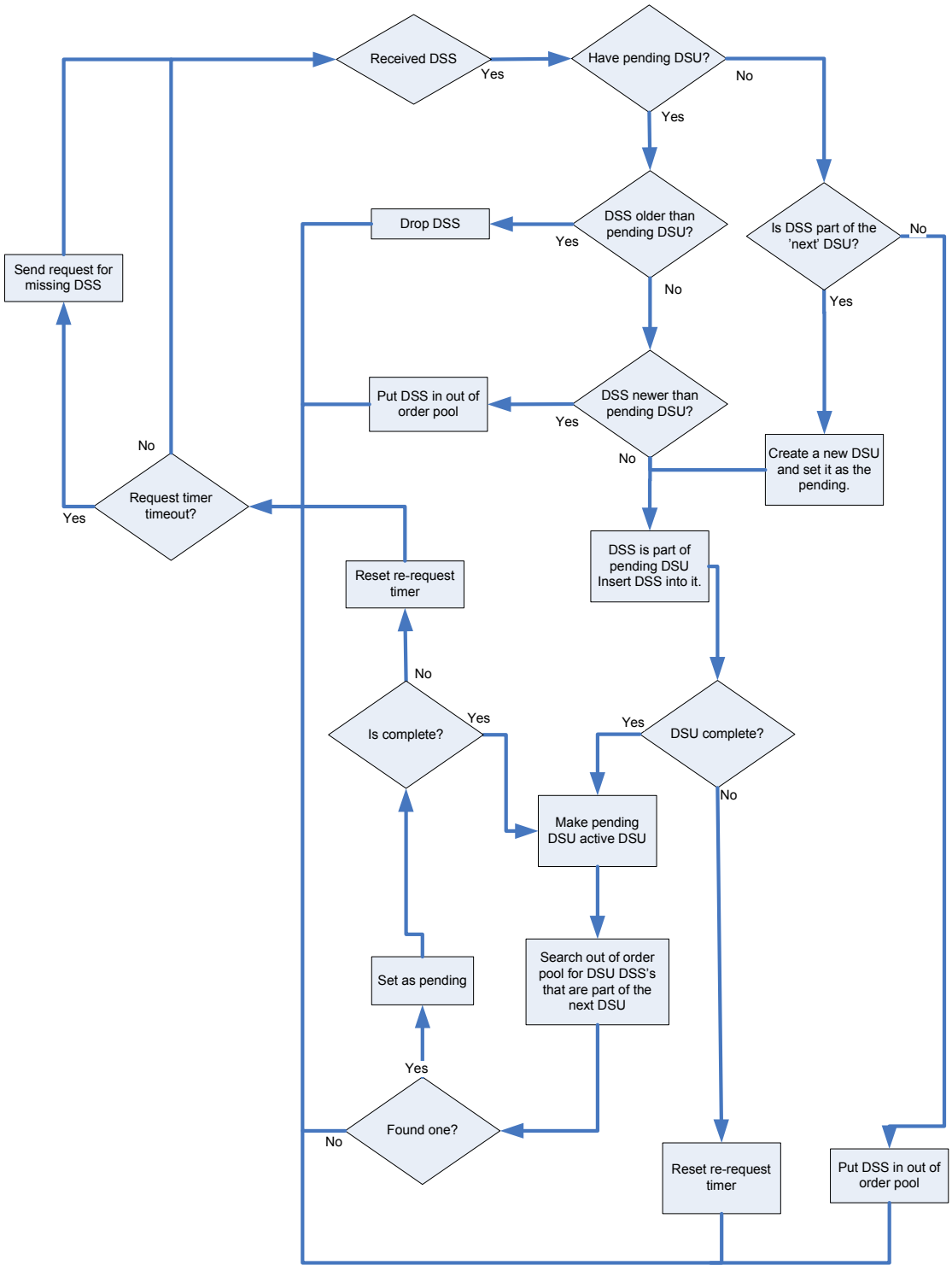
[25]Sally Floyd the work on Promoting the Use of End-to-End Congestion Control in the Internet
Sally Floyd and Kevin Fall
Submitted to IEEE/ACM Transactions on Networking
February 10, 1998
http://www.icir.org/floyd/papers/collapse.feb98.pdf

[26] "TCP/IP LEAN: Web Servers for Embedded Systems"
Jeremy Bentham, 2nd edition, CMP Books
ISBN: 1-57820-108-X

Appendix A. Server Side State Chart



Appendix B. Client Side State Chart



Appendix C. Client Side Building DSU