



20427 U.S. PTO

PROVISIONAL APPLICATION COVER SHEET [37 CFR 1.53(c)]

This is a request for filing a PROVISIONAL APPLICATION under 35 U.S.C. §111(b) and 37 CFR 1.51(a)(2)

112935 U.S. PTO
60/647601



Date : January 26, 2005
Docket No. : 54191/RAG/I343
EXPRESS MAIL NO. EV327914314US

INVENTOR(S)/APPLICANT(S) (LAST NAME, FIRST NAME, MIDDLE INITIAL, RESIDENCE (CITY AND EITHER STATE OR FOREIGN COUNTRY))

ZIJST, Erik van, De Weteringsbrugmolen 107, 1188GX Amstelveen, The Netherlands

TITLE OF THE INVENTION (280 characters max)

LIVE MULTICAST ON WIDE AREA IP NETWORKS

APPLICANT(S) STATUS UNDER 37 CFR § 1.27

☒ Applicant(s) and any others associated with it/them under § 1.27(a) are a SMALL ENTITY

ENCLOSED APPLICATION PARTS

☐ 110 Specification (number of pages including attachments)
☐ N/A Drawings (number of sheets)
☒ Other: 6 Attachments

FEE AND METHOD OF PAYMENT

☒ Our check which includes the filing fee of \$100.00.
☒ Our check which also includes the application size fee (for each additional set of 50 sheets, i.e., specification, claim, drawings, over 100) of \$125.00.
☒ Total Check Amount \$225
☒ The Commissioner is hereby authorized to charge any fees under 37 CFR 1.16 and 1.17 which may be required during the entire pendency of the application to Deposit Account No. 03-1728. Please show our docket number with any charge or credit to our Deposit Account. A copy of this letter is enclosed.

The invention was made by an agency of the United States Government or under a contract with an agency of the United States Government.

☒ No

Please address all correspondence to **CHRISTIE, PARKER & HALE, LLP, P.O. Box 7068, Pasadena, CA 91109-7068, U.S.A.**

Respectfully submitted,
CHRISTIE, PARKER & HALE, LLP

By

Robert A. Green
Reg. No. 28,301
626/795-9900

PROVISIONAL APPLICATION FILING ONLY

MAC PAS605220.1*-01/26/05 8:40 PM

Live Multicast on Wide Area IP Networks

Problem Description

The Internet has been tremendously popular and has seen a phenomenal growth during its few decades of existence. Especially since the birth of the World Wide Web the amount of online content has been overwhelming and steadily growing. Today the Internet. However, despite its versatility in digital communication, interoperability and internationally accepted communication protocols, its fundamental design has not changed much since its conception and it is easy to forget that the system does not just excel in everything. Watching live TV for example is something which has still not happened over the Internet, even though television has been around almost twice as long as the Internet Protocol and represents a huge market. The technical reasons for this are fairly fundamental. This thesis studies the Internet's shortcomings regarding live content and in particular content that is to be delivered to thousands or more subscribers simultaneously. Also, having identified the problematic areas, a software based system is introduced that can run on top of the existing, unmodified Internet as a virtual network and alleviates some of the issues, making the network more suitable for live data distribution to large audiences.

The Internet is a packet-switching network where data is exchanged in small units or packets that are independently transported over the network and concatenated again at the receiver into its original form. An important strength of packet-switching is that it allows for very flexible use of the physical network wires. When two communicating parties have no data to exchange for a certain period of time, no packets are sent and the wires can carry packets from other parties. More conventional systems such as the old telephone system, most of the cable television networks as well as radio are called circuit-switched. They reserve a fixed amount of capacity or bandwidth for each communication session, regardless of the channel's use. This fundamental difference has important consequences for the type of applications they support best. The cable television network for example is perfect for delivering a live video stream of constant quality and without noticeable delay because the necessary bandwidth was reserved on the network and can never be used by other streams. This of course also puts a hard limit

to the total number of concurrent streams. On the Internet, bandwidth is not reserved, but available to and shared by everyone. The consequence is that it cannot guarantee a minimum amount of end-to-end bandwidth, making live video streams often appear jerky and skip frames.

5

Even though with a little help from specialized protocols such as DVMRP or PIM, the Internet protocol allows for data packets to be multicast to a large number of receivers simultaneously, using this feature to successfully realize a live video broadcast can be quite a challenge. Assuming today's Internet would globally support multicast traffic, it could be used to multicast a live video stream. However, since the stream is transmitted in a fixed, usually rather high rate, it is entirely possible not all parts of the network have sufficient bandwidth available to forward the stream. When this happens, the router that finds itself receiving more data than it can forward, simply discards the packets that cannot immediately be forwarded. This causes two problems. Not only does this corrupt the data stream that is eventually received by one or more receivers further down the network, but also has negative impact on communication sessions of other nodes that go through this router. These packets will also be dropped with an equal probability. The only way to avoid this problem is to carefully find a transmission rate that can be supported by all parts of the network. Unfortunately, since the network is available to anyone, this rate will continuously change. Choosing a certain transmission rate and accepting the packet loss appears most practical. However, when packets are dropped randomly by overloaded routers, that means that every data stream will suffer the same percentage of packet loss. Therefore, if you send more packets, there is a larger chance that the router will choose one of yours when the overloaded link is ready to send another packet, implicitly rewarding heavy streams during congestion. If you added enough redundant packets to compensate for the loss of a significant part of them, you could effectively steal all link bandwidth away from the other applications. It would be fairer if the network by default divided the available resources equally among the users, not proportionally to their wishes. Currently the Internet is mainly accessed through the unicast TCP protocol, which carefully attempts not to overload the network by letting the receiver inform the sender about the transmission delay, packet loss and actual available

bandwidth, so the sender can slow down if packets are dropped. This flow control helps independent TCP streams to implicitly share the available bandwidth equally among each other, earning them the name of being “well behaved”. Applying this kind of active flow control to a live multicast is difficult for two reasons. First is that when feedback

5 information is sent from thousands of concurrent receivers to the source, this will put significant stress on the network close to the source and may even overload it if there are enough receivers. A second, more fundamental problem of flow control is that slowing down the data may not be an option in live streams. Accepting the packet loss in this case seems unavoidable and while audio and video data can usually even withstand quite some

10 packet loss without becoming too corrupted to play, this does not apply to all types of live data. Real-time financial data for example will become useless and even dangerous to use if random packets of trades are lost. Under these circumstances the network will evidently have to offer enough guarantees regarding the delivery to satisfy even those applications that cannot withstand the arbitrary packet loss introduced by overloaded

15 routers and links.

Multicast Routing

At the very least, the network needs to support one-to-many communication in that it can send data packets from a data source to more than one receiver, ideally without putting

20 extra stress on the network or source when the number of receivers increases. Multicast routing can be offered in various ways. The most common way is to let the receivers tell the network, but not necessarily the source, which streams they want to receive and let the network compute data distribution paths that deliver just the right packets to each receiver. Multicasting can also be done by letting the source encode the list of receivers

25 in each data packet, thereby freeing the network from the potentially computationally intensive task of maintaining multicast distribution paths. Although this approach usually does not scale to large audiences, applications exist. A third approach puts all logic at the receivers by letting the network apply a broadcast mechanism whereby each packet is delivered to every connected node and letting the receivers filter out only those packets

30 that are interesting. Although this may seem very wasteful, its simplicity can still make it

attractive under certain circumstances. In fact, even though it does not use packet-switching, conventional radio does exactly this.

Adequate Flow Control and Timely Delivery

- 5 Because non interactive live data streams do not actively anticipate network congestion, the network will need to manage the available bandwidth in a way that allows for fair or equal division among the streams. Without this, high volume streams will be assigned a larger capacity percentage of overloaded links and there will be little benefit in keeping the bandwidth requirements of a stream low, as the packet loss percentage is determined
- 10 by how much the network is overloaded by all streams combined and not by the requirements of the individual stream. An alternative to letting the network handle the flow control and congestion is to put the responsibility at the source and receivers. If this makes the stream well-behaved, network resources will be implicitly divided fairly in a way, similar to when only TCP connections are used. However, letting data streams
- 15 anticipate network conditions usually requires a form of feedback information from the network or the receivers. In this case it is important that the amount of feedback does not grow linearly with the size of the audience, as that would reduce the scalability of the multicast. Even when a scalable form of feedback information can be realised and the data stream adapts its transmission rate according to the network conditions, the problem
- 20 remains that live streams often loose their value when they are slowed down and delivered late .

Guaranteed Delivery

- Of course it would be ideal if every receiver would receive the data stream without loss or corruption. However, when the content is 'live' and cannot be slowed down, while the
- 25 network has insufficient capacity, packet loss is hard to avoid. In fact, even when there is sufficient bandwidth at all times, store-and-forward packet-switched networks usually cannot guarantee the delivery of all packets. For example, when a router crashes, all packets in its buffers will be irrecoverably lost. In networks that use dynamic routing,
- 30 packets can also be dropped when routing policies change or packets are trapped in an occasional, temporary routing loop. In cases where packets are 'accidentally' lost, an

end-to-end mechanism of retransmissions may be applied that can compensate for the loss, similar to retransmissions in TCP. However, since this requires a form of feedback information, it is again important for reasons of scalability that the overhead involved with retransmissions is not linearly related to the size of the audience. For TCP

5 connections this is no issue, as they only support a single receiver. Fortunately, end-to-end retransmission feedback can be avoided in at least two ways. First, it is possible to let the network components keep a copy of the most recently forwarded packets and let them participate in retransmissions by intercepting the retransmission requests and servicing them locally. This approach may lead to aggressive storage and increased processing

10 power requirements at the network components. The second alternative to end-to-end retransmission requests is that of encoding redundant information in the data packets. If enough redundancy is encoded, a lost packet's content can be entirely recovered from the extra information in the other packets. This technique is commonly known as Forward Error Correction and is applied in several systems. The downside of FEC is that it comes

15 with a constant level of bandwidth overhead that is related to the level of packet loss tolerance, regardless of whether packets are actually lost. Whichever approach to packet loss is taken, all will fail in case of a live data stream that produces more bytes than the network can forward. When local or end-to-end retransmission requests are used, the problem will even be increased as the retransmission requests use extra bandwidth,

20 causing more data packets to be lost, leading to even more retransmission requests.

Having identified the requirements for live multicasts over packet-switched networks, it is understandable why the largest publicly available network, the Internet, is less suited for this. Although IP natively recognizes multicast packets by their class-D destination

25 IP address, actual multicast routing is disabled on a large part of the network, making it unavailable to almost all end users. As of 1992 an application based overlay solution to IP multicast exists in the form of the Mbone experiment that uses software based multicast routers to allow native multicast network segments to be connected to each other, even while the global carrier networks that physically connect those network

30 segments do not support multicast. While the Mbone is over 12 years old and still in use at the time of writing, its penetration is still limited mainly to academic institutions and

larger ISPs. Hence, native multicast routing is not available to masses. However, even with a connection to the MBone, the issues of flow control and guaranteed delivery remain unsolved, seriously limiting the use of multicast traffic to a relatively small number of applications and content .

5

Given the requirements for live multicast and the lack of support for them by the Internet the problem of large-scale multicast is better dealt with in the application layer, rather than on the network layer. This leads to the popular research area of application layer multicast where a virtual network is constructed from software based router nodes that
10 employ their own routing algorithms and congestion control mechanisms to support efficient multicast distribution. Over the last decade or so, research in this area has lead to an impressive number of application layer multicast solutions. A list of solutions includes Scattercast, Narada, TrafficCore, Overcast, etc, etc. The fact that new initiatives continue to emerge illustrates the difficulty of solving multicast on packet-switched
15 networks in a generic way. This text introduces another application layer multicast solution that will be called *Terracast*. Although Terracast has many similarities with existing solutions, its specific focus on scalability and live data lead to a design that distinguishes itself from related work on a number of important points. As discussed in the previous chapter Terracast was designed to offer an end-to-end solution to efficiently
20 multicasting live stock market data to potentially anyone connected to the Internet. The fact that live market data is in many ways more demanding and much less tolerant to data loss than audio and video, meant that, aside from multicast routing and flow control, Terracast must be capable of giving certain hard guarantees over what is delivered to receivers. If packets must be dropped due to congestion or other irrecoverable problems,
25 it is crucial that this is done in a fully deterministic way that does not corrupt the financial data. Where a viewer of a film can perfectly accept the random loss of one or two video frames, this type of behavior can wreak havoc in financial data when the missed packet contained the update of a rarely traded company. The need for Terracast to be able to deterministically deliver only certain parts of a data stream when the network lacks
30 sufficient capacity, leads to another area of research that studies layered multicast. With layered multicast the packets of a live data stream are sent as individual streams .

This allows receivers to subscribe to only those layers that the network can handle, so that random packet loss can largely be avoided. For Terracast, an enhanced form of layered multicast is proposed that can even guarantee complete delivery for certain layers to avoid random loss altogether, making it suitable for market data. Essentially the two

5 core activities of Terracast can be isolated. The first is that of running and managing a robust and scalable overlay network that uses its own routing algorithms and supports multicast, while the other core activity is that of managing flow control and congestion when live streams overload the network and making sure layered multicast can be offered with guarantees.

10

Because Terracast combines two research subjects, it is difficult to find comparable, existing solutions. Because of this the evaluation of Terracast is divided in two and the part that manages the overlay network topology and multicast routing protocol are compared to similar, existing overlay products, while comparing the flow control,

15 bandwidth allocation scheme and layered multicast protocol to other existing layered multicast solutions.

Terracast Architecture

In this chapter is a detailed description of how the Terracast system works. However,

20 before the inner workings of the various components are discussed, a more high level overview of the system as a whole is given. This overview includes an explanation of how router daemons communicate, how they switch packets and how user applications can use the Terracast network. The overview section will also explain how routers and applications are addressed on the network, how unicast traffic works and how Terracast

25 supports single-sourced multicast. Although the section is quite lengthy, it does not explain the theory behind routing or packet discarding algorithms .

General Overview and Software Components

Being an overlay network, the Terracast network is essentially a number of software

30 router daemons, interconnected by normal TCP connections. Together this forms an

intricate web of routers and virtual links, Much like the Internet itself, but fundamentally different because it operates on the application layer. Each of these router daemons has a unique address and a routing algorithm computing shortest paths that allows each router to send data packets to any other router.

5

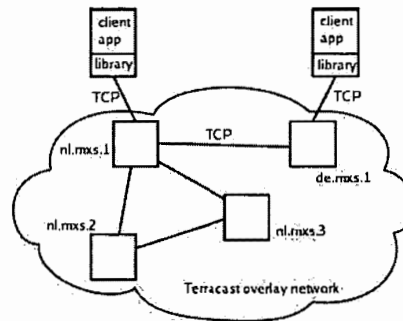


Illustration 1: A Terracast network is formed by software router daemons, interconnected by TCP connections

The system comes with a runtime client library and API for sending and receiving packets that can be used within user applications. This library connects to a nearby Terracast router daemon through a TCP connection or native inter-process communication. The latter is typically possible only if both the router and the client application run on the same physical machine. When connected, the user application can send and receive data packets from the Terracast network under the router's unique address. This mechanism is somewhat comparable to TIBCO's RendezVous product, where each host runs a long-lived daemon process that relays data packets from the TIBCO bus to the local client applications. In terms of addressing packets, an analogy can be made with a UNIX system offering TCP/IP connectivity to its user programs. The operating system's kernel is long-lived and can be compared in that sense by a Terracast router daemon. The kernel runs the TCP/IP protocol stack and carries the IP address. The user applications can use the kernel's protocol stack through library calls. When they want to receive packets from the network, they bind a logical IP port and packets addressed to the kernel's IP address and the IP port of the application will be sent to the application by the kernel. Terracast works similar in that only router daemons have a Terracast address, while user applications connected to a router daemon are identified

through logical ports¹. Illustration 1 shows how Terracast routers and client applications are connected. The diagram shows a Terracast overlay network of 4 router daemons, connected by a number of TCP connections. Two of the routers have a local client attached. The illustration also shows the unique, string-based addresses of the router daemons.

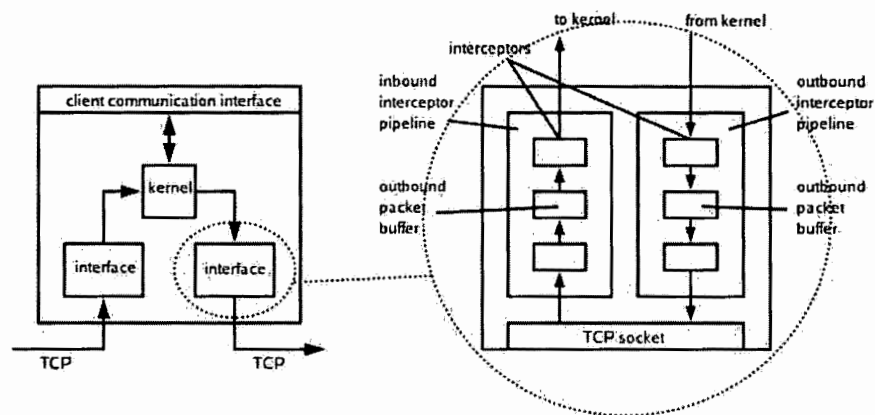


Illustration 2: A Terracast router daemon as showed on the left consists of a switching core or kernel and one or more interface modules that connect to neighbor routers. Using the unicast routing table, packets are forwarded from one interface to the other. The right hand side of the illustration shows the internals of an interface module. It shows how inbound and outbound packets are processed in separate interceptor pipelines and temporarily stored in packet buffers¹

- Internally, a router daemon consists of a number of components. At the heart of the router is the kernel or switching core that is responsible for forwarding packets between links. Preferably, it always processes every packet. The kernel gets its work from one or more interface modules. Each interface module manages a single TCP connection to a neighbor router. In illustration 1, the right-most router daemon has only a single neighbor node and therefore only a single interface module. When an interface module receives a

¹ The analogy made between an operating system's kernel running the IP protocol stack and a Terracast router daemon is illustrative, but not entirely correct. Among other things, the way IP addresses are assigned to interfaces is different from Terracast, where a single address is assigned to each router.

packet from its neighbor, it temporarily stores the packet in its inbound packet buffer and notifies the kernel thread. When the kernel has time, it takes the packet from the interface, looks at its destination and uses the routing table to decide through which interface the packet must be forwarded. The packet is then delivered to the selected

5 interface which temporarily stores it in its outbound packet buffer and transmits the packet as soon as the TCP connection is available. This flow of packets through the router is depicted in the left part of illustration 2. Aside from switching packets from one interface to the other, the kernel also maintains the unicast routing table that is used for switching. To make it possible for the network to find shortest paths as well as adjusting

10 these paths when the underlying physical network's characteristics change, each interface module constantly measures the quality of its virtual connection. These measurements are passed on to the kernel when the link quality was found to have changed. Inside the kernel, the measurements are fed to the routing algorithm to see if the changed link quality changed any of the routing entries. If the routing table was indeed changed, it is

15 advertised to the neighbors by encoding the entire table in a data packet and passing it to each interface. When such packet is received from a neighbor and propagated to the kernel through the receiving interface, the kernel inspects the message and runs it through the routing algorithm. Again, when the new information leads to changed routing entries, the router sends its own routing entry to all its neighbors. A similar thing happens when a

20 neighbor router crashes. The interface will detect this through an error on the TCP connection. It will then instantly change the link's cost indication to infinite and notify the kernel. Hence, the kernel does not see the difference between a crashed neighbor and an infinitely slow link. Note that the actual unicast routing table is omitted from the illustration.

25

The right hand side of the illustration shows the contents of an interface module. Because an interface constantly accepts outbound packets from the kernel that need to be transmitted to the neighbor node, it uses a buffer to temporarily store packets when the kernel delivers them faster than the TCP connection can send them. Incoming and

30 outgoing packets are processed separately inside the interface. It maintains an inbound and outbound pipeline of packet interceptors. Inside the pipeline the interceptors are

configured as a chain. A packet is passed to the first interceptor, which passes the packet on to the next interceptor until the packet re-appears at the end. The interceptor pipelines perform an important role for the interface in that they bridge between the network and the kernel and vice versa. For the outbound pipeline we have already showed that this

5 means it needs capacity to temporarily store packets when the TCP connection cannot keep up with the pace in which the kernel delivers new packets to the interface. The same issue exists in the inbound pipeline where a thread is continuously reading packets from the network. If at some point the interface receives packets from its TCP connection faster than the kernel can process them, packets need to be temporarily stored. Storing

10 packets is done in a special interceptor. This interceptor always accepts packets from its parent interceptor and buffers them. At the same time, the interceptor uses an internal thread to dequeue packets from the internal buffer and delegate them to the child interceptor. This way the parent interceptor will never be blocked when the children further down the pipeline cannot deliver their packets quickly enough. It is important to

15 keep a maximum size limit on the packet buffers to prevent them from exhausting the router's memory. In fact, storing packets before processing them also means that they will be delayed. The larger the buffer gets, the longer the packets will be delayed. Because of this, the interceptor will drop packets when the buffer reaches its maximum size. Since Terracast uses reliable TCP connections for transmitting packets between routers, packets

20 can only be dropped inside the buffer interceptors of the interface modules. Therefore this interceptor plays a key role in making packet loss deterministic². Because the buffer interceptors are implemented as normal packet interceptors, they can be configured before or after other interceptors. Examples of interceptors that would usually be configured in the outbound pipeline, after the buffer interceptor, include a traffic monitor

25 interceptor that measures the actual outgoing traffic rate. When incoming bandwidth is to

² Packets are not explicitly dropped in any other part of the Terracast network, but the buffer interceptor in the interface pipelines. However that does not guarantee that a packet that successfully makes it through all buffer interceptors of the network's routers is delivered at its destination. Aside from controllable packet loss, the network can also occasionally lose packets in a non-deterministic way, for example when a router crashes with pending packets in its buffers, or when a TCP connection between adjacent routers is unexpectedly closed during transmission of a packet .

be measured, a traffic monitor would be configured as the first inbound interface. With one interceptor before and one after the buffer, the interceptors can measure the amount of packet loss that is introduced, as well as the average packet delay due to buffering.

- Other examples of interceptors include an access control interceptor that can be
- 5 configured to drop packets from certain destinations or an interceptor, configured after the buffer, that actively limits the total throughput to ensure Terracast's use of the underlying network remains below a certain threshold.

- Every router daemon supports clients to use their services. This mechanism of accessing
- 10 the router's services was already briefly explained in the previous chapter. It is now explained in more detail. When a user application wants to send messages over the Terracast network, it uses the router's client library to program against. This library offers methods for sending and receiving packets, while in the background it uses regular TCP sockets or native inter-process communication to proxy these method invocations to the
- 15 Terracast router, although the current version of the software only offers an RPC mechanism for client to router communication. This is shown in illustration 3.

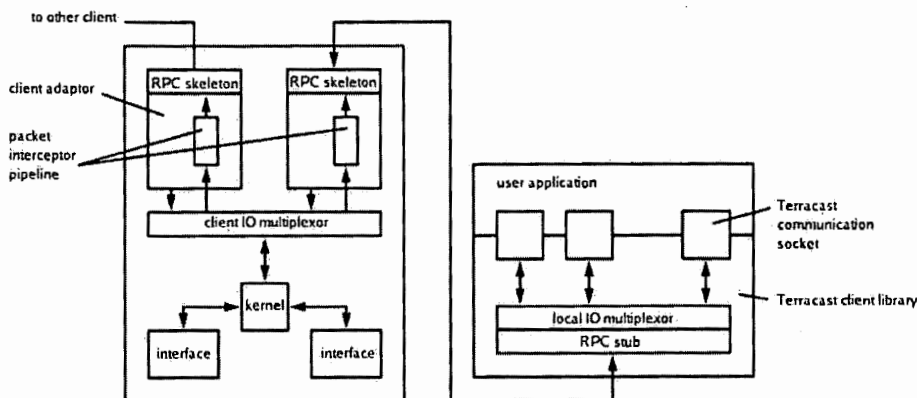


Illustration 3: User applications use the Terracast network through RPC using a client library. For each connected client, the router runs a client adaptor. When packets arrive addressed to a connected client, the router's client IO multiplexor passes it to the corresponding client adaptor where the packet is run through a packet interceptor pipeline and temporarily stored in a buffer interceptor until the client library actively fetches the pending packets³

On the right is a user application that uses the client library through instances of the Terracast communication sockets, essentially a socket in Terracast is comparable to a socket in TCP/IP. When a user wants to be able to receive packets, it needs to reserve a network address. A network address in Terracast is represented as a combination of the router's address and a unique port identifier, reserved for the client's socket. This is similar to IP where a unique network address, capable of receiving packets, is represented by an IP address and a port number. From here on, a port in Terracast will be referred to as a *tport* and a Terracast socket as a *tsocket*. Terracast node addresses as well as *tports* are represented by dotted strings, rather than numerical identifiers. A typical node address for a router daemon is `n1.ams.mxs.gw`. The same rules apply to *tport* identifiers³. The motivation behind stringbased addresses and the meaning of the dots is explained below when the unicast routing protocol is discussed. When a user application creates a *tsocket* for receiving normal unicast packets, the *tsocket* automatically binds a local *tport* at the router daemon through a remote call to the router's client IO

³ Note that node addresses have no relation to the Internet Domain Name System. The Terracast network is an independent network that uses strings to address nodes, rather than numbers. Because of the way nodes are named, their addresses will often resemble records from the DNS system.

multiplexor. Since tports are bound exclusively, no other clients can use it concurrently. Terracast distinguishes between two different network addresses, namely those used for sending and receiving unicast packets and those for publishing and receiving multicast packets. Both use independent network addresses. A typical unicast network address is
 5 nl.ams.mxs.gw:util.time that may be bound by an application that responds to requests for the local time. A typical multicast address is
 nl.ams.mxs.gw:quotes.nyse.sunw that may be bound by an application publishing the stock quotes for symbol SUNW on the NYSE. Syntactically there is no difference between unicast and multicast addresses. In fact, because unicast and the
 10 multicast addressing is independent, the address nl.ams.mxs.gw:util.time can be bound both as a unicast address as well as a multicast address by different applications at the same time without interference. Multicast distributions on Terracast are limited to a single source. While many existing multicast networks can offer multi sourced distribution, only few applications really achieve this functionality. In single-source
 15 multicast networks, the data stream is published by one source, while there are no restrictions on the number of receivers. Multicast group identifiers in these networks typically contain the node address of the multicast source. This is also true for Terracast, where a multicast group is formed by the source's address and some unique tport identifier, concatenated by a colon. Although many applications that would benefit from
 20 multisourced multicast can also be made to work with several single-sourced multicast groups, the multi-sourced paradigm remains more powerful in terms of loosely coupled services. While in most single-sourced schemes the source does not need to know about the presence or location of the receivers, multi-sourced schemes also apply this the other way around in that the receivers do not need to know anything about the source(s). This
 25 makes it trivial to move a multicast source application to a different location on the network, without the need for clients to anticipate this. Nevertheless, because single-sourced schemes are often technically more simple to realize and because Terracast's future role in the distribution of market data does not require it, Terracast was limited to single-sourced distributions only. When a source application connected to router *S* wants
 30 to publish a live data stream to multicast group *s:mygroup*, where applications on node *p* and *q* want to receive it, the source first binds the group in *publish-mode*. Binding a

multicast group in publish-mode means that the group is owned by the binding process. Only the owner of the group is able to publish to it. Note that in Terracast is impossible to bind multicast groups sourced at remote routers in publish-mode. Hence, the source application at connected to router *s* can bind group *s:mygroup* in publish-mode, but

5 never *q:mygroup*. When no other application connected to the same router had already bound the group earlier, the source application will succeed in binding and be ready to start its multicast. The sink applications connected to routers *p* and *q* now bind group *s:mygroup* in *subscribe-mode*. Binding a multicast group in subscribe-mode always succeeds and results in the router node being connected to virtual multicast distribution

10 tree. The subscribers will receive a copy of any packet published by the source. Multicast groups do not necessarily need to be bound in publish-mode first. Any application can subscribe to any multicast group at any time. If there is no source, there will be no data. Binding a multicast group either in publish-mode or subscribe-mode are different operations. The tsocket of a subscribe-mode group cannot be used to publish data, while

15 the tsocket of a publish-mode group cannot automatically receive the data packets it published. When the source also wants to receive a copy of its own stream, it will need to bind its group in subscribe-mode and use the resulting tsocket to read the data back. Data packets carry both the address of their source application, as well as the network address of their destination. Since addresses of unicast and multicast destinations are syntactically

20 equal, each data packet carries a special flag that indicated whether it is multicast or unicast. Router nodes use this flag to determine how the packet must be routed. For unicast packets, the router uses its unicast routing table to find the preferred next hop, while multicast packets are routed according to the subscription list in the router's multicast subscription table. As mentioned, a unicast data packet contains the node

25 address of the source router, the tport of the source application, the address of the destination router and the tport of the destination application. A multicast data packet however contains the multicast group it was published to, represented by the node address of the source router and the group identifier that was bound by the source application. It does not contain any other addressing information.

30

Unicast Routing Algorithm and Logical Address Summarization

An overlay network is a mesh in which every software router is connected to one or more other routers by means of a virtual connection. Two routers that are connected this way, are adjacent in the perspective of the overlay, but in reality may be many physical hops away from one another. Also, a router that has three neighbors has three independent virtual links. However, it is possible that this machine only has a single physical network

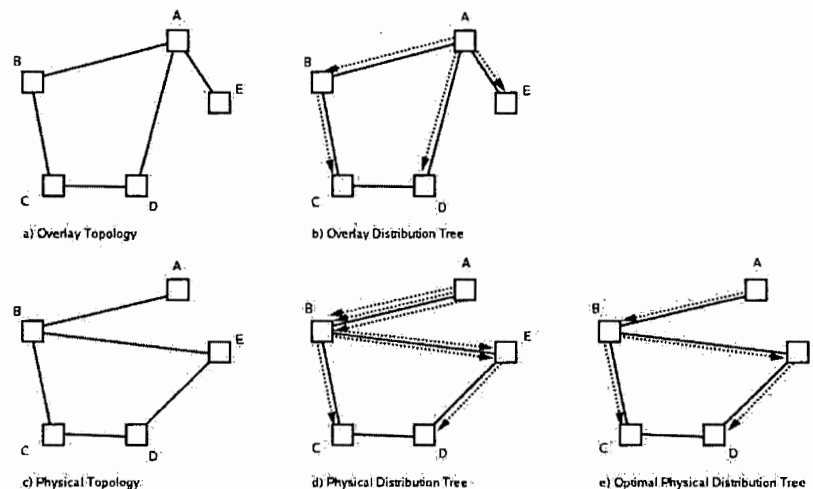


Illustration 4: Finding the best overlay network topology requires some knowledge of the physical network¶

connection that is shared by the three virtual links. Constructing an overlay mesh topology either manually or automatically by means of a specialized algorithm, requires knowledge of the physical network. This is illustrated in figure 4 where different overlay topologies are used on network. The first diagram a) shows a possible overlay topology based on 5 nodes and without knowledge of the physical network. Diagram b) shows what happens if node A starts a multicast distribution that the other nodes subscribe to. If it is assumed that the routing algorithm considers every link to be of the same quality, the multicast distribution tree would be as illustrated by the dotted arrows. Since node A has a direct overlay connection with B, D and E, it will send the data directly. Since there is no direct connection between A and C, node B will route the data on to C. From a virtual perspective the network looks quite healthy. Diagram c) shows the actual physical

network layout. We see that in reality there is no direct link between A and E, nor to D. This means that if the multicast distribution from diagram b) is run the physical data streams will be as depicted in diagram d). It shows that because the overlay topology does not match the physical network layout, several links carry two or even three copies of the multicast stream. In this network it is easy to see that when an overlay topology was chosen in which for every virtual connection there is direct physical link and vice versa, the multicast stream would run as depicted in diagram e). Unfortunately, this is often not possible in a real network as this rule would imply that the software router daemons must run at the actual IP router devices, instead of dedicated server machines.

However, when a server computer has only one physical network connection and no software routers are installed on the IP router devices, it is no longer possible to build an overlay network that strictly matches the physical layout, as each router daemon would be allowed to have only a single virtual link.

In the end, finding the optimal degree for each node is often a matter of finding the delicate balance between robustness and efficiency. Moreover, the optimal overlay topology changes when the topology or bandwidth characteristics of the underlying physical network changes. In Terracast, the overlay mesh topology is manually configured. This does of course not mean that the actual routing of packets on top of the overlay network is static as well. Because an overlay network shares the capacity of the physical network with other applications, the effective bandwidth of the overlay connections will constantly vary. This means that when the overlay topology offers more than one path between any two router pairs, the shortest paths on the network also change over time. To ensure packets are always routed along the currently shortest virtual path between source and destination, the network runs a fully adaptive routing protocol on top of the statically configured overlay mesh. The remainder of this chapter will discuss the design of the routing algorithm for Terracast.

Dynamic Routing - Distance-Vector Vs. Link-State

The problem of finding the shortest path between any two nodes in a network of arbitrary topology without prior knowledge of the network's layout, has been solved long ago.

When packet-switching was first introduced by Paul Baran and then independently a few years later by Donald Davies, its focus lay on military applications where it allowed networks with many redundant links and routers to survive destruction of a significantly large part of the network, without the need for manual intervention and reconfiguration.

- 5 Obviously this required routing algorithms running at the nodes and through ARPANET, Baran's early work influenced today's protocols. Currently two major classes of dynamic routing protocols exist that differ in the way shortest paths are computed and what routing updates are propagated to other routers. The first class of protocols are known as *distance-vector* protocols, the latter as *link-state* protocols .

10

Link-state protocols take the relatively simple approach of propagating the state of the entire network as a list of all known links in the network with their cost. This way, every node in the network will have complete overview of the topology of the entire network. Changes to the topology can be propagated quickly among all other nodes as the routing

15 updated can be forwarded unchanged. Finding the next hop in the shortest path between any two nodes can then later be done by running Dijkstra's algorithm on the network topology information. To be able to store the entire network topology in memory, the typical space complexity of a linkstate protocol is $O(N^2)$ where N represents the number of nodes in the network. Distance-vector protocols are based on the Bellman-Ford

20 protocol. They work by letting every router periodically advertise its own routing table to its neighbors. More correctly, it advertises the hops through which it will route packets for various destinations. One such entry is called a distance vector. The receiving router uses the distance vectors to adjust its own routing table and then derives its own set of distance vectors from this updated routing table and sends it to its direct neighbors. The

25 presence of new nodes it learned when a set of distance vectors contains a new destination address. An important consequence of the way routing information is exchanged is that the information is first processed and merged into the local routing table, after which a new routing update is derived from it. Because of the processing time required at each node, propagation of routing changes through the network is somewhat

30 delayed. An advantage of distance-vector protocols is that they maintain relatively little state information. The space complexity of this class of protocols is typically $O(N * e)$

where e is the average degree of a node and N represents the total number of nodes in the network. Some widely accepted implementations of distance vector protocols include RIP and Merlin-Segall. Although the use of localized information and customized routing updates makes the protocol friendly in terms of storage requirements, it is susceptible to longlived routing loops. This is probably the biggest disadvantage of the classical Bellman-Ford protocol and is illustrated in figure 5 with a simple network of 4 nodes and 3 links with equal weights.

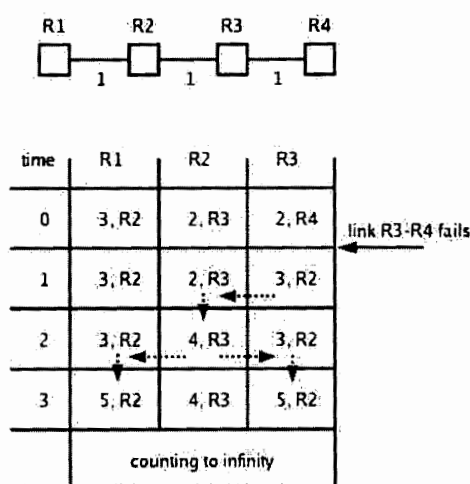


Illustration 5: The classical Bellman-Ford distance-vector protocol suffering from the counting-to-infinity problem when the link between node R3 and R4 fails after time 0.

Each router maintains a distance table with distances to every destination address for each link and from this table, a reduced routing table is derived that only contains the fastest route for each destination. The table in figure 5 shows what each router believes to be the shortest path to destination R4 at several time intervals. At time 0, after the Bellman-Ford protocol has successfully converged, we see that node R1 learned that it can reach node R4 through hop R2, with a total distance of 3. As R1 only has single link, this is its only path to R4. This is illustrated in the table by 3, R2. This means node R1's shortest path to destination R4 has distance 3 and goes through hop R2. Node R2 knows 2 possible paths to R4. One through each of its neighbors. The fastest path is that through

hop R3 with a total distance of 2. However, because Bellman-Ford periodically lets every node advertise its distance routing table, R1 has told R2 about its path to R4 with distance 3. When processing this distance vector from its neighbor, R2 now thinks it can reach R4 through R1 with total distance 4 (distance 3 to go from R1 to R4 and another 1 to go from R2 to R1). Node R3 also has two paths to R4. One directly through hop R4 with distance 1 and another path that goes through R2 with a distance 3. When the link between nodes R3 and R4 fails after time 0, this is detected by R4. In response to the link failure, R3 removes its fastest path to R4 from its distance table and concludes that it will now use its sub-optimal path that goes through R2. This is depicted in the last column of the second row of the table in figure 5. Because it updated its routing table, Bellman-Ford advertises the new paths to its neighbor R2. R2 understands that the distance of the path through R3 has now changed from 2 to 4. That leaves it with 2 paths, both with distance 4. Of course, none of these paths will actually work, but the lack of overall topology knowledge hides this from the protocol. If we assume, at time = 2, that R2 will choose the path through R3 to be the preferred path, this is advertised to its neighbors, illustrated by the two dotted arrows to R1 and R3. When R1 receives the update, it learns that its path to R4 through R2 no longer has distance 3, but 5. At node R3 the same thing happens. This is illustrated by the dotted arrows. In the next iteration at time = 3, both R1 and R3 will advertise their new distance 5 to node R2. In turn, R2 will again increase its distance and advertise it. This process is called the *counting-to-infinity* problem, as the distances will continue to be increased in every iteration of the algorithm. This problem effectively leads to a long-lived routing loop and data packets addressed to the disconnected node, will continue to circulate among the routers that created the loop. There are a number of solutions to the counting-to-infinity problem. One workaround is to put a hard limit to any path's distance. With a hard limit of 16 for example, counting-to-infinity becomes counting-to-16 and no longer poses a serious problem when the routing loop has a limited length. Unfortunately it has the side effect that nodes that are more than 15 hops away from each other cannot communicate. This workaround was applied to the RIP protocol that managed the ARPANET in the seventies. Two other solutions are known as *poisoned-reverse* and *split-horizon*. Both techniques will prevent the network from illustration 5 from entering the counting-to-infinity routing loop by not advertising routes to neighbors

that have that neighbor as preferred hop. In illustration 5 that means that although R1 initially has a route to R4 through R2, it will not advertise this path to R2, knowing that if it did, it could potentially lead to a loop when R2 decides to use that path. Poisoned-reverse differs from split-horizon in that the first explicitly advertises the destination as being unreachable, while the latter simply omits the path in the advertisement. Unfortunately, neither poisoned-reverse, nor split-horizon can prevent all counting-to-infinity problems. This is illustrated in figure 6 that uses link cost metric values, rather than simple hop counts.

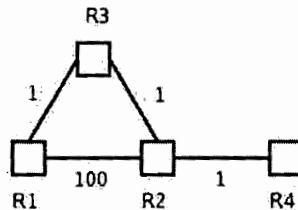


Illustration 6: Simple distance-vector network that uses split-horizon, but still suffers from a routing loop

We can see that the link between R1 and R2 is 100 times more expensive than the other links. When the network runs a distance-vector protocol that uses split-horizon, node R1 will advertise its distance vectors to R2. This includes the path to R4 (via R3 and R2) with distance 3. R2 will have 2 routes to node R4. One directly to R4 with distance 1 and one via R1 with distance 103. Note that R3 will not advertise a route to R4 to R2 because of the split-horizon mechanism. Now when the link between R2 and R4 fails, R2's preferred hop for destination R4 changes from R4 to R1. This creates a routing loop as well as the counting-to-infinity problem. R2 advertises its new route to R3 (not to R1, due to split-horizon). R3 only had one route to R4 (via R2) and learns the new distance of 104, which it advertises to R1. R1 now updates its distance to R4 to 105 and advertises it to R2. This continues until the distance hits infinity or a pre-defined hard limit.

Whether a link-state protocol is better than a distance-vector depends on the characteristics of the network. It is interesting to see that the Internet's predecessor, the

ARPANET, initially used the Bellman-Ford-based RIP protocol, but because of its long-lived loops was replaced by the SPF (Shortest Path First) link-state protocol in 1979 after almost a decade of research. In 1987, the protocol was given a revision that made its reaction to topology changes less aggressive by applying exponential averaging over link costs. After SPF, OSPF was introduced and gone through several versions. In 1998, OSPF Version 2 was introduced and remains one of the most widely deployed protocols for intradomain routing. In 1999 OSPF Version 3 also provided support for IPv6. After RIP, the ARPANET and later the Internet never switched back to a distance-vector protocol. However, since 1979 many new distance-vector algorithms have been designed that structurally solve the long-lived loops that killed the Bellman-Ford protocol in the ARPANET. The Merlin-Segall algorithm mentioned is among them. Another important one is the Extended Bellman-Ford or ExBF protocol proposed by Cheng and others. Merlin-Segall avoids all routing loops, both short-lived and long-lived, while ExBF only prevents all long-lived loops. An interesting comparison between the SPF link-state protocol and the newer, loop-free distance-vectors MS and ExBF showed that ExBF is as good as SPF under most circumstances with a much lower space complexity. MS generally performs worst.

ExBF - The Extended Bellman-Ford Protocol

Following the outcomes of previous comparisons between the more modern, loop-free distancevector protocols and link-state protocols, the Extended Bellman-Ford protocol, hereafter referred to as ExBF, is used as the basis of the Terracast overlay network. Longlived routing loops in distance-vector algorithms have their origins in the lack of global topological knowledge at the routers. Because of this, routers cannot foresee the possible consequences of advertising paths. As discussed in the previous paragraph, classical tricks offered by poisoned-reverse and split-horizon can only circumvent loops between adjacent nodes, but still fail to avoid loops formed by a collection of nodes. Link-state protocols do not suffer from this problem as they always have a complete overview of the network's topology. This allows them to compute their place in the shortest path between any two nodes and prevent data packets for a destination from ever

being forwarded to a neighbor that is lower in the shortest path tree originating in that destination.

j	d	n	h
A	0	*	*
B	3	F	E
C	4	F	B
D	6	F	G
E	2	F	F
F	1	F	A
G	5	F	C

5 *Illustration 7: ExBF uses its head-of-path information to backtrack each path through the network. The figure shows the routing table of node A, where the columns j, d, n and h represent the destination, distance, preferred neighbor and head-of-path. By recursively following the head-of-path nodes, the entire path to destination B is traced¶*

10

In the ExBF protocol [CHE89], the authors have slightly increased the amount of information is kept for every destination. Instead of storing just the distance to each destination for every neighbor, ExBF also stored the address of the pre-final node of each path. Hence, instead of storing the collection of distances $\{D_i(j)\}$ where j represents a destination, D the distance between this node and j , while i ranges over the node's neighbors, each router also stores $\{P_i(j)\}$ where P is the pre-final, or so-called *head-of-path* node in the shortest path to destination j . Now because the router knows the pre-final node of every destination, it can backtrack the full path to any destination by recursively looking at the head-of-path of a destination and treating it as a new

20 destination. This is illustrated in figure 7 where the routing table of node A is given.

Tracing the full path from this node to destination B goes by first looking up the head-of-path to B which is E and then using E as a destination. The head-of-path to E is F, while the head to F is node A itself. This concludes the reverse trace of the path that lead to A, F, E, B. To avoid long-lived routing loops, ExBF uses this information to apply the

25 following trick. When node i advertises a routing entry to destination j , it refrains from sending it to any neighbors whose address is in the path $n_0, n_1, \dots, n_r, j$, where n_0 is

i 's next hop to j and n is the head of the path. For the example of illustration 7 that means that node A will not advertise its path to destination B, E or F to its neighbor F.

Instead, it will explicitly report that A cannot reach those destinations.

5 Hierarchical Addressing in ExBF

- Adaptive routing protocols such as distance-vector and link-state protocols maintain routing tables that contain a routing entry to each destination in the complete network. This way each packet can be forwarded directly to the neighbor that is in the shortest path toward the destination. When something changes in the network topology or available
- 10 bandwidth, this may trigger changes to certain routing entries. In link-state protocols these changes need to be propagated to all nodes in the network as every host maintains an image of the complete network topology. With distance vector protocols the situation is usually slightly better, but some changes also need to be propagated throughout the entire network. The obvious problem is the limited scalability of these routing tables.
- 15 When the network grows in size, so do the routing tables. This requires more bandwidth to exchange the necessary routing data between hosts. Also, the more links there are, the more often something will change in the network. These factors severely limit the scalability of these protocols. To overcome the problem of growing routing tables, computation time and excessive advertisement overhead, large networks are often
- 20 partitioned into smaller sub sections, connected by so-called gateways. While the gateway routers maintain routing information necessary to route packets to nodes in other network sections, nodes inside a section or domain only maintain information for those nodes inside the same domain. This mechanism introduces a form of hierarchy that allows the network as a whole to grow far beyond the practical limits of standard
- 25 distance-vector or link-state algorithms. In most hierarchical routing algorithms, the network's division in domains subdomains is reflected in its node addresses. This makes it possible for a routing table to substitute a set of destination addresses by a single wildcard entry. The further away groups of destination addresses are, the more efficient aggregation of them can be applied. This approach does come with the consequence that
- 30 connecting new nodes to the network requires the assignment of an address that is not only unique throughout the network, but also suits the range of addresses used by its

neighbors. Node addresses now become location based. Hierarchical addressing schemes have been studied extensively in the past, including their applicability to distance-vector and link-state protocols. An important and well-known application of hierarchical addressing to allow for aggregation of routing tables on the Internet is Classless Inter-

5 Domain Routing or CIDR, which was introduced to allow for smaller subdomains and more flexible and efficient aggregation than what was possible with the classical, pre-defined network classes A, B and C. Other examples of applications of hierarchical routing include the ATM PNNI standard and the Internet Nimrod architecture. Address aggregation or summarization can often significantly increase the network's scalability,

10 but because groups of nodes or even groups of entire subdomains, are reached through a single routing entry, the path taken by data packets is often not the optimal path. The more summarization is applied, the less efficient the paths become on average. The factor by which the actual data paths on summarized networks differ from the optimal paths, is known as the *stretch-factor*: the maximum ratio between the length of a route computed

15 by the routing algorithm and that of a shortest path connecting the same pair of nodes.

Terracast takes a relatively straightforward approach to address summarization on its overlay network. First, an administrator decides on deploy time which nodes form clusters and which clusters form aggregations of clusters. This is done by encoding hierarchy in the Terracast node addresses using a dotted notation. As mentioned earlier,

20 node addresses are ASCII strings of at most 127 characters. Only [a-z] and [0-9] are available. Addresses are case-insensitive. An example of how a Terracast network is divided into logical clusters through hierarchical addresses is illustrated in figure 8.

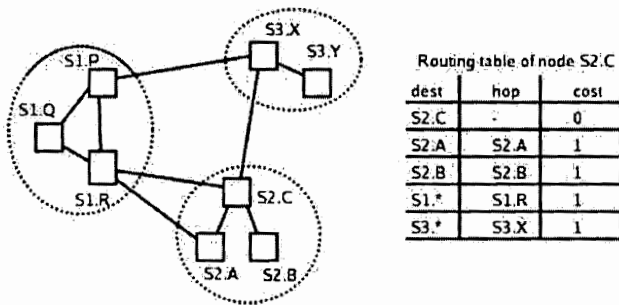


Illustration 8: Example network with hierarchical, location-based node addresses that define logical subnets or domains⁴

Here we see a network of 8 nodes, divided into 3 clusters. Assigning nodes to clusters is typically based on geographical properties, administrative boundaries and wide area links. All nodes inside a corporate network would typically all be assigned the same logical domain, whereas a Terracast network that connects nodes from different corporations, would usually assign a separate domain to each corporate network part. Another criterion may be that nodes that often disconnect and reconnect again later – possibly because these Terracast software routers run on PCs – are placed in a subdomain, to avoid the routing updates triggered by their changing state to propagate far into the network. Given the logical domains that now cluster groups of nearby nodes, each node can treat domains other than its own as a single entity and use a wildcard address that matches every host inside that domain. This reduces the size of the routing table, as well as the amount of routing information that needs to be exchanged between the nodes when the topology changes inside a domain. When a new host is added to domain S1, there is no need to propagate that information to the other domains, as they already have a wildcard entry that will match the new address. Illustration 8 also shows the effect of address summarization in this network on the routing table of node S2.C.

The cost value that is listed in the third column represents the cost of the shortest path to the nearest node inside that domain. In the 4th routing entry, that is the cost to reach S1.R from S2.C. If it is assumed that both interdomain links S2.C-S1.R and S2.A-S1.R have the same weight or cost, S2.C will route all traffic for domain S1 through neighbor S1.R. When nodes exchange distance vectors, either because of a link change, or as a normal, periodic exchange, the receiving node first summarizes the destination address of each distance vector relative to its own address. Summarization is done by matching the destination address with its own address field-by-field⁴ and when a field does not match, the rest of the address is substituted by a wildcard. When S2.C receives a distance-vector

⁴ Note that a network will typically have a hierarchy of multiple nested domains, rather than the 2 level of the example. This is much like the real world where a street can be seen as a domain inside a city and a city is a domain inside a country, etc.

from S1.R that contains a path to destination S1.P, it is immediately summarized to S1.* upon arrival at node S2.C. This is because the first field differs from the first field of the local address and as such the remaining fields are replaced by a wildcard. This wildcard is then fed to the distance vector algorithm that checks whether the new cost or path-length is shorter than the cost of the entry that was already in the routing table. In the present example there already is an S1.* wildcard entry in the routing table that was derived from neighbor destination S1.R. Since the path to S1.P runs through S1.R, the path to S1.R will always be shorter than the path to S1.P, so the entry in the routing table will not be changed and no further routing updates will be propagated to S2.C's neighbors. In general, when the local address is 1.2.3.4 and an incoming distance-vector advertises destination 1.2.2.2, it will be summarized to 1.2.2.*. Destination 2.1.6 becomes 2.*, destination 1.2.3.4.5.6 becomes 1.2.3.4.5.*, destination 1.2.3.5 stays 1.2.3.5 and destination 1.2.3.4.5 also stays 1.2.3.4.5. Note that 1.2.3 and 1.2.3.* are two different addresses. The first only matches the exact address 1.2.3 while the second is a wildcard that matches everything that starts with 1.2.3 and has at least 4 address fields. This includes 1.2.3.4 and 1.2.3.4.5.6, but not 1.2.3. When this mechanism of address summarization is used in an ExBF implementation that carries triplets containing destination, cost and head-of-path attributes in its vectors, that means both the destination address are summarized, as well as the head-of-path address. The example network topology as depicted in illustration 9 is used to demonstrate the exchange of ExBF distance-vectors and the creation of summarized distance or routing tables, as well as the preemptive detection of long-lived routing loops.

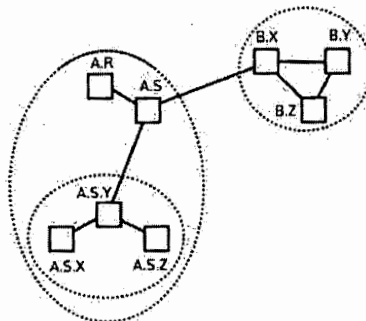


Illustration 9: Example network topology that to demonstrate head-of-path address summarization in ExBF¶¶We assume all links have cost 1¶¶

When the network first starts to converge, node B.X advertises the following routing information to neighbor A.S: $DV_{B.X,A.S}: \{(B.X, *, 0), (B.Y, B.X, 1), (B.Z, B.X, 1)\}$ ⁵.

- 5 When A.S receives the vectors, it summarizes its entries. What remains is the single vector $DV_{B.X,A.S}: \{(B.*, *, 0)\}$. The routing table of node A.S now contains $RT_{A.S}: \{(A.S, *, *, 0), (A.R, A.R, A.S, 1), (B.*, B.*, A.S, 1)\}$. Note that all addresses are summarized before processing. This includes neighbor addresses. The consequence of this is that when a node has more than one connection with a foreign domain, both neighbor
- 10 addresses will be summarized into the same wildcard. This leads to ambiguities and non-deterministic routing when this wildcard is listed as the preferred hop in a routing entry, as it can not identify a single outgoing link. This problem was solved in Terracast's implementation by assigning a local identifier to each link and using these numbers in the preferred hop column, rather than the addresses of the links peers. The routing update
- 15 A.S sends to A.S.Y contains $DV_{A.S,A.S.Y}: \{(A.S, *, 0), (A.R, A.S, 1), (B.*, A.S, 1)\}$ and leads to A.S.Y's routing table $RT_{A.S.Y}: \{(A.S.Y, *, *, 0), (A.S.X, A.S.X, A.S.Y, 1), (A.S.Z, A.S.Z, A.S.Y, 1), (A.S, A.S, A.S.Y, 1), (A.R, A.S, A.S, 2), (B.*, A.S, A.S, 2)\}$. When A.S.Y has finished updating its routing table, it advertises $DV_{A.S.Y,A.S}: \{(A.S.Y, *, 0), (A.S.X, A.S.Y, 1), (A.S.Z, A.S.Y, 1), (B.*, *, *), (A.S, *, *), (A.R, *, *)\}$ back to
- 20 neighbor A.S where the asterisk indicates unreachable in the last three records to avoid long-lived loops. These loops were detected through the normal backtrace mechanism of ExBF.

Summarization means that only a single path for a range of remote nodes is maintained.

- 25 As previously mentioned, the consequence of this is that packets will not always be

⁵ The following notation is used to describe a set of distance-vectors exchanged by peers: $DV_{from,to}: \{(destination, head-of-path, distance), \dots\}$. To describe the contents of a routing table, we use the notation: $RT_i: \{(destination, hop, head, distance), \dots\}$ to describe one or more entries from the routing table of node i. Note that records in a routing table only contain the information of the best known path to a destination, whereas a distance table is a data structure that contains information on all paths to a certain destination – one path for every connected neighbor node.

routed according to the real shortest path between source and destination. To illustrate this effect in the Terracast network, consider the example network of figure 9. Node S1.R receives the distance-vectors of both S2.A and S2.C. And after summarization learns that both neighbors offer a route for wildcard S2.*. If it is assumed that both interdomain

5 links (S1.R – S2.A and S1.R – S2.C) have equals costs, then S1.R will choose S2.A to be the preferred hop for S2.* because of the fact that its address logically comes first⁶. When S1.R now needs to forward a data packet to S2.B, it uses the S2.* wildcard entry and forwards the packet to neighbor S2.A. Unfortunately this is not the shortest path to S2.B, as S2.A first has to route the packet through S2.C. Instead, S1.R should have sent the

10 packet directly through neighbor S2.C. This ratio between the length of the actual path and the length of the optimal path between the endpoints is called the stretch factor.

The path between S1.R and S2.B is known to stretch as a consequence of the address summarization. Path stretching on Terracast can occur when packets are forwarded

15 between nodes that are in different logical domains. An entire subnet is treated as a single node with several outgoing links. Because the virtual node often contains a large number of nodes, connected by its own internal network structure, it is sometimes better to choose a different interdomain link when sending packets to the domain. Consider the left part of illustration 10.

⁶ When two or more neighbors offer a path to a destination with equal cost, any one of them can be used as the preferred hop. However, to avoid non-deterministic routing behavior, a router will always assign only one preferred hop for each routing entry. Which criterion is used to select one of them is irrelevant. In Terracast, the simple rule of selecting the neighbor with the address that is syntactically lowest is employed.

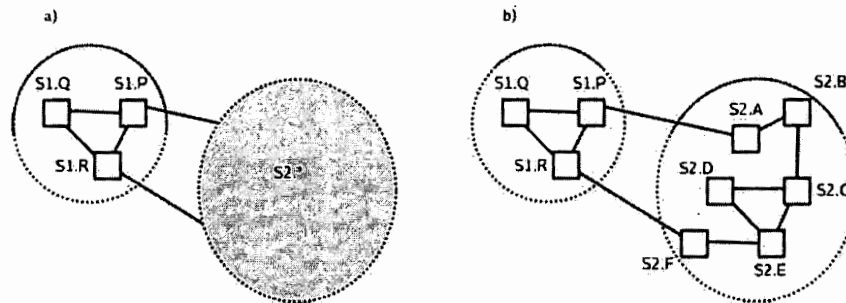


Illustration 10: This illustration shows the effect of decreasing the summarization opacity on the average stretch-factor. In illustration a) the default opacity is used, while illustration b) decreases the opacity level to get more detailed topology information¶

5

When node S1.P needs to forward a packet for destination S2.F, it will send the packet directly down its interdomain link to the S2 domain. In this example both interdomain links are assumed to have equal costs. The right side of illustration 10 shows the inner network of the S2 domain. And it becomes clear that the stretch-factor is quite high for packets from S1.P to S2.F, as the optimal path runs through S1.R instead. Although Terracast's address summarization technique can not guarantee a maximum upper bound on the stretch-factor, it can manipulate the stretch-factor by changing the *summarization opacity*. By default, an address is summarized after the first field that differs from the local address. However, if that is charged to the second field, terracast can look inside other domains for one level. A node with an address 1.2.3.4 will then summarize 2.3.4.5 into 2.3.*, rather than 2.* and 1.3.4.5 into 1.3.4.*, rather than 1.3.*. Doing this at least on the border nodes that have the interdomain links might be a good way to reduce the stretch-factor under certain circumstances. As the Terracast network was designed to be administrated by independent parties, administrators are free to experiment with different summarization opacity levels without jeopardizing the other subnets or domains.

As discussed above, address summarization can reduce the space complexity of the distance-vector protocol used. However, it is preferable to quantify by how much the scalability is improved, as well as finding out what the influence is of the having many or few nodes inside logical domains. Illustration 11 shows two possible network layouts

where the nodes in the network are systematically divided in a number of hierarchical domains. Figure a) shows a network of 8 nodes that are divided over 3 levels of hierarchy.

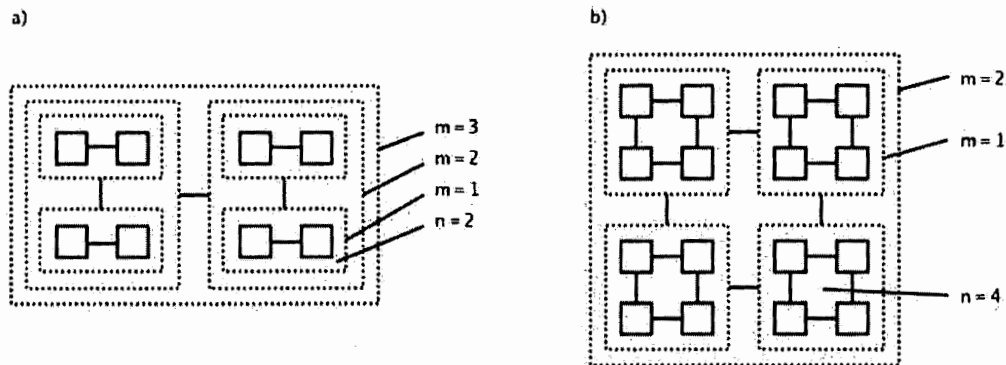


Illustration 11: Two systematically connected network topologies with hierarchy.

The levels of hierarchy are identified by $m = 1$ to $m = 3$. Each domain at level $m = 1$ contains 2 nodes and every domain at $m = 2$ contains 2 subdomains. The domain at $m = 3$ contains 2 subdomains from level $m = 2$. Figure b) shows a similarly well-structured topology where each domain contains 4 entities. This network is nested to a depth of $m = 2$. The illustrations define links between the direct nodes at level $m = 1$ and links between the dotted domain boundaries to show that there is at least one link between those domains. How many links there are between nodes from different logical domains is irrelevant with respect to the number of routing entries in the routing table of the distance-vector protocol. Since these tables only contain destinations with a single forwarding strategy, the number of interdomain links does not affect the size of the routing tables.

Now if y is identified to be the number of entries in a node's routing table, x to be the total number of nodes in the entire network, n to be the number of entities (nodes or nested domains) inside a domain and m to be the depth of the hierarchy, network a) from illustration 11 has $x = 8$, $n = 2$ and $m = 3$. From this it follows that the total number of nodes in the network can be calculated with:

$$\forall x=n^m \text{ where } n \in \mathbb{N} \wedge n > 1 \text{ and } m \in \mathbb{N} \wedge m > 0$$

- It is assumed that the relation between domain depth, number of nodes and domain
 5 density is expressed by the following formula:

$$y = \left(\frac{\ln(x)}{\ln(n)} \right) * (n-1) + 1$$

- This is proven by induction. First, take $n = 2$. Then it holds that $y = 2$ for $m = 1$. This
 can be seen in figure 11a. Now for each consecutive value of m , one summarized routing
 10 entry is added and it follows that

$$\begin{aligned} y &= 2 + m - 1 \\ &= m + 1 \end{aligned}$$

which is correct according to the hypothesis, because

$$\begin{aligned} y_{(2)} &= \left(\frac{\ln(2^m)}{\ln(2)} \right) * 1 + 1 \\ &= m + 1 \end{aligned}$$

- Analogue to what was seen for the proof of $n = 2$, it can be concluded that one
 15 summarized entry is required for each level when $n = n + 1$. So if there are m levels, this
 contributes to $1 * m = m$ more entries. Hence, $y_{(n+1)} = y_{(n)} + m$. It is next shown that this
 indeed follows from the formula we stated earlier. Assume that the formula holds for $y_{(n)}$,
 then

$$\begin{aligned} y_{(n+1)} &= \left(\frac{\ln((n+1)^m)}{\ln(n+1)} \right) * (n+1-1) + 1 \\ &= m * n + 1 \\ &= m(n-1) + 1 + m \\ &= \left(\frac{\ln(n^m)}{\ln(n)} \right) (n-1) + 1 + m \\ &= y_{(n)} + m \end{aligned}$$

- 20 Because the hypothesis holds for $n = 2$ while $y_{(n)} \rightarrow y_{(n+1)}$ for $n > 2$, the hypothesis holds for
 all n . Hence, it can be computed that the number of routing entries y each node maintains

in a structurally organized hierarchical network of m nested levels where each domain contains n entities with the formula

$$y = \left\lceil \frac{\ln(n^m)}{\ln(n)} \right\rceil * (n-1) + 1$$

Given this formula of the address summarization's effectiveness, scalability graphs are drawn for a large number of structurally organized network topologies. Figure 12 shows the size of the routing tables as function of the total number of nodes in the network for three different values of n . The graph uses logarithmic axes to allow for the reference function $y = x$ which represents the number of routing entries in each routing table when no address summarization is applied. What the graph clearly shows is that the number of routing entries in any network node remains under 100 even when the network as a whole grows to well over 10 million nodes. The graph also shows that the domain size n has no spectacular effect on the scalability. While a small value n yields a relatively deeply nested network hierarchy, which implies more routing entries, it also means that each hierarchy level only contains a small number of entities. A large value n in a network of the same size yields a relatively flat hierarchy with few levels, each level contains a large number of entities. It is important to understand that the points of the summarized graphs cannot be interpolated with a smooth line. As a routing table can only contain a

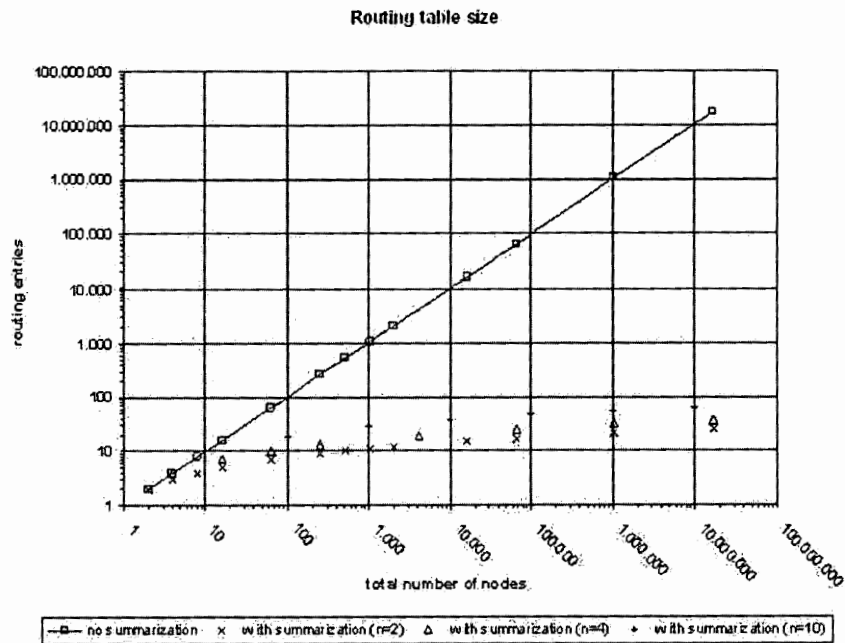


Illustration 12: This graph shows the effect address summarization has on the size of routing tables. The reference line $y=x$ shows a network that does not use summarization⁷

5

natural number of entries, its number remains constant as nodes are added to domains that are already captured by the existing wildcard entries⁷. The reason the $y = x$ function is interpolated is that without summarization, a new entry is added for every new network

⁷ In fact, more points for the graph cannot be computed, as the formula for computing the size of the routing tables requires both n and m to be natural numbers. This does not mean that the routing table size is undefined for any value x that cannot be computed by nm . When new nodes are added to the network in a structured, deterministic way, creating a new logical domain only when all others are full, the number of routing entries will remain constant until a new domain triggers a new wildcard. The reason the formula does not express this is because in between defined values of y , one or more domains will be partially filled with nodes or subdomains and because address summarization yields routing tables with localized wildcards, the number of entries will differ among nodes until all domains are filled up. Only then does every routing table contain the same number of entries. Obviously the actual wildcards differ between nodes from different domains.

node. Hence, if a point is drawn for every node on the x-axis, these points would be so numerous that they would appear as a smooth line.

Before this chapter is concluded on the unicast routing layer of the Terracast network, some consequences of using string based node addresses are discussed. A very strong advantage of such scheme is its large address space and flexible naming. New subdomains can be added at any time as there is no need for predefined or fixed hierarchical levels. All nodes inside a flat domain can be reconfigured into nested subdomains without the need to reconfigure routers or compute netmasks. Also, as address fields are not bound to a limited number of characters⁸, they can contain location or company names to make administration of the network easier and more transparent. An example might be the address nl.amsterdam.level3.marketxs.node4, or simply nl.ams.l3.mxs.n3 to keep the address relatively short. The drawback of these kind of unrestricted addresses is the amount of bytes they require. Because every data packet always contains a source and a destination address⁹, the amount of overhead per packet can be excessive at times. To reduce the amount of bandwidth wasted on this overhead, a mechanism is used for substituting the address strings by small 16 bit values when transmitting a packet to a neighbor. Also, a translation table is kept at both sides of the connection that is used to put the original address string back in the packet before passing it to the router's packet switching kernel. This optimization works on a per-link basis and is totally independent from the communication with other neighbors. When it is used on more than one interface, each interface maintains its own translation table. When a packet is transmitted to a neighbor, all address strings are stripped from the packet header. For

⁸ Of course an address as a whole does have a size limit. In the current version of the Terracast design, an address is limited to a total of 127 characters. This includes the separation dots.

⁹ In most cases a packet will contain more than two of these fields. Normal unicast data packets on Terracast travel between a source and a destination program and each program is identified by the combination of its router's node address, together with its logical tport identifier, as was explained in paragraph 5.1. Since a tport is syntactically equal to a node address, a unicast data packet normally carries 4 address fields.

each address string a 16 bit substitute value is generated and stored in the translation table. These 16 bits replace the original addresses and before the packet is transmitted, its protocol version field in the first header of its wire-level representation is changed to a higher version. This version number is reserved for optimized packet that would be unreadable for routers without support for address substitution. To ensure proper translation by the peer, the interface first sends a special packet that contains the new address substitutions. Such a packet is tagged with the higher protocol version number and contains one or more tuples of address string and its substitute value. It is important that this packet is received by the peer prior to the data packet itself, which is something that cannot be guaranteed by every transport layer. This is one of the reasons why Terracast purely uses TCP for communication between neighbors. Configuring an interface for address substitution can be done manually, or automatically. In the latter case the interface will use a special handshake packet that is part of the higher protocol version. A peer transmits such a packet when the connection is first established. When the packet is echoed back by the peer, the interface knows that the neighbor also supports substitution and goes into substitution mode. After a connection with a neighbor has been lost and is re-established, the handshake must be performed again as the peer could have been replaced by software that does not support substitution. Entries in the translation table all have a time-out. After this period the entry is removed from the table and any packet still using the substituted value will not be processed and cause the receiving node to respond with a special packet containing the 16 bit substituted values that weren't recognized, allowing the peer to synchronize translation tables by sending a packet containing the substitution tuples. Since the invalidation of timed-out entries is automatic, it is important both peers use the same time-out values. When the translation table is full, entries are best removed according to a last recently used scheme. While address substitution have not been implemented in the current version of Terracast, adding it could easily be done through the interceptor framework inside each interface. The substitution interceptor would normally be the lowest interceptor, reconstructing packets immediately on arrival. However, substitution will always introduce additional processing overhead and since application layer network routers already use plenty of that, enabling it may not be an option for every node.

Single-Source Multicast Routing

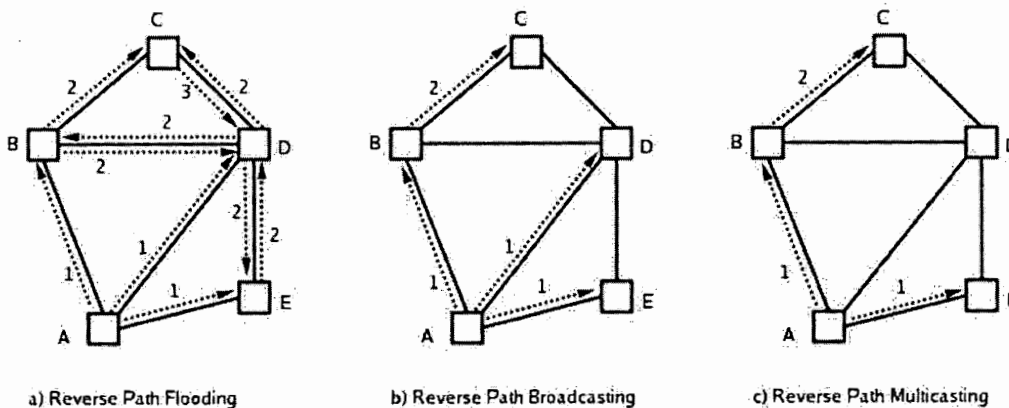
In the previous chapter the unicast routing layer operating on the Terracast overlay network is explained. It allows networks of arbitrary topology to dynamically discover the shortest paths to any destination node, while quickly anticipating crashing nodes and varying link characteristics. Although the Internet itself already offers the same basic functionality, the property of quickly converging to changing network conditions could in some cases already be enough reason to operate an overlay network, rather than directly using the Internet. This is because the Internet's inter-domain routing protocol does not always use the fastest paths between any pair of nodes. Nor does it converge to new routes immediately when important carrier domains go offline. The latter is often caused by the Border Gateway Protocol BGP that is run by backbone routers that connect different Internet domains and allows for dynamic routing between them. In many cases, BGP needs many minutes to reach a fully consistent view of the network after a fault. In contrast, switches in the public telephone network recover from link failure in the order of milliseconds. This makes the Internet sensitive to configuration errors, failure and abuse. This is reflected by the reports of the North American Network Operators' Group that shows that hardly any week is without serious disruptions among major ISPs. These persistent instabilities were the motivation for a series of academic studies into overlay networks that would be able to circumvent these issues and use their own application layer routing to recover from failures in the underlying IP network more quickly than BGP does. Aside from academic studies, many commercial offerings are available that essentially provide overlay services over the Internet with increased availability and end-to-end latency figures. These include Atrium Networks and Internap. Although Terracast could possibly also be used for this purpose, its main feature is to offer multicast support over existing IP networks. With this objective, it falls in the popular category of application layer multicast networks that include Narada, YOID, Scattercast and many others. Terracast however tries to go beyond offering plain multicast functionality and additionally focusses on supporting live multicast streams with limited jitter and delay through the combination of overlay multicast, multicast congestion control and layered multicast. In that sense Terracast distinguishes itself from most existing application layer

studies. Next is discussed the algorithm used by Terracast to build multicast distribution trees, using information from the unicast routing layer.

Multicast routing in packet-switched networks has a long history, but was added to the Internet Protocol relatively late. In a series of landmark papers Stephen Deering introduced multicast extensions for the Internet Protocol that eventually lead to the birth of the MBone in 1992. Deering's work is based on a strategy called *reverse path forwarding* or RPF. With RPF, data packets are not forwarded based in their destination address, but rather on their source address. A packet is only forwarded by a router when it was received over the link that would normally be used to send data packets to that source. RPF is therefore ideal to build implicit shortest path distribution trees from a source to all other nodes in the network¹⁰. In his 1990 paper "Multicast Routing in Datagram Internetworks and Extended LANs", which is incorporated herein by reference, Deering uses this technique to extend distance-vector protocols with multicast support. In its most basic version RPF is used to quickly flood packets originating at source S to all other nodes in the network. When a router receives a multipoint data packet sourced at S on the link that marked as the preferred hop for packets towards node S, then and only then is the multipoint packet forwarded to all links, except the link on which the packet was received. The result of this algorithm is illustrated in figure 13a. In this network node A sends a multipoint data packet to all its neighbors at time $t = 1$. The dotted arrows labeled "1" represent these data packets. When this packet is received by node B, the RPF check is done to see if the packet was received on the link which is also the preferred link in reaching node A. Since this is the case, node B reacts by forwarding the packet to all links, except the receiving link. This is done at time $t = 2$. Concurrently the same thing is done at nodes D and E. They too detect that the packet was received over their preferred link to A and respond by forwarding the packet to their other neighbors.

¹⁰ Because RPF builds shortest path distribution trees from source to receivers, based on distance figures that apply to the path in the opposite direction, RPF only works well in networks with symmetric links whose bandwidth and delay characteristics are the same in both directions. In reality this symmetry cannot always be assumed, especially with the rise of asymmetric "last-mile" technology such as xDSL and cable modems.

- All packets exchanged at $t = 2$ are labeled with a 2. At $t = 2$ node C receives 2 replicas of the multipoint packet on both its links. As the packet from B comes over the preferred link to A, C sends a packet to its other neighbor D at time $t = 3$. The illustration clearly shows the significant number of redundant packets that are exchanged. The link between
- 5 B and D for example carries 2 copies of the data packet, while both were unnecessary. To avoid the duplicate packets generated by the reverse path flooding algorithm, it is necessary for each router to know which of its links are *child links* in the shortest reverse path tree rooted at any source S. Then when a multipoint packet from S is received over the preferred links back to S, the router will forward the packet only over its child links.
- 10 Dalal and Metcalfe proposed a technique which involves periodic notification packets sent by each node to its parent node (preferred hop). Such notification informs the receiving node that it is the preferred hop for packets towards S. When a node receives a notification, it knows that this neighbor is a child in the shortest path tree rooted at S.



- Illustration 13: Different applications of reverse path forwarding. Figure a) shows how straightforward reverse path flooding is used to deliver multipoint packets to all nodes in the network. In figure b) reverse path broadcasting is used to eliminate duplicate packets while figure c) adds explicit membership information to ensure packets are only sent to nodes that need it.
- 20

Only those links will be used when forwarding multipoint packets. On a Terracast network, these notification packets are not necessary, as the ExBF protocol already

implicitly contains information on which links are child links. This is a by-product of the long-lived loop avoidance mechanism. In ExBF, a node will inform its neighbors about the fastest route it offers to a destination S, except the neighbor that is in the path towards S. Instead, an infinite distance is advertised to this neighbor. This information can be

5 used by a router to conclude that each neighbor advertising that node S is unreachable, is actually using it as their parent node towards S and hence will immediately mark those links as child links in the shortest path tree rooted at S. Neighbors that do advertise reachable routes to destination S will not be flagged as child links and therefore not be used when forwarding multipoint packets from S. The resulting multipoint forwarding

10 mechanism is known as *reverse path broadcasting* and is depicted in figure 13b. Note that this implicit information is also available in any distance-vector protocol that uses poisoned reverse. Both reverse path flooding and reverse path broadcasting can be used to quickly deliver a copy of any multipoint packet to all nodes in the network. The obvious disadvantage of both mechanisms however is that a copy of every packet is

15 delivered to every node, regardless of whether or not that node is actually interested in multipoint packets from that source. This may not be an important issue in small networks with few sources where usually all or most nodes are interested in receiving a copy of the data, but in large wide area networks with thousands of nodes and many independent multipoint sessions, it is crucial to the overall health of the network to

20 deliver packets only to those nodes who are interested in receiving them. This is called *reverse path multicasting* and can essentially be done in two ways. The first and probably oldest technique is called pruning. With pruning, multipoint packets are in principle still delivered to every node, unless a node explicitly informs its parent node that it does not wish to receive multipoint packets for a certain multicast group from a certain source.

25 When a router receives prune messages for a specific multicast group at S from all its child links, it will prune itself from the shortest path tree by sending a prune message to its preferred hop to S. Pruning reports are assigned a time-out value, after which forwarding for this multicast distribution tree is enabled again. The other mechanism for reverse path multicasting uses a technique opposite to pruning and instead uses

30 notifications to explicitly report interest in certain multicast groups. In its most common form, the network will not forward any multipoint packets over any link, unless a child

link is explicitly marked as being subscribed to the multicast group of a data packet.

When a node on the network becomes interested in receiving multicast packets for a certain multicast group, it will actively announce this. Again, subscription information is assigned time-outs and needs to be renewed frequently. The effect of reverse path

5 multicasting is illustrated in figure 13c. Protocols that use pruning are often referred to as dense-mode, while protocols using explicit membership reports are called sparse-mode.

Which type is better often depends on the type of network, and multicast interest patterns.

In a network with few multicast sources where everyone is interested, dense-mode

forwarding is often convenient, while networks with few subscribers per multicast group

10 are better off using a sparse-mode protocol. Sparse-mode forwarding is also important in networks with high volume streams, that cannot afford to temporarily forward these

streams over links that have insufficient capacity and no interested receivers. Examples of dense-mode protocols include Protocol Independent Multicast – Dense Mode, or PIM-DM and the Distance Vector Multicast Routing Protocol, or DVMRP which is the

15 multicast protocol used on the MBone. It is based on the previously discussed distance-vector RIP protocol. The most commonly used sparse- mode protocol is Protocol

Independent Multicast – Sparse Mode, or PIM-SM. As sparse-mode protocols are usually better suited for use over wide area networks, Terracast was equipped with such forwarding mechanism. Before plunging into describing this protocol in detail, different

20 types of sparse-mode multicast, their intended use and real-world implementations of them are discussed. Sparse-mode protocols come in to different flavors: those that offer full multisource multicast and those that only allow a single source per multicast group.

The latter are generally technically easier to implement and less complex. The first however are usually more powerful both because they better facilitate distributed

25 applications where all participants both consume and produce data, but also because in multi-source multicast both the receivers and the senders are anonymous. This anonymity for example makes it possible to switch to a different source application in a single-

source broadcast for example when the original source crashes. Because the actual

sender address is irrelevant in multi-source multicast protocols, the switch can be made

30 without the need to let the clients join a backup multicast group. This is different in most single-source multicast protocols that only allow a single source application per multicast

group, because here the multicast group often implicitly contains the unicast address of the source. As such, switching to a secondary source application running under a different unicast node address is different if not impossible as it requires the receivers to join a new multicast group. Well-known and established multi-source protocols are *Core Based Trees* or CBT, PIM-SM, PIM-DM and DVMRP. The latter two actually being dense-mode versions of multi-source protocols. Examples of the less versatile, single-source protocols include *Source Specific Multicast* or SSM and *EXPLICITly REQUESTed Single-Source* multicast, or Express. The increased flexibility of multi-source protocols comes at the expense of more complex implementations. This is illustrated by comparing a popular single-source protocol, SSM, with the most widely used multi-source protocol PIM-SM. Multicast distribution with PIM-SM begins as a shared tree, rooted at a central, well-known rendezvous point. All sources send their data packets to this rendezvous node, which in turn publishes the packets to the shared multicast tree that connects it to all receivers. In this mode of operation it is very similar to CBT. When new receivers join the multicast, they send a join message towards the rendezvous point, whose address is either known to the application, or was discovered through one of the various rendezvous point discovery mechanisms such as *Bootstrap Router*, *Auto-RP*, *Anycast RP*, *Embedded RP* or plain static configuration. In contrast to CBT, PIM-SM is also capable of setting up individual distribution trees for each source node. This has the advantage over a shared tree rooted at a more or less central rendezvous point, that data is delivered to all receivers over the shortest path between source and receiver. The downside of this of course is that it requires more overall membership state information in the network. Switching from the shared tree to one or more source-rooted trees is done by the last hop router that connects clients. When this router decides that it is more efficient to join the direct source-rooted distribution trees, it sends a join message directly to the sources and then detached itself from the shared tree by sending a leave message towards the rendezvous point. Being a single-source multicast protocol, SSM's operation is less complex than PIM-SM. In SSM a multicast group is identified by the an (S, G)-tuple, containing both the unicast address of the source node and a group identifier that needs to be unique inside the source node only. When a new receiver wants to join the SSM multicast, it must know the full (S, G) combination. Obtaining this identifier is left to the

application, using any kind of outbound mechanism such as a web page, directory server or email. The receiver then sends a join message directly towards the source node, leading to a new branch in the source-rooted multicast distribution tree. When choosing a multicast protocol, it is important to consider its intended use. While a true multi-source

5 protocol may give most flexibility, there are only few types of applications that could not do without it. Most many-to-many applications can also simulate multi-source multicast by letting every participant create its own single-sourced multicast group which every other participant joins, or by letting the nodes elect a master or proxy node that relays multicast traffic from all sources over a single multicast distribution tree, much like CBT

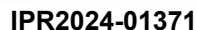
10 does. The intended use on the Terracast network is to have a single source multicast high volume market data or video streams to large numbers receivers. Here it is not important that receivers themselves or other nodes are also capable of publishing data packets to this multicast group. Instead, for security reasons it is even good to know that only the real source can publish to the group¹¹. Given these requirements, together with the fact

15 that multi-source protocols are more complex than their single-source alternatives, Terracast preferably uses a custom single-source protocol for multicast distribution on the Terracast network. Although this protocol has many similarities with SSM, the remainder of this paragraph explains it in detail.

20 Initially a source q is active, but as this is a sparse-mode protocol, with no receiver is yet subscribed, no distribution tree exists. When the first receiver application p_0 subscribes, its router initiates the reverse computation of one branch of the sink tree rooted at source q . It does so by first marking that p_0 now has a local subscriber for $q:G$ by setting the bit in $LS_{p0}[q,S]$ ($LS_p[]$ is a local array at node p that keeps track of all subscriptions of the

25 local user applications connected to router p) to true and then sending a $\langle \text{join } (q:G) \rangle$

¹¹ Of course, one can never blindly rely on the network's single-source properties for security. It will probably always be possible to inject malicious packets in a data stream by spoofing node addresses or compromising routers. Hence, for real security guarantees it will be necessary to use data encryption, electronically signed data packets, or a combination of both .



Ex. 1014, p. 45 of 112

44

The sparse-mode distribution tree generated when processes subscribe to a S:Q group originally equals a part (or all) of the optimal sink tree rooted at S. However, when the underlying unicast routing algorithm detects changes in the network performance and updates some of the routing table entries, the optimal sink tree changes accordingly and

5 may no longer be matched by the multicast distribution tree, rendering the tree as well as the multicast performance sub-optimal. An extreme example of this is when a link that is part of the multicast distribution fails entirely. In illustration 14b that happens when the link between u and w fails. At this point, the distribution tree becomes partitioned. When the underlying routing algorithm detects the failure (in Terracast as well as most

10 networks, routers actively detect link failure), it updates the routing table and selects the adjacent link to r with the second-best distance to q to become the new preferred hop towards q . To reflect this change in the distribution tree, node u invalidates its subscription at w by sending a <leave (q :G)> packet. In the example this leave packet can be omitted, as the link to w is unavailable. It then sends a <join (q :G)> packet to the

15 new preferred hop r . As an enhancement to minimize packet loss, node u would normally send the join packet first and optionally even wait for the first packet addressed to q :G to arrive before sending the leave packet. Aside from packet loss, this step could have the consequence of packet duplication. However, packet duplication, as well as loss are considered accepted behaviour on the Terracast network and it is left to the receiving

20 application or a specialized protocol layer on the receiving side to handle this.

When a node u currently in the distribution tree for q :G only has a single route to q via w , when the link between u and w fails, u concludes that it can no longer be part of the distribution tree (as u and q are no longer connected) and invalidates all child link

25 subscription information. In case $LS_u[q$:G] is set, u will rejoin the distribution tree as soon as a new path to q is found. Optionally the algorithm may send a notification to the locally subscribed application to indicate that u is no longer connected to q . This situation is illustrated in figure 15a. Here it is assumed that all links to have weight 1, while the link between p and w has weight 100. The result is that the link connecting p

30 and w will not be in the optimal sink tree rooted at q . Since only t and v have a local subscriber ($LS_t[q$:G] and $LS_v[q$:G] are set), the sparse distribution tree equals the dotted

arrows of figure 15a. Now it is also assumed that the underlying unicast routing algorithm is ExBF, nodes w , u , r and s all have only a single route to q ¹². Node p has a path through both s and w , while s is the preferred hop towards q . When the link between u and w fails, u becomes disconnected from q and invalidates the subscription information for link $u-r$. Node u informs neighbor r of this fact as part of an immediate distance-vector exchange. This routing update implicitly tells node r that its join for $q:G$ through u is no longer valid and that it should look for an alternative neighbor to rejoin.

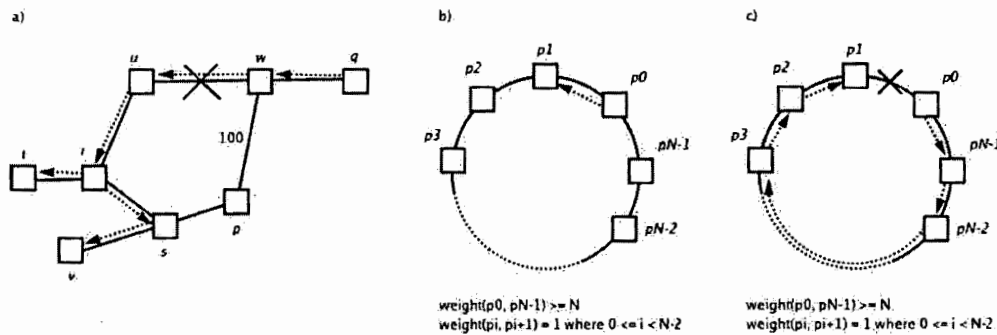


Illustration 15: Figure a: in some cases ExBF causes nodes to become disconnected to a finite period of time after a link failure, causing the recomputation of the distribution tree to become more complex. Figure b and c: when ExBF causes nodes to become temporarily disconnected while the unicast routing algorithm converges, recomputation of the distribution tree can take up to $3N$ time units. Hence, the worst-case time complexity for recovering from link failure is $3N$.

An explicit leave packet to u is unnecessary. Since r is also left with no alternative path to q and has no locally subscribed applications, it invalidates its $q:G$ subscription and is no longer part of the distribution tree. Neighbors t and s are implicitly informed about this through the ExBF routing update sent by r . Eventually w , u , r and s all leave the distribution tree, while nodes t and v schedule a rejoin when a new path to q is found. Again, they may inform their local subscriber applications about this. When p receives

¹² Note that in case of figure 15a, ExBF's loop avoidance mechanism lead to p advertising to s that q was unreachable. As a result, s was left with only one path to q (via r). The same applies to nodes r and u .

the routing update from S , it loses its preferred hop for q and switches to neighbor w . The new cost to q is now $100 + \text{distance}(w, q)$ and neighbor S is informed. Upon receipt, S discovers the new path to q and informs its neighbors r and v . Node v now sends the $\langle \text{join}(q:G) \rangle$ packet to S . Nodes S , p and w now construct the first part of the new,

5 optimal distribution tree rooted at q . When the new path reached r and t , t sends a join packet to r and r to S , reconnecting all receivers to the data stream. Effectively, recovering from the above situation is divided in three steps. In the first step the underlying ExBF routing protocol starts a wave that propagates the link failure through the network. Depending on the actual topology, this can take up to $N-1$ packets, where N

10 represents the number of nodes in the network. To understand this, consider the ring network of illustration 15b. Assume p_1 is subscribed to group $p_0:G$. Due to the link costs, the sink tree rooted at p_0 runs anti-clockwise from p_0 to p_{N-1} . The multicast distribution tree only contains the edge between p_0 and p_1 and nodes p_1 to p_{N-2} only have a single path to p_0 via their clockwise neighbor. Node p_{N-1} has both a path to p_0 via neighbor p_{N-2} (with

15 cost $N-1$) and via neighbor p_0 (with cost N). The preferred therefore runs through p_{N-2} . When the link between p_0 and p_1 fails, it is propagated through the ring in anti-clockwise direction from p_1 to p_{N-1} . When p_{N-1} loses its preferred route, it switches to neighbor p_0 and tells p_{N-2} about the new route. This update is passed through the network in clockwise direction from p_{N-1} to p_1 . When p_1 learns about the new path to p_0 after $2N$ time

20 units, it sends a $\langle \text{join}(p_0:G) \rangle$ packet to p_2 . This subscription is propagated through the network in $N-1$ time units and after it has been received by p_0 , the new multicast distribution tree is computed as depicted in figure 15c. It is concluded that the worst case time complexity for recovering from a link failure is $3N$ when ExBF is used as the underlying unicast routing algorithm. Note that when only p_{N-1} was subscribed to $p_0:G$,

25 time complexity would be down to $2N$.

Until now situations have been described in which a link that is part of the multicast distribution tree fails. More often however, links will not fail completely, but rather fluctuate in quality, causing the underlying routing algorithm to reassign preferred

30 neighbors. In this case it is not a hard requirement that the distribution tree is changed to reflect these changes, as it is not partitioned. However, since the tree became sub-

optimal, it could be preferred. In general, the decision whether or not to recompute a distribution tree after a routing table update, should be made according to the quality of the used path. When a node p in the tree detects that the neighbor in the optimal path towards the multicast source has a cost that is only marginally lower than the neighbor that is currently p 's parent in the actual distribution tree, the overhead of recomputation and the risk of packet loss may not weigh up to the increased performance of the new tree. Also, since the quality of each link is continuously re-evaluated, the cheaper sink tree may only be temporal. Two straightforward solutions to this are available. Either the algorithm could set a threshold on recomputation so that subscriptions are only moved to the neighbor with the currently optimal path towards the source if the decrease in cost is at least a ratio x , where $x > 1$. A larger value of x will then postpone tree adjustments until a substantial improvement can be gained, while a smaller x makes the tree actively follow the changing characteristics of the underlying network. Another solution could be to postpone recomputation of the tree for a period inversely proportional to the cost decrease ration x . The latter has the advantage that the optimal distribution tree is always guaranteed to be reached in finite time since the last routing table change. The current version of the Terracast software does not apply any of the above two recomputation strategies. After a distribution tree is constructed, it is not changed unless it becomes partitioned.

20

Reliable Layered Multicast

In the previous paragraphs the Terracast platform is described as an overlay network that runs its own, hierarchical unicast routing protocol to find the shortest paths over the mesh topology and comes with a single-source, sparse-mode multicast distribution algorithm that is similar to SSM. However, for most types of applications, offering best-effort multicast services is not sufficient. Like unicast applications, most multicast programs require some level of reliable communication. Examples of this are uploading files to several receivers simultaneously or replicating web server caches. Without guaranteed delivery, a multicast transport service has only limited applicability. Over time, many mechanisms for reliable multicast have been proposed in the literature. PGM, MTCP and

30

TRAM are examples of reliable multicast protocols that use selective packet retransmissions when a receiver detects a missed packet. This is comparable to the way most reliable unicast protocols guarantee delivery. Protocols such as RMDP and the work of Nonnenmacher apply forward error correction techniques to compensate for packet loss, without retransmission packets. Essentially, these reliable protocols either slow down the source to a speed all receivers can handle, or they signal the receiving applications individually when they are unable to keep up with the source due to bandwidth constraint or excessive packet loss. Slowing down the publisher is the same approach taken by unicast TCP sessions. Unfortunately this mechanism only suits bulk data transfers with no real-time requirement. Protocols that do not adjust the sender speed based on receiver feedback, are usually a better choice for live distributions, although they leave the source application with the important responsibility of choosing an appropriate transmission rate. Choosing a high transmission rate may lead to packet loss and corrupted streams for some receivers, while a low rate may lead to unnecessarily low quality data. The heterogeneity of wide area networks further complicates this problem. As one context of Terracast is financial data distribution, it needs a transport service that can multicast live data without packet loss or corruption and without substantial end-to-end delay when parts of the network are temporarily or permanently congested. At this point it is important to note that while live market data cannot tolerate random packet loss, it can be *thinned*. Under certain circumstances it is acceptable – often even preferred – to omit certain stock quote updates. An example is a desktop application that displays real-time updates for hundreds or sometimes even thousands of financial instruments. If all quote updates were delivered to this application, this would require a substantial amount of bandwidth and would cause the value of the more volatile instruments to change faster than a human can read. At the time of writing, most of the applications of this type work with these thinned data streams. Aside from end-user products, at least one major data vendor even thins the streams before broadcasting them to their customers¹³. For now it will suffice to say that the process involves inspection of the individual stock quotes and encoding them in individual data packets, tagging each with a priority value. As long as all packets labeled with the highest priority number are

¹³ Standard and Poor's Comstock delayed satellite feed.

received, the partial stream can be considered intact. Additionally, when all updates of the second highest priority are also received, the quality of the partial stream is increased – usually in the form of less latency. In Terracast, the priority numbers of the data packets represent the logical layers inside the data stream. When packets are labeled with priorities in the range 0 to 3, the stream is said to have 4 layers. Also, the convention is to treat 0 as the highest priority and 3 as the lowest. When a stream only contains a single layer, all packets are labeled with priority 0. To the software router daemons, a packet's priority value has no meaning other than that it is used as a criterion for discarding packets on congested links. When an outgoing link of a router has insufficient bandwidth to transmit all pending packets, it will forward only those with the highest priority. This technique may also be applied to other data types, such as audio/video data. It will then also be shown that the packet priority numbers cannot be misused by sources to make their data packets more important by giving them the highest priority, than those sent by other sources. Priorities can only be compared between packets that are part of the same data stream¹⁴. Knowing that routers will use the packet priority numbers when making forwarding selections on congested parts of the network, a source can carefully divide its data packets over different layers or priorities, in a way that a subset of the layers still contain a useful, uncorrupted representation of the data. This is especially useful when multicasting a live data stream with a high data rate to a large number of receivers, scattered over a heterogeneous wide area network. Receivers that are on congested parts of the network will then receive the most important parts of the data only. This eliminates the need to lower the publishing rate to match the slowest receiver, while still being able to offer live, uncorrupted, thinned data streams to clients suffering from a lack of sufficient bandwidth. This concept of layering a live multicast stream and allowing receivers to receive as many layers as bandwidth capacities permit, is an active area of research. One of the pioneers of layered multicast schemes are Steven McCanne and Van Jacobson who in 1996 divided a data stream in layers that were sent to separate IP multicast addresses. As long as bandwidth is available, McCanne's receivers

¹⁴ A data stream in Terracast is identified by its source address. For unicast streams this is the combination of unicast node address and a local tport. A multicast stream is identified by its multicast group name .

would subscribe to more and more layers until packet loss is detected. Loss of packets is detected by appending incremental sequence numbers to the packets and is interpreted by receivers as a lack of sufficient bandwidth or network congestion. To safeguard the quality of the layers, the algorithm unsubscribes a layer as soon as packets are being

5 missed. Aside from layered protocols, there is also active research in content and applications that would benefit from network functionality such as proposed by McCanne. Examples are audio and video codecs that divide live multimedia content over layers to enhance the user experience over wide area networks Terracast uses layered multicast in combination with a scalable, selective packet retransmission mechanism to

10 offer a service that can meet the demands of real-time financial data. Selective packet retransmissions are crucial so Terracast can guarantee that subscribers will receive one or more layers that are completely intact. When an occasional packet in a high priority layer is missed, it should be repaired. However, the receiver should not attempt to recover missing packets from low priority layers that are missing due to network congestion.

15 Trying to repair these layers would require even more bandwidth, resulting in even more packet loss. Yet, as more end-to-end bandwidth becomes available, the receiver should detect this and repair additional layers, so more intact layers can be delivered to the application. When layers are dropped due to congestion, the Terracast programming API that is exposed to user applications should notify the user by means of an exception or

20 special return value when it is reading packets. These notifications should be non fatal and only serve to inform the user that the quality of the stream is changed.

Localized Packet Retransmission

Terracast comes with a receiver-side multicast socket that receives the raw packets from

25 the multicast stream and is in charge of repairing damaged stream layers before forwarding them to the user application. To this end there is a retransmission request packet. When the receiver detects a missing packet in a layer that should be repaired, it sends a retransmission request for this packet towards the source. Detecting packet loss is done in the conventional way by sequencing each packet with an incrementing number.

30

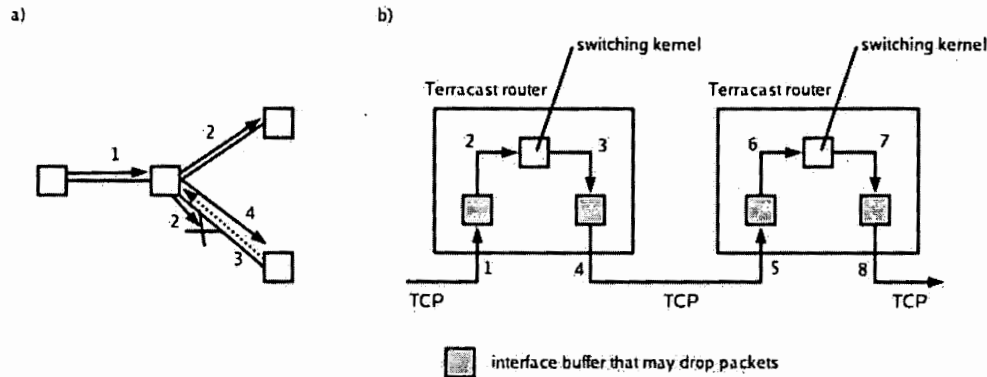


Illustration 16: Figure a) shows reliable multicast with tree-based negative acknowledgments. (1) Data packets are published by the source. (2) The branch point sends a copy of the packet to both neighbors, but one packet is lost. (3) The receiver detects the packet loss and sends a retransmission request to the source. (4) The branch point has a copy of the data packet buffered, responds to the request and suppresses the retransmission request. Figure b) shows possible places inside a Terracast router daemon where packets can be stored to service local retransmission requests. Because packets should be collected before they enter a component that may drop them, points 1 and 3 (5 and 7 respectively) are good, while 4 is not.

In the present case however, each layer has its own sequence number, so all packets in one layer are sequenced independent of the other layers. This way, packet loss can be detected in every layer, regardless of the conditions and packet loss in other layers. To avoid a cascade of retransmission requests when a packet is dropped close to the source, a tree-based negative acknowledgment is used similar to those used in PGM and TRAM. This technique can provide localized repair without communication with the source, because an intermediate router that has a copy of the requested data packet will respond by sending it again and suppress the retransmission request. This is visualized in illustration 16a. Localized repair works by having each router daemon store a window of transmitted packets. Packets are either stored for a fixed period of time, after which they are discarded, or a fixed amount of buffer space is reserved to store the most recently forwarded packets. Since the Terracast overlay network by default uses reliable TCP connections between router daemons, packets will never get lost while they are in transit

between nodes. The only place where packets are discarded is in the interface buffer interceptors that were discussed above. Hence, there is no use in buffering packets that were actually transmitted over the TCP connection, as those packets are guaranteed to have reached the neighbor node¹⁵. Instead, a Terracast router daemon should copy

5 packets to its retransmission packet store before those packets reach a buffer interceptor that may discard packets. Illustration 16b shows the router's buffer interceptors and strategic places to copy packets to the retransmission store. The gray components represent the interfaces with their interceptor pipelines. Because these are the only places where packets are discarded during congestion, the best places to copy packets to the

10 packet store are 1 and 3 (and 5 and 7 respectively). Data packets stored at 4 are guaranteed to have reached 5 and are better stored by the neighbor router daemon. A convenient way of implementing localized packet repair in Terracast is by adding an interceptor to both interceptor pipelines of each interface and letting those interceptors simply store a copy of each packet. Additionally, when a retransmission request passes

15 through the router on its way to the multicast source, the interceptor inspects the request packet and if it has the requested packet in its temporal packet store, it will not forward the retransmission request to the next interceptor, but inject the requested data packet into the network again. This system is illustrated in figure 17 where two adjacent router daemons are displayed. The picture shows the two interceptor pipelines inside the

20 interfaces. Each interface contains both an outbound and an inbound interceptor pipeline. The latter processes packets received from the network, while the outbound pipeline processes packets just before they leave the router daemon. The pipelines show two different interceptors; the CCI (Congestion Control Interceptor) and the PRI (Packet Retransmission Interceptor). In reality a router daemon usually contains more interceptor

25 instances per pipeline, but the most important one that is always present is the CCI which is the unit that implements Terracast's bandwidth allocation rules and is responsible for

¹⁵ As mentioned in paragraph 5.1., packets are not explicitly dropped in any other part of the Terracast network, but the buffer interceptor in the interface pipelines. However the network can occasionally lose packets in a non-deterministic way, for example when a router crashes with pending packets in its buffers, or when a TCP connection between adjacent routers is unexpectedly closed during transmission of a packet.

discarding packets. During normal operation the source publishes its multicast packet stream to its router daemon that sends the packets to all interfaces that lead to interested receivers according to the multicast routing table. In the present example that means the

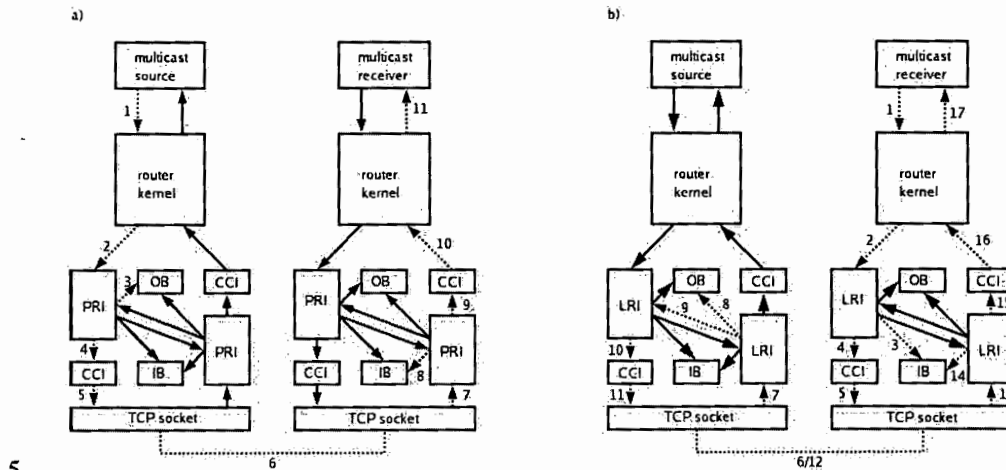


Illustration 17: Figure a) show two router daemons connected by a TCP link. The left router has a multicast source application publishing data packets, while the right router connects a multicast receiver that is subscribed to the multicast group. The figure shows the normal packet flow through the router components. In figure b) the same configuration is illustrated, but now the flow of packets is depicted when packet loss is detected by the receiver and a retransmission request packet is sent. The lost data packet is recovered by the interface of the left router.

- packets are passed to the first outbound interceptor (PRI) of the interface. The PRI is responsible for storing a copy of each recoverable packet that is transmitted. It stores the copies in the private temporary packet store of its interface. Then the interceptor passes the packets to the next interceptor (CCI) where it is buffered until the TCP connection with the adjacent router daemon has time to transfer it. If the packets are not dropped in this CCI, they are received by the neighbor router daemon and sent through that router's inbound interceptor pipeline. They first enter the PRI, which stores copies in its inbound buffer (IB in the illustration) and then passes them on to the next interceptor (CCI) where

they are temporarily parked until the router's kernel thread picks them up and delivers the packets to the subscribed multicast tsocket. When a packet is lost in the outbound packet interceptor pipeline of the source router because the TCP connection between the routers was not fast enough to transmit all the packets, the receiving tsocket will notice the loss and send a retransmission request. This process is depicted in illustration X. The destination address of this unicast retransmission request packet is that of the multicast group. The Terracast packet switching kernels recognize the retransmission packets and use the multicast group to route the packet towards the source of the group. The retransmission request packet of our example passes the router kernel and reaches the outbound packet interceptor pipeline and goes into the PRI (2). This interceptor scans specifically for retransmission requests and tries to answer them locally. To this end it inspects the packet, looks up the multicast group, the stream layer identifier and the layer's sequence number and then checks the inbound packet buffer (IB) to see if it contains the packet (3). In the present example this is not the case, so the packet is passed on to the rest of the interceptors (4) and eventually transmitted to the adjacent router (5, 6) where it is fed to the inbound interceptor pipeline (7). The PRI checks the interface's outbound packet buffer (OB) (8) and finds the multicast packet that was dropped by the congested outbound CCI earlier. It is then re-inserted into the outbound packet stream via the outbound PRI (9). The retransmission request packet is then dropped and not forwarded any further. Assuming the packet is not dropped again, it travels the normal way towards the receiver (10-17). At (14) the receiver's PRI sees the multicast packet for the first time and stores a copy in its outbound packet buffer. This mechanism of unicast retransmission packets and interceptors that offer localized repair is flexible in the way that not all router daemons need to support local packet repair. A router that does not support it simply forwards the requests and lets the upstream routers handle them. Of course the network as a whole becomes more efficient when more routers support it. Because storing a copy of every multicast data packet requires storage capacity, Terracast links the packet buffers of the packet retransmission interceptors to a central data structure that will only store unique packets. This is important because in multicast transmissions, a router that is a branch point in the multicast distribution tree will otherwise end up storing at least three copies of every data packet. This is because

the packet is received on one interface and sent out over at least two other interfaces. If every interface would individually store its packets, the inbound packet buffer of the receiving interface as well as the outbound packet buffers of the forwarding interfaces would contain the same packets. The central packet store eliminates duplicate packets by

5 storing each packet only once and by keeping reference tables that point to packets for every interface. When an interface buffer stores a new packet in the central store, while that same packet was already stored by another interface, the central packet store will merely add a pointer to that packet instance to the interface's packet reference table. Each entry in the reference tables has a time-out attached to it. This is to ensure packets are

10 only temporarily stored. Because the same packet could pass different interfaces at different times, the time-outs are not attached to the packets in the central store but rather to the packet references in the interface reference tables. A packet will only be expunged from the central store when all references to a certain packet have timed out. Another optimization that can substantially reduce the amount of required storage space is to only

15 store packets that were explicitly discarded by the Congestion Control Interceptors because of bandwidth constraints. Storing all other packets as well has little value, as those are guaranteed to have arrived at the next router. No retransmissions will be requested for them unless some non deterministic packet loss occurs as a result of a crashing node. However, even if a packet was lost close to the multicast source because

20 of this and all receivers simultaneously send a retransmission request, the upstream routers can still apply conventional nack-suppression techniques to combine all concurrent retransmission requests from its child links into a single request that is forwarded upstream towards the source. Although this is not expected to jeopardize scalability of retransmissions in large receiver groups, applying local packet stores at

25 every router is still preferred for live streams, as it reduces the latency of packet recovery. An interesting question is how long to store packets for local retransmissions. A short time-out minimizes storage requirements, but will also lead to more overhead towards the source as more retransmission requests need to be propagated further upstream. Large time-outs will be more tolerant to late retransmissions, but require more storage. It is

30 difficult to choose an optimal time-out value, but when live data is distributed, a large retransmission time-out is probably unnecessary. When a time-out of 60 seconds is used

by the routers that means a lost packet can still be recovered after a minute. However, when the data stream contains real-time stock quotes, such a delay in delivery cannot be tolerated. Problems get worse if the data packets also have to be delivered to the user applications in their original order. In that case the user may not receive any data for up to a minute while the missed packet is recovered and all newer packets are waiting for it. For this reason the current version of the Terracast network uses a 10 seconds packet retransmission time-out¹⁶. Another issue to consider is how to respond to a retransmission request for a packet that has timed-out. Without counter measures, requests for purged packets will simply be forwarded towards the source. However, when all routers in the network use the same time-out value, there is no use in propagating these requests as the upstream routers are guaranteed to have purged their copies as well. To avoid network implosions from slow receivers right after a packet is purged, each router interface remembers which packets it has purged. When a retransmission request is received for such packet, the interface will respond by sending a special packet to the receiver that indicates that the requested packet had timed-out and cannot be recovered anymore. The retransmission request will not be propagated further upstream. When a receiver gets this time-out notification, it knows it will not have to wait any longer for the missed packet. It will notify the user application of the fact that data was lost and continue to deliver the next packets. Whether or not an application can tolerate packet loss will depend on the type of content. Audio and video will usually not be severely impacted by the loss of an occasional packet, while real-time stock quotes may become dangerous to use when it is not known what quote was lost.

¹⁶ Additionally it is important to understand that storing copies of packets in the network can significantly increase the time a packet can survive in the network. However, it is crucial that all previously stored packets are purged by the time the sequence numbers are reused to avoid echoes of old packets to end up in new streams. For the same reason TCP uses the TIME_WAIT interval to wait for all old packets to disappear before reusing network addresses .

Selective Packet Retransmission and Out-of-Band Reception

When layered multicast is combined with localized packet retransmission, a service can be realized that applies packet recovery only to those layers that can be delivered with the current bandwidth capacity. When the network has insufficient resources to deliver all layers of a stream, it is important that the receivers know this and will not attempt to repair all missed packets from all layers. Instead, the receivers should use a mechanism that provides them feedback about the current network capacity and use that to decide for which layers it will recover missed packets. The algorithm proposed monitors the status of the total stream and from this information derives how many layers of the stream can reliable delivered to the user without stressing the network. It will then mark these layers as being *intact* or *reliable* and will send retransmission requests when an occasional packet is lost from these layers. In principle the algorithm will only deliver packets from the reliable layers to the application, but not before they have been put back into their original global order. The process of restoring the global packet order is described in the next paragraph. In order for the algorithm to decide which layers can safely be marked as intact, it requires some status information about the reception of the stream as a whole as well as the network conditions. To provide this, every packet contains the number of the most recent packet from all other layers. Depending on the type of content, it is said that the received packet is causally related to all packets from its list. This means that aside from carrying its own sequence number, each packet contains the current sequence number of all the other layers too. Now by inspecting these sequence numbers each time a packet arrives, the receiver can determine whether it has missed any packets from the layers it marked as reliable. If that is the case, these packets will have to be recovered. Each time a missed packet is detected, a countdown timer is started for it. If the packet is recovered before the time-out expired, the timer is canceled and removed. If however the timer manages to expire, the layer is considered impossible to repair and is, together with all higher reliable layers¹⁷, removed from the list of reliable layers. How long the time-

¹⁷ A higher layer is a layer that is less important than the lower layers. In terms of priorities, packet from the lower layers have a higher priority than the packets from the higher layers. When the network is congested, the routers will only forward the lower layers of each stream.

out interval should be can be application or content dependent. Essentially all it determines is how long much delay is tolerated by the user. Analogue to the time-out used by the central packet retransmission store of the router interfaces, setting it to a low value means a temporary congestion can potentially corrupt a layer long enough to cause the receivers to drop it. Note that since packets can occasionally also get lost when unicast routing tables converge that changes the shape of multicast distribution trees, or when a router crashes that had pending packets in its interface buffers, appropriate time-out values will probably have to be found experimentally. Also, it is important to realize that normal out of order delivery effectively means that one or more packets are delayed.

10 In a Terracast network that uses packet prioritization, routers explicitly introduce out of order delivery when a burst of packets queues in the outbound interceptor pipeline of a router interface with a slow TCP connection. The time-out should be large enough to allow for this. Setting the time-out to a long period makes the stream much more resilient to congestion, but also increases the time for the receiver to discover that a layer must be

15 dropped due to bandwidth constraints. Until congestion is finally detected, delivery of the previously received packets that causally depend on missed packets is postponed. Note that starting timers when packet loss is detected, allows for congestion detection. However, it cannot be used to guarantee low end-to-end latency. When no packets are lost and all are received according to their global order, the receiver cannot measure the

20 total transmission delay. As such, a receiver cannot distinguish between an idle and a crashed source. Of course this applies to most best-effort protocols. Aside from removing layers from the reliable-list, the algorithm must also be able to detect when a more bandwidth becomes available and new higher layers can be added to this list, so that a higher quality stream can be delivered to the user. This is done by passively monitoring

25 the layers that are not currently in the reliable-list and not under repair. Every time a packet is received from these layers, this fact is stored for a period that is equal to the repair time-out discussed earlier. When a packet from a layer is received who's sequence numbers show that we have missed a packet from a higher layer that is not in the reliable-list, no retransmission request is sent, but the fact that we should have received this

30 packet is recorded and stored for the same repair time-out period. If during the time-out period the packet is received after all, possibly because it was delayed by a router, the

stored record is marked as received. Because the state of each packet from every layer is recorded, Terracast builds a packet arrival history that is used to determine whether the reception quality was high enough to add a higher layer to the reliable-list and repair any further packet loss. In Terracast the reception history is used to calculate a moving

5 average that describes the amount of packet loss over the last x seconds, where x is equal to the repair timeout. With a small moving average history, the receiver will quickly respond to increased network capacity, while a longer history will only add layers when the network's capacity was sufficiently high for a longer period of time. The receiver keeps track of the reception quality of each layer by inspecting the list of sequence

10 number that is attached to each data packet. Unfortunately this causality information adds overhead to each packet, linearly related to the number of layers used in the stream. When sequence numbers are 32 bits and the publisher uses all 256 available layers, each packet comes with a kilobyte of causality information, which contributes to at least

15 12.5% for an 8 kb packet that is filled to the brim¹⁸. To ease layer administration in the receiver socket, a Terracast multicast stream only supports a static number of layers. When a publisher binds a multicast group address for reliable, layered communication, it explicitly specifies the number of layers that will be used.

Restoring Global, Local, or No Packet Order

20 By default, packets are delivered to the user application in their original order. Not only are the packets inside the individual layers restored to their natural order using their sequence numbers, but the total ordering across the layers is also restored. When the source sent three packets with different priority (each in a different layer), they are all three received by the user application in the exact same order. A prioritized, layered

¹⁸ In Terracast, data packets are restricted to at most 8 kilobyte in size. While strict maximum transfer units or MTU's are common to communication protocols that run directly on the network's physical layer – for example when the IP protocol is used on an Ethernet, the IP packets cannot exceed 1500 bytes – overlay networks do not have strict packet size limitations. In principle Terracast could allow packets of arbitrary size, however because Terracast is a store-and-forward network, smaller packets experience less hop to hop delay than large packets. Terracast's 8 kilobyte packet size limit is to keep an upper bound to this effect.

packet is described as $P_{1(9,4,6)}$ where 1 represents the packet's priority or layer, 9 the sequence number of the last packet of layer 0 (the lowest, most important layer) that was sent at the time this packet was published, 4 represents the sequence the sequence number of this packet and 6 represents the sequence number of the last packet of layer 2. Also, the packet tells that the stream uses 3 layers in total: layer 0 up to and including layer 2. Hence, it can be concluded that this packet causally depends on packet 9 from layer 0 and packet 6 from layer 2 and will only be delivered to the user application after those packets have been delivered.

a) $\{0(9,2,6); 1(9,3,6); 2(9,3,7); 1(9,4,7); 2(9,4,8); 0(10,4,8); 2(10,4,9); 1(10,5,9)\}$

b) $\{0(9,2,6); 0(10,4,8); 2(9,3,7); 1(9,3,6); 1(10,5,9); 2(10,4,9)\}$

c)

0(9,2,6)	0(10,4,8)	
1(9,3,6)		1(10,5,9)
2(9,3,7)		2(10,4,9)

d)

0(9,2,6)	0(10,4,8)	
1(9,3,6)	1(9,4,7)	1(10,5,9)
2(9,3,7)		2(10,4,9)

e) $\{0(9,2,6); 1(9,3,6); 1(9,4,7); 0(10,4,8); 1(10,5,9)\}$

Illustration 18: The process of selective recovery and reordering. a) A packet stream as published by a source. b) The packet stream as received by a subscriber. c) Packets as they are stored in the subscriber's receive buffers. d) Packets after selective repair. e) Reordered, partial packet stream as delivered to the application

Illustration 18 shows the process of selective packet repair and ordering. We use the notation $\{0(9,2,6); 1(9,3,6), \dots\}$ to describe a sequence of packets where $P_{0(9,2,6)}$ was sent prior to $P_{1(9,3,6)}$, etc. The first line in the illustration, labeled a), shows the sequence of packets originally published by the source. It reads from left to right, so $P_{0(9,2,6)}$ was published first, followed by $P_{1(9,3,6)}$, etc. In this example the source has published 8 packets, divided over 3 priority levels or layers (0, 1 and 2, where 0 is the lowest layer with the highest priority). The second line b) shows the packet stream as received

by one of the subscribers. It shows 2 lost packets and an incorrect ordering. Before the packets are delivered to the user application, they are stored in internal buffers during the repair and reordering process. This state is depicted in 18c. Here the packets are in separate buffers, each representing a layer. The packets are ordered inside the buffers.

- 5 The illustration shows the missing packets $P_{1(9,4,7)}$ and $P_{2(9,4,8)}$. If the receiver currently only has layers 0 and 1 in its reliable-list, it will attempt to repair the hole in layer 1 by sending a retransmission request. Note that if the given sequence was just a snapshot of a running stream, the receiver would have detected the missing packet $P_{1(9,4,7)}$ when $P_{0(10,4,8)}$ was received, because this packet says it depends on packet #4 from layer 1. So even before
- 10 $P_{1(10,5,9)}$ from layer 1 was received in our example, the receiver already detected loss in layer 1 and immediately scheduled a repair timeout for the missing packet and sent a retransmission request. In fact, if it is assumed that the receiver had received packet #2 from layer 1 prior to our snapshot of illustration 18, then the conclusion would be that packet #4 as well as packet #3 were lost. Shortly after $P_{0(10,4,8)}$ was received, packet $P_{1(9,3,6)}$
- 15 is received. The receiver places the delayed packet in the appropriate receive buffer and cancels the repair time-out it started earlier when it detected that the packet was missing. This is illustrated in 18d. Note that the hole in layer 2 is also detected when $P_{0(10,4,8)}$ is received, as that packet claims to be sent after packet #8 of layer 2 was published, so either packet #8 from layer 2 got lost in the network, or was delayed. However, since
- 20 layer 2 is not in the reliable-list, a retransmission request is not sent. However, as for all packets, a timer is started for packet #8 of layer 2. When the user application is ready to read packets, the algorithm will return only packets from layers that are in the reliable-list. Even though some of the layer 2 packets were received, they are discarded and not delivered. The resulting stream of packets that is delivered to the user is equal to the
- 25 stream originally published by the source, with all layer 2 packets removed. Despite the fact that the network has insufficient capacity, having dropped packets from every layer and delivered the packets out of band, the user received a deterministic subset of the stream that is uncorrupted. In most network protocols that restore ordering and guarantee packet delivery, packets are stored in a receive buffer that has a maximum size. In TCP
- 30 for example, the receiver no longer accepts packets if its receive buffer is full. For TCP this is not a problem, as the source is notified of this event and stops sending new

packets. The receive buffer can become full either when a packet retransmission takes long, while the set of pending packets build up, or when the user application simply does not pick up the packets by reading the socket. In Terracast, the first problem does not really exist. Since Terracast sources publish live data streams that cannot slow down or pause and because real-time data should not be buffered by the source too long, receivers will only attempt to recover lost packets for a limited period of time. Whether a packet is received, recovered or lost, its buffer slot will be freed after this timeout, resulting in a natural size limit of the receive buffer. How big the buffer can get is related to the average speed of the stream and the length of the repair time-out, so in principle the algorithm implementation should not enforce a hard size limit during packet repairs. A more troublesome situation occurs when the user application does not read data from the socket fast enough. When this happens, the amount of pending data builds up in the socket and both storage requirements and the transmission delay increases. A safe way of handling this could be to remove a layer from the reliable-list, causing those packets to be discarded from the buffers, while decreasing the amount of data that is delivered to the application. If a clean feedback algorithm that can keep the number of layers in balance with the application's reading speed is not used, the Terracast implementation throws a fatal exception to the user application and closes the socket when the receive buffer reaches a certain maximum size, or when the total time between arrival of packets and their actual delivery reaches a threshold. Although restoring the original global packet ordering before delivering the data to the user application is assumed to be appropriate for most types of data, there is content for which each packet invalidates all previous packets. This is the case for stock quotes. When a new stock quote update is received for a financial ticker symbol, it renders the previous updates useless. For most applications that process real-time financial data, only the most recent information is interesting¹⁹. When global ordering is restored, the algorithm will postpone the delivery of the most recent data until all prior packets have been received also. For a typical application, such as a market data terminal, that merely displays quote updates to the screen, this

¹⁹ In principle this applies to all applications that can also tolerate a thinned data stream. Some applications however, such as those computing volume weighted average pricing (VWAP), specifically require all quotes. As these applications do not tolerate delay, nor thinning, they are the most demanding and in principle cannot be supported by Terracast.

postponing adds little value. When the burst of pending updates is finally delivered by the tsocket, the application will update the symbol's last value on the screen, thereby leaving only the last and most recent update visible and overwriting all pending updates immediately. Because the reordering process adds additional delay to the data delivery, it

5 can be switched off by applications that do not benefit from it. Without reordering, global (causal relations between packets from different layers) and local ordering (order of individual packets inside a single layer) is ignored and packets are delivered to the application immediately after they have been received. Disabling reordering has no impact on the reliability. Lost or delayed packets are still recovered for all layers the

10 reliable-list, only the recovered packets will be delivered with additional delay . Whether or not this makes them useless is up to the application to decide. For market data it is important to know whether an update is older or newer than the one previously received.

15 This is because a stock quote may be overwritten by a newer one, but not the other way around. In this case, the source could add a (logical) time stamp to each stock quote, so the receiver can decide how to handle the update²⁰. If the receiver does not require the network to restore the packet order, it is interesting to see if we can take advantage of that by omitting the expensive sequence vector from the packets. Of course a packet still

20 needs to carry at least a sequence number that represents its location inside its layer to allow packet loss detection. Unfortunately this introduces another problem. Without sequence vectors, packets only maintain state information of their own layer. When suddenly no new packets are received for a certain layer, the receiver is unable to tell whether that layer is simply idle, or whether all its packets are getting lost for some

25 reason. As a result, the receiver cannot decide whether the layer should be dropped because it is missing packets, or kept because there is nothing wrong. Hence, it is concluded that the sequence vectors are required, regardless of the receiver's need for in

²⁰ Alternatively, the receiver could manually inspect the sequence vector of the received packets. However, as the sequence vectors are generated by a lower layer in the protocol stack, relying on them in a higher layer is generally not recommended.

order delivery. Aside from configuring a layered multicast tsocket to restore global ordering or no ordering at all, a receiver can configure a tsocket to only restore local ordering. Whether or not this is useful will depend on the type of application and the content, but it offers the advantage that delivery of packets from lower (high priority) layers is not delayed during recovery of packets from higher, less important layers.

Layering Market Data

The previous paragraphs have explained the use of layered multicast for live data streams in networks of varying quality. The downside of this technique however is that the content has to be pre-processed and offered to the network in packets with layer numbers. This is considerably less user friendly than a TCP connection that will reliably transfer any raw byte stream. Essentially this means that without layering algorithms for various types of live content, the Terracast network or any other communication mechanism based on layered multicast will not be able to offer guaranteed delivery of partial streams. Because of their importance to layered multicast and because Terracast specifically targets market data, rather than multimedia content, in some embodiments, this paragraph will discuss an algorithm that is used to divide a stream of real-time stock quotes over one or more layers in a way that when a subscriber receives only a subset of the layers, it still will get a consistent and up to date view of the market without the risk of stale symbols for which the most recent update was missed. Before the algorithm is explained, a leaky bucket implementation is described that can trim an incoming stream of market data down to a fixed bandwidth outgoing stream. This algorithm is then extended to produce several, incremental layers that together contain all incoming data. An important criterion in deciding whether or not a certain quote update is important enough to be forwarded by the leaky bucket, or to discard it in favour of another update is related to the volatility of the symbol. When bandwidth is lacking to forward all quote updates, the updates for the most volatile stocks should be dropped in favour of the less volatile ones. The leaky bucket's core works by arranging the individual symbols in a circular, linked list. Every item set in the list represents the most recent update for its symbol.

Every time a new update is received, it is stored in the appropriate slot, overwriting the previous update. This way, the linked list always contains the most recent trade for every symbol. Next, a virtual token traverses the list at a fixed speed. If the stream is to be thinned to 1 update per second, the token will visit 1 slot every second. When the token
5 visits a slot, it removes the quote update and forwards it, leaving an empty slot. Empty slots will be skipped by the token without any delay. When one of the symbols has 2 updates per second in the incoming stream, the second incoming update after the last visit of the token will overwrite the update pending in the slot and the older pending update is dropped. This allows the leaky bucket to always provide the thinned clients with the most
10 recent, live update of every symbol. The algorithm is illustrated in figure Xa. While the virtual token visits the slots that have a recent quote update pending to be sent out, the left arrow represents a thread that receives the incoming market data stream and inserts new trades in the slots. To limit the outgoing transmission rate to 1 quote per second, the token thread (represented by the right arrow) sleeps for 1 second after it removed and
15 sent an update from a slot. When the token is able to complete one full circle between two incoming updates of every symbol, there will be no data loss. Instead of limiting the outgoing bandwidth to a fixed maximum, the dequeuing thread could also be configured to run as fast as possible, relying on external feedback such as a blocking write call from the network, to reduce publishing speed. This could be particular useful when the
20 algorithm is used on the server side of a point-to-point TCP connection to a client application. In principle the server could use the algorithm to send the data at original speed, until network saturation slows down TCP. Unfortunately the algorithm also introduces two potential problems. The first is that the original order of the updates is lost. In fact, when every symbol updates more than once while the token circles the list,
25 the token will find every slot filled with a quote update, causing the order of the output stream match the fixed order of the slots in the linked list. For quote updates from different symbols, this is usually not a problem, as they are not related to each other.

MAC PAS605217.1-*01/26/05 7:54 PM

Fair Bandwidth Allocation and Packet Prioritisation for TerraCast

Erik van Zijst

Introduction

The algorithms described in this text are used as a congestion control unit in the interfaces of the TerraCast router daemons. Handling congestion in a multicast network is different from more conventional networked applications using unicast messages with explicit feedback messages (speed-up, slow-down, ack, nack, window-size). TerraCast is designed to be used for real-time distribution of high volume, non-interactive data streams to many simultaneous receivers. When a data stream reaches a point in the network that has insufficient bandwidth to forward the whole stream, the router divides the available bandwidth equally among the individual data streams and needs to drop packets in order to prevent buffer overflows. When carefully dropping packets when necessary, the near realtime delivery of the remaining packets can be ensured. This, together with a technique that allows application developers to “rank” individual packet to tell routers when to drop which packets, is the subject of the following text.

Previous Work

In this text two problems commonly seen in networks that are used for dissemination of realtime data streams (for example audio, video and market data) in heterogeneous environments are addressed. The first problem is that of fair bandwidth division. When a network link is temporarily or permanently congested and lacks sufficient bandwidth to forward all data in real-time, the upstream router drops the packets that cannot immediately be transmitted. This is done in nearly all store and forward packet switched networks to prevent packet buffer overflows. With well-behaving transmission protocols such as TCP, this random packet loss is not a serious problem. When a link is congested, all TCP connections running over it, will automatically reduce their transmission speed, effectively removing congestion. In networks that deal with live content however that cannot reduce their transmission speed, and in particular networks that multicast a live stream to a large number of concurrent receivers, congestion cannot be dealt with by sending less data. If data was still dropped “randomly” by routers, that would mean each stream loses the same relative amount of packets. In absolute term that means that the more packets a stream transmits, the more bandwidth it can use. In the preferred system, during congestion, the total available bandwidth must be divided equally among all data streams, regardless of their data rate. For example, when two streams traverse a congested link, both are allowed to use 50% of the link's capacity. If one of those streams requires only 30% of the capacity, none of its packets will be dropped.

The remaining 20% if its share are available to the second stream. If that stream requires the total 100% of the available link bandwidth, it will loose 30% of its packets, leading to a 30/70 percent bandwidth division where only the second stream loses packets.

Another problem of fixed rate live streams is the inherent loss of packets when receivers have insufficient bandwidth capacity. A solution for this problem has been suggested in

the literature in the form of layered multicast. In a layered multicast session, a data stream's packets are divided over a number of layers, each of which can be received and addressed individually by receivers. A receiver then subscribes to as many layers as the network can support. Essentially these layered multicast protocols can be classified into two categories. Feedback-based and feedback-free, depending on whether or not the protocol delivers feedback to the source of the multicast stream. With feedback, the sender uses the explicit receiver feedback to better align the rates of the individual layers with the receiver capabilities.

Without feedback the source transmits data at predefined rates in a pre-defined number of layers.

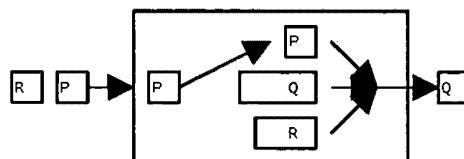
Whether or not these layers match properly with receiver capabilities depends on network characteristics. Examples of feedback-based protocols include SAMM (Source Adaptive Multilayered Multicast) and SIM (Multicast Congestion Control with Selective Participation, Intra-group Transmission Adjustment and Menu Adaptation).

Examples of feedback-free protocols include FLID (Fair Layered Increase/Decrease), RLM (Receiver- Driven Layered Multicast) and RLC (Receiver- Driven Layered Congestion Control). In this text a feedback-free layered multicast algorithm for use on the TerraCast network is proposed. It differs from most other layered schemes in that it does not encode the layers in separate data streams, but uses the TerraCast packet prioritisation to multiplex all layers in a single data stream. This has the additional advantage that the network multicast routing algorithm needs to maintain significantly less subscription information¹.

Stream Resource Allocation Algorithm

In our congestion control unit, incoming messages are sorted by unicast sender address. Note that a sender address is the combination of source router address and application session. An example of a source address is `nl.mxs.office.erik:video.an2` which could be used by a user application broadcasting a video channel. Note that only the source address is relevant, not the (uni- or multicast) destination address.

Each incoming message is added to the queue that holds messages sent by that particular sender. If this is the first message from a sender, a new queue is automatically created to store it. While these queues grow in size, a background thread constantly dequeues message from the queues and transmits them over the network. This is illustrated below.



¹ Multiplexing all layers in a single stream does however come with the disadvantage that some of the low priority layers are inevitably sent needlessly down some branches of the multicast tree, only to be discarded further downstream.

The algorithm that manages in- and output for the queues is divided in two parts. The first part is run by the background thread that constantly de-queues packets from the queues and transmits them over the network, while the second part is in charge of queuing new packets that are to be transmitted. The latter also implements the logic that defines when and which packet should be discarded (due to buffer overflow). The dequeuing part is described first.

Dequeuing Packets for Transmission

When selecting a packet for transmission, the algorithm only looks at the first packet (most urgent) of each queue. To make sure each stream gets an equal share of bandwidth, it takes the individual packet sizes into account.

When all packets (note that only the first packet of each queue is observed) have equal size, the dequeuing thread simply selects a random queue with equal probability and dequeues one of its packets.

Since each queue has the same probability of being selected, each queue will deliver an equal amount of packets per time unit. Hence, each source will transmit an equal amount of bytes per second.

Since individual packets can have any size between 1 and MTU bytes (currently the MTU is set to 8192 bytes), their size must be considered when the transmit thread selects queues. Queues with many large packets should have a smaller probability of getting selected than queues with lots of small packets in order to keep the bandwidth division fair.

In the present algorithm, only the first packet of each queue is considered to represent its queue and give it a probability of being selected that is reversely proportional to its size in bytes.

In the above example assume the first packet in P is 100 bytes, the first packet from sender Q is 300 bytes, while the packet from R is 500 bytes. To calculate the selection probabilities, we first define their respective selection weights. The weight of the packet in P is computed by dividing the total size of P, Q and R by P's size: $(100 + 300 + 500) / 100 = 9$.

Q's weight is 3 and R's weight is 9/5. We convert the weights into selection probabilities by dividing them by the sum of all weights. This gives P a probability of $9 / (9 + 3 + 1.8) = 0.65$, Q a probability of 0.22 and R of 0.13.

When a queue becomes empty after dequeuing, it is removed. More formally, the weight of queue m is defined in the formula below:

$$w_m = \frac{\sum_{i=0}^n s_i}{s_m}$$

where s represents the size of the first (“most urgent”) packet and n the total number of queues. The formula captures the inverse relation between packet size and selection weight. We derive the probability of queue m being chosen to transmit one packet directly from the absolute weights:

$$P_m = \frac{w_m}{\sum_{i=0}^n w_i}$$

we can easily substitute the above two equations into one that directly computes the queue selection probabilities:

$$P_m = \frac{\sum_{i=0}^n s_i}{\sum_{j=0}^n \left(\frac{\sum_{h=0}^n s_h}{s_j} \right)}$$

Packet Queuing and Removal

To keep the delay introduced by buffering packets under control, the component enforces a maximum total queue size. The sum of all packets in all queues may never exceed this threshold.

When a new packet comes in, it is always accepted at first. If necessary a new queue is created for it, or it is added to its designated, existing queue. After adding a new packet, the total size of all queues is checked.

If it is larger than the configured maximum, the algorithm runs a removal round in which it first selects a queue and then tells that queue to remove one packet. If the queues are still too big after removal of one packet, the process of selecting a queue and removing a packet is repeated iteratively until the total queue size is smaller than or equal to the configured maximum.

Shrinking the queues is always done after a packet was added, never pre-emptively. The reason for this is to let the new packet immediately participate in the removal selection process, rather than discarding it or making room for it at the expense of the other queues.

Contrary to de-queuing packets, where the algorithm tries to select the “most urgent” packet, we now need to select the “least urgent” one. This packet is determined by looking at the size of the individual queues (the largest queue should generally be shrunk to keep resource division among the streams fair) and which queue can match the required size most accurately. This comes from the fact that individual packets can differ greatly in size, so removing the “least urgent” packet from queue P could result in freeing 8 kilobytes, while removing the “least urgent” packet from queue Q yields 40

bytes of space for example. Now if the queues in total exceed the maximum size by only a couple of bytes, it seems logical to remove the small packet from Q. This policy comes with a consequence, namely that it implicitly promotes the use of larger packets. After all, streams with lots of very small packets are more likely to accurately match the amount of buffer space that must be freed than stream queues with few very large packets. This property may encourage developers to use larger packets, increasing the efficiency and throughput of the network.

In short, when selecting a queue for shrinking, the algorithm favors queues that are will expunge the least amount of bytes in order to match the total buffer capacity and are large compared to the others.

Before the selection can be made, the absolute weight factor for each queue is computed. The weight factor combines the queue's size and ability to accurately match the overcapacity of the buffer by removing packets. The latter is exposed in v . It represents the sum of the sizes of the packets that a queue must remove in order to eliminate the buffer's overcapacity. For example, if a queue P has 10 packets of 100 bytes each while the buffer currently has an overcapacity of 170 bytes (suppose the maximum is 10000 bytes, while all queues combined add up to 10170 bytes), P would need to remove at least 2 packets ($2 * 100$ bytes). In this case, v_p is 200.

The formula to compute the absolute weight factor of queue m is defined below.

$$w_m = \frac{\left(\sum_{i=0}^n v_i \right) \cdot s_m}{v_m}$$

In this formula, n represents the total number of queues, v represents the number of bytes that would be removed if the queue was selected and s represents the total size of a particular queue. The formula captures the direct relation to queue size and the inverse relation to the amount of bytes that would be removed by the queue if it was selected.

When the absolute weight factors of all queues have been derived, we can convert them to weighted selection probabilities. This is easily done by dividing the individual weight values by the sum of all weights. The selection probability of queue m is expressed in P_m .

$$P_m = \frac{w_m}{\sum_{i=0}^n w_i}$$

For the sake of completeness, we can substitute w_m by the definition of the absolute weight. This leads to the formula below that directly computes the selection probability P for queue m .

$$P_m = \frac{\left(\sum_{i=0}^n v_i \right) \cdot S_m}{v_m \left(\sum_{h=0}^n \frac{\left(\sum_{j=0}^n v_j \right) \cdot S_h}{v_h} \right)}$$

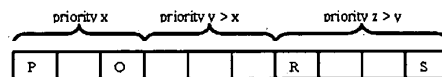
After the selection probabilities have been computed, a roulette-wheel algorithm² is run to select a queue and let that queue remove its v bytes. If v is smaller than the current overcapacity of the buffer in total (note that this implies that the selected queue became empty and was thus removed automatically), another selection round is done until the buffers shrinked sufficiently. Note that the queue selection process is implemented as an atomic operation. During selection rounds no new packets may be added to or removed from the queues.

Packet Prioritization

So far, two algorithms have been discussed that together attempt to equally divide the available bandwidth between streams, where streams are defined as the collection of packets originating from the same source. However, to give a source application more control over what is received by the clients when a certain percentage of the data is discarded by intermediate routers, relative packet priority levels are introduced. This allows a source to tag each outgoing packet with a priority value from a fixed range. These values are only used within the stream and have no influence on the bandwidth allocation between different streams. This means that a stream where every packet has a high priority value has no precedence over streams carrying only low priority packets.

The packet prioritization is enforced independently inside the packet queues. Although many different techniques are possible, the pending packets in a queue are sorted by priority and time of arrival. This leads to a datastructure as illustrated below.

Four packets that occupy interesting slots are labeled.



When the dequeuing thread has selected a queue, it dequeues the packet whose size was used during the selection probability computation. By default, this is the packet in slot S. This is generally the oldest packet with the highest priority. When a new packet queued, it is appended at the most left slot of the subqueue that represents its priority. For example, a new packet with priority z is placed in slot R. The packet currently occupying that slot (together with all other packets to the right of slot R) are shifted one position to

² The roulette-wheel selection algorithm divides an imaginary wheel in slices (each representing one selection option) with sizes proportional to their probability. The wheel is then virtually spun once to make a selection.

the right. When a packet is queued with a new priority level, a sub-queue is immediately created and inserted so that all priorities are sorted incrementally.

While the individual sub-queues have no hard bound on their size, the queue as a whole does have a maximum size. Together with the other streamqueues, it may not be larger than the unit's maximum buffer size. If this ever happens, the packet queuing and removal algorithm described in the previous paragraph will select one of the stream-queues and instruct it to remove enough packets to satisfy the unit's maximum buffer size. The default queue implementation will always remove packets left-to-right.

The oldest packet with the lowest priority is always dropped. If the last packet from a sub-queue is removed, it is automatically removed.

When high priority packets come in at the same rate at which they are dequeued from the queue, it is possible for lower priority packets to get stuck in the queue for longer periods of time. This has at least two negative consequences. Since there is no guaranteed delivery, a client should proceed reading even if a certain packet was not received. However, it could be that the packet was not dropped, but merely significantly delayed by router interface queuing. Because a client never knows whether or not a packet was simply dropped and lost, or merely delayed, it cannot wait for it very long. Also, when the client proceeded stream processing, the delayed packet is often worthless and receiving it after all is a waste of time. Additionally these, often worthless, delayed packets occupy queue space that negatively influences the capacity of the other stream queues within a router. Although this problem is currently not anticipated, it may be useful, depending on the requirements of the client application, to discard a packet when another packet was forwarded whose sequence number is more than x units higher.

This threshold could be specified per packet.

FRS PAS605210.1-* -01/26/05 8:34 PM

Computing Sparse Distribution Trees for Extended Bellman-Ford

Erik van Zijst

Abstract

The algorithm proposed in this text computes optimal, sparse sink trees for delivering multicast content from one source to multiple receivers in a distributed way. To this end it utilizes the routing information that can be provided by most conventional routing algorithms that compute shortest paths. In this text it is assumed that the underlying routing protocol is ExBF.

Tree Building

Initially, a source q is active but since no receiver is subscribed while this is a sparse mode protocol, no distribution tree exists. When the first receiver application p_0 subscribes, its router initiates the reverse computation of one path of the sink tree routed at source q . It does so by first marking that p_0 has a local subscriber for (q, G) by setting the boolean in $LS_{p_0}[q, S]$ ($LS_p[]$ is a local array at p that keeps track of all subscriptions of the local user application running on top of p) to true and then sending a **<join** (q, G) message to the neighbor that has the shortest path towards the source. This neighbor p_1 is known as the preferred hop of node p_0 for destination q and it is assumed that this information is available from the underlying routing algorithm that computes shortest paths.

Process p_1 receives the **<join** (q, G) message and marks the adjacent link with p_0 as a child link for group (q, G) . It then forwards the **<join** (q, G) to its next hop (p_2) in the path towards q . Eventually node p_n sends the **<join** (q, G) message to q . On receipt, q marks the link on which the message was received as a child link for local group G and starts forwarding all messages addressed to G over the link to node p_n . The messages addressed to G are inserted into the network at node q by a higher level user application that runs on top of the path from p_i to q that receives the **<join** (q, G) message and suppose p_{i+n} is also in the path between p_0 and q . In this case, p_{i+n} is already in the established part of the sink tree of q , so it does not need to forward the **<join** (q, G) message further towards q . It only marks the link on which it received the message as a child link for (q, G) . In general: a node u only forwards a **<join** (S, G) message towards S if it is not already subscribed to (S, G) (hence, $LS_u[S, G]$ is not set and the collection of child links for (S, G) is zero). This is shown in illustration 1.

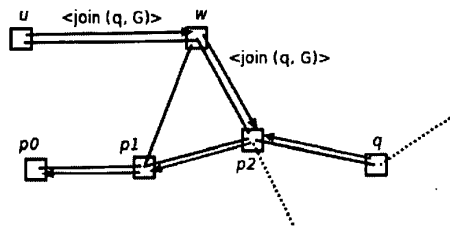


Illustration 1: When u joins the multicast tree, a $\langle \text{join } \dots \rangle$ message is forwarded towards q until it reaches a node already in the tree

Adaptation

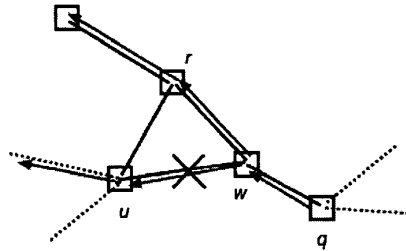


Illustration 2: When link $[u, w]$ fails, the tree becomes partitioned. The disconnected fragment then joins the connected tree at a different node.

The sparse mode distribution tree generated when processes subscribe to a (S, G) group originally equals a part (or all) of the optimal sink tree rooted at S . However, when the underlying routing algorithm detects changes in the network performance and updates (some of) the routing tables, the optimal sink tree changes and is no longer matched by the multicast distribution tree and multicast performance becomes sub-optimal. An extreme example is when link uw fails, while node u and w are both in the distribution tree for (q, G) and w was the parent of u in the sink tree rooted at q . This is shown in illustration 2. At this point the multicast distribution tree becomes partitioned. When the underlying routing algorithm detects the failure (it is assumed that it is notified on link failure), it updates the routing table and selects the adjacent link to r with the next-best distance to q to become the preferred hop towards q . To reflect the change in the optimal sink tree, node u invalidates its subscription at w by sending a $\langle \text{leave } (q, G) \rangle$ message (in case of link failure between u and w , this step can be omitted). It then sends a $\langle \text{join } (q, G) \rangle$ message to the new preferred hop r . As an enhancement to minimize message loss, node u would normally send the $\langle \text{join } \dots \rangle$ message first and optionally even wait for the first message addressed to (q, G) to arrive before sending the $\langle \text{leave } \dots \rangle$ message. Aside from message loss, this step can also lead to message duplication. This is accepted and left either

to the user application or a higher layer in the product to handle.

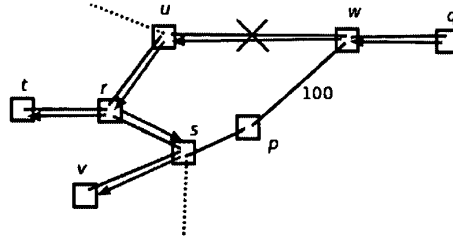
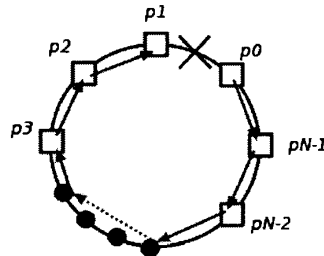


Illustration 3: In some cases ExBF causes nodes to become disconnected for a finite period of time after a link failure, causing the recomputation of the distribution tree to become more complex.

When a node u currently in the distribution tree for (q, G) only has a single route to q via w , while link uw fails, u concludes it can no longer be part of the distribution tree (u and q are no longer connected), and invalidated all child link subscription information. In case $LS_u[q, G]$ is set, u will re-join the distribution tree as soon as a path to q is found. Optionally the algorithm may send a notification message to the subscribed application to indicate that u is no longer connected to q .



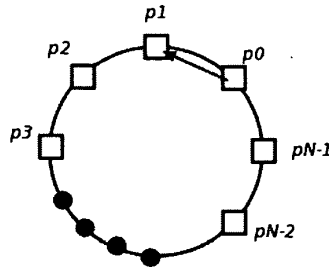
weight(p_0, p_{N-1}) $\geq N$
weight(p_i, p_{i+1}) = 1 where $0 \leq i < N-2$

Illustration 4: When ExBF causes nodes to become temporarily disconnected, recomputation of the distribution tree can take up to $3N$ time units. This is described in illustration 4 and 5.

This situation is illustrated below. All links are assumed to have weight 1, while link pw has weight 100. The result is that link pw will not be in the optimal sink tree rooted as q . Since only t and v have a local subscriber ($LS_t[q, G]$ and $LS_v[q, G]$ are set), the sparse mode distribution tree equals the red tree. If we assume that the underlying routing protocol is ExBF, nodes w , u , r and

s all have only a single route to q . Node p has a path through both s and w , while s is the preferred hop towards q . When link uw fails, u becomes disconnected from q and invalidates the subscription information for link ur . Node u does explicitly inform neighbor r of this fact. Since ExBF exchanges routing data with its neighbors, and links in the present model are FIFO, this notification would always be preceded by the routing update sent by ExBF in u , telling r that u can no longer reach q . Since the routing update implicitly tells r that its join for (q, G) through u is no longer valid and that it should look for an alternative neighbor to re-join, an explicit failure or subscription-cancellation message is unnecessary. Since r is also left with no path to q and has no locally subscribed application, it invalidates its (q, G) subscription information and is no longer part of the distribution tree. Neighbors t and s are implicitly informed about this through the ExBF routing update sent by r . Eventually w , u , r and s all leave the distribution tree, while nodes t and v schedule a re-join when a new path to q is found. Again, they may inform their local subscriber about this. When p receives the routing update from s , it loses its preferred hop for q and switches to neighbor w . The new cost to q is now $100 + d(w, q)$ and neighbor s is informed. Upon receipt, s discovers the new path to q and informs its neighbors r and v . Node v now sends the $\langle \text{join}(q, G) \rangle$ message to s and s , p and w construct the first part of the new, optimal multicast distribution tree rooted at q . When the new path reaches r and t , t send a $\langle \text{join}(q, G) \rangle$ message to r and r to s .

Essentially, recovering from the above situation is divided in three steps. In the first step the underlying routing protocol (ExBF is assumed) starts a wave that propagates the link failure through the network. Depending on the actual topology, this can take up to $N-1$ messages. To understand this, consider the ring network of illustration 4. Assume p_1 is subscribed to group (p_0, G) . Due to the link costs, the sink tree rooted at p_0 runs anti-clockwise from p_0 to p_{n-1} . The multicast distribution tree only contains the edge between p_0 and p_1 and nodes p_1 to p_{n-2} only have a single path to p_0 via their clockwise neighbor. Node p_{n-1} has both a path to p_0 via neighbor p_{n-2} (cost $n-1$) and via neighbor p_0 (cost n). The preferred path runs through p_{n-2} . When link p_0, p_1 fails, it is propagated through the ring in anti-clockwise direction from p_1 to p_{n-1} . When p_{n-1} loses its preferred route, it switches to neighbor p_0 and tells p_{n-2} about the new route. This update is passed through the network in clockwise direction from p_{n-1} to p_1 . When p_1 learns about the new path to p_0 after $2N$ time units, it sends a $\langle \text{join}(p_0, G) \rangle$ message to p_2 . This subscription is passed through the network in $N-1$ time units and after it has been received by p_0 , the new multicast distribution tree is computed as depicted in illustration 5. It can be concluded that the worst-case time complexity for recovering from an error such as this when ExBF is used as the underlying routing protocol is $3N$. Note that when only p_{n-1} was subscribed to (p_0, G) , time complexity would be down to $2N$.



$\text{weight}(p0, pN-1) \geq N$
 $\text{weight}(pi, pi+1) = 1$ where $0 \leq i < N-2$

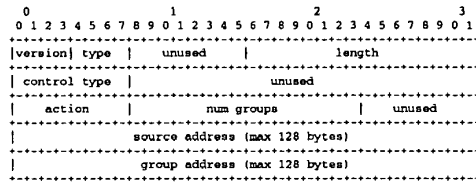
Illustration 5: Recovery from a failed link has worst-case time complexity of $3N$.

So far situations have been described in which an edge in the multicast distribution tree fails. More often however, links will not fail completely, but rather fluctuate in quality, causing the underlying routing algorithm to reassign preferred neighbors. In this case it is not a requirement that (part of) the distribution tree is recomputed, however since the tree became sub-optimal, it could be preferred. In general, the decision whether or not to recompute a distribution tree should be made according to the quality of the used path. When a node p in the tree detects that the neighbor in the optimal path towards the source has a cost that is only marginally lower than the neighbor that is currently p 's parent in the actual distribution tree, the overhead of recomputation and the risk of message loss may not weigh up to the increased performance of the new tree. Also, since the quality of the neighbor links is continuously evaluated, the shorter sink tree could only be temporal. Two straightforward solutions come to mind. Either the algorithm could set a threshold on recomputation so that subscriptions are only moved to the neighbor with the optimal path towards the source if the decrease in cost is at least a ratio x where $x > 1$. Another solution could be to postpone recomputation for a period inversely proportional the cost decrease ratio x . The latter has the advantage that the optimal distribution tree is always guaranteed to be reached in finite time since the last routing table change.

Group Membership Messages

A multicast group is S, G tuple. Both are defined to be a normal address. This means the address space of a group is equally large as a regular host address. The UserData message that is used to carry normal application data for both unicast and multicast connections doesn't need to be extended.

To run the tree building protocol, two new messages are introduced: the join message and the leave message.



Both actions are described by the new GroupData control message. It is used to send a list of subscriptions to a neighbor node or to cancel an aggregated list of subscriptions with a neighbor. The action field (9th byte) is used to distinguish between join and leave. The 10th and 11th byte indicate the number of (S, G) groups in the message. The action field is either "join" (0) or "leave" (1). In the future this may be extended to "stale", "dead", etc, to indicate the state of the publishing application.

MAS PAS605211.1-* -01/26/05 7:16 PM

Address Summarization for the Extended Bellman-Ford Protocol

Erik van Zijst

Abstract

This text describes the application of address summarization to the Extended Bellman-Ford distance vector protocol, which is a descendant of the well known distributed Bellman-Ford routing protocol (implemented initially in ARPANET). This paper is based on earlier work and describes the technique we have chosen for our experiments in greater technical detail and specifically how to deal with the head-of-path information that ExBF attaches to each individual path and is used to eliminate the bouncing effect and the counting-to-infinity problem. Additionally, this paper presents a wire-level optimization algorithm that could be used between peers to decrease the size of the packet headers significantly by substituting the dotted address strings.

Introduction

"Address Summarization for Distance Vector Protocols" describes two hierarchical addressing schemes that allow ranges of hosts, subnets or groups of subnets to be substituted by a single wildcard address in a node's distance table or routing table. This is an essential enhancement for traditional distance vector protocols when the network is very large in size or has frequent topology changes. One of the proposed schemes uses variable length dotted strings to represent both unicast and multicast addresses, while the second is a more traditional numeric scheme with a 64 bit address space and fixed domain classes. It is a simple and somewhat limited scheme as it does not support variable subnet lengths as used in Classless Inter-Domain Routing (CIDR) for IP. For current lab experiments I have chosen to use the more flexible, easy to understand and rather unconventional address strings instead of the more efficient but complex numeric scheme. The major drawback of using variable length dotted strings is that they require more bytes to serialize than a 32 or 64 bit numeric address. If we consider the string `com.gblx.level3.mxs.gw3` to be a typical unicast host address, it requires at least 23 bytes or 184 bits to include this address in a packet header. Note that most packets will carry at least two addresses, namely a source and a destination address. Also, with the current maximum address length of 127 bytes, that could contribute to 254 bytes of address headers, opposed to 16 bytes per packet when two 64 bit numeric addresses are used. This makes the protocol very bandwidth inefficient when the payload per packet is small. Also, routers will probably require more computing power and storage capacity to process strings instead of doing numeric operations. Nevertheless, we think modern computers are cheap and fast enough to allow for the added computational complexity but instead present network administrators with a more flexible and easy to use addressing scheme.

Head-of-Path Summarization

The Extended Bellman-Ford protocol, that is at the base of the experiments, stores the last hop to each destination as the head of the path to that destination. This head address is included in every distance vector and in each entry in a node's routing table for each neighbor. With this

information it is possible to compute the complete path from every reachable destination back to the node itself by recursively using the head of path as a destination itself. This allows nodes to discover routing loops (non-simple paths) and prevent them by not advertising paths to neighbors that would lead to non-simple paths. This approach is comparable to the *split-horizon* or *poisoned-reverse*¹ technique in that they both advertise destinations as being unreachable that would otherwise lead to routing loops. The difference is that where route-poisoning can only prevent loops with only two nodes, while the head-of-path technique detects loops between any number of nodes. The problem can be shown in the network in illustration 1.

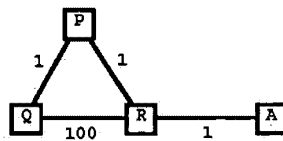


Illustration 1: Example distance vector network that uses poisoned-reverse but still suffers from a routing loop.

When run by the distributed Bellman-Ford protocol, Q will advertise its distance vectors to R. This includes the path to A (via P and R) with distance 3. R will have two routes to node A, one directly to A with distance 1 and one via Q with cost 103. Note that P will not advertise a route to A to R because of the poisoned-reverse mechanism. Now when the link between R and A fails, R's preferred hop for destination A changes from A to Q. This creates a routing loop as well as the counting-to-infinity problem. R advertises its new route to P (not to Q, due to poisoned-reverse). P only had one route to A (via R) and learns the new distance of 104, which it advertises to Q. Q updates its distance from A to 105 and advertises it to R, etc.

ExBF prevents the loop using the head-of-path information. When the routing tables initially converge, every node verifies its neighbor is not in the path it advertises to it. Before Q advertises destination A to R, it backtracks the path from A back to Q and discovers the presence of R. This would create a three-node routing loop and as a result Q does not advertise the route to R.

Applying address summarization on received ExBF distance vectors means also summarizing the head-of-path addresses.

¹Split Horizon consists of omitting routes learned from a peer when sending updates back to that peer. With Poisoned Reverse instead of omitting those routes, they are advertised as unreachable.

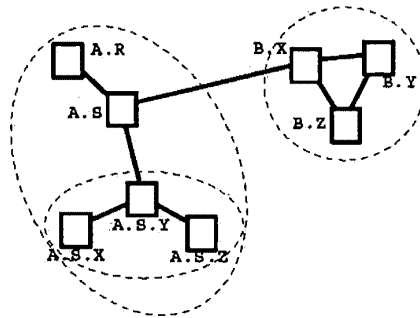


Illustration 2: Example topology to demonstrate head-of-path address summarization in ExBF. We assume all links have a cost 1.

The example network topology as depicted in illustration 2 is used to demonstrate the exchange of ExBF distance vectors and creation of summarized distance tables as well as the preemptive detection of routing loops. When the network first starts to converge, B.X advertises the following routing information to neighbor A.S: $DV_{B.X, A.S}: \{(B.X, *, 0), (B.Y, B.X, 1), (B.Z, B.X, 1)\}$ ². When A.S receives the vectors it summarizes the entries. What remains is the single (shortest) vector $DV_{B.X, A.S}: \{(B.*, *, 0)\}$. The distance table of node A.S now contains $DT_{A.S}: \{(A.S, *, *, 0), (A.R, A.R, A.S, 1), (B.*, B.*, A.S, 1)\}$. Note that all addresses are summarized before processing. This includes neighbor addresses. The consequence of this is that when a node has more than one connection with a foreign domain, both neighbor addresses will be summarized into the same wildcard. This is a known limitation of the current architecture that assigns addresses on a per node basis, rather than per interface and also occurs when configuring the same neighbor twice without summarization. The routing update A.S sends to A.S.Y contains $DV_{A.S, A.S.Y}: \{(A.S, *, 0), (A.R, A.S, 1), (B.*, A.S, 1)\}$ and leads to A.S.Y's distance table $DT_{A.S.Y}: \{(A.S.Y, *, *, 0), (A.S.X, A.S.X, A.S.Y, 1), (A.S.Z, A.S.Z, A.S.Y, 1), (A.S, A.S, A.S.Y, 1), (A.R, A.S, A.S, 2), (B.*, A.S, A.S, 2)\}$. When A.S.Y has finished updating its distance table, it advertises $DV_{A.S.Y, A.S}: \{(A.S.Y, *, 0), (A.S.X, A.S.Y, 1), (A.S.Z, A.S.Y, 1), (B.*, *, *), (A.S, *, *), (A.R, *, *)\}$ back to neighbor A.S where the * indicates unreachable in the last three vectors.

Packet-Level Address Substitution

As explained before, the use of large human readable text strings as addresses can introduce a large amount of overhead in packet headers and be wasteful for bandwidth. Favoring bandwidth efficiency over computing power efficiency, I propose a mechanism for substituting the address strings by small 16 bit values when transmitting a packet to a neighbor and keeping a translation table at both sides of the connection that is used to put the original address string back in the packet before processing it further. This optimization works on a per link basis and it totally independent from the communication with other neighbors. When it is used on more than one interface, each interface maintains its own translation table. When a packet is transmitted to a

²The following notation is used to describe a set of distance vectors exchanged by peers: $DV_{from, to}: \{(destination, head-of-path, distance), \dots\}$ When describing a distance or routing table, we use the notation: $DT: \{(destination, hop, head, distance), \dots\}$

0										1										2										3									
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
version										type										unused										length									
address 1 (max 128 bytes)																				16 bit substituted value																			
address 2 (max 128 bytes)																				16 bit substituted value																			
address n (max 128 bytes)																				16 bit substituted value																			
address m (max 128 bytes)																				16 bit substituted value																			

0										1										2										3									
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
version										type										unused										length									

Ex. 1014, p. 84 of 112

After a connection with a peer has been lost, the handshake must be done again since the peer could be replaced by software that does not support substitution. Entries in the translation table each have a time out. After this period the entry is removed from the table and any packet still using the substituted value will not be processed causing the peer to send back a packet (with type 0x2) containing the 16 bit substitution values that were not recognized. Since the invalidation of timed out entries is implicit, it is essential both peers use the same time out to avoid re-use of values by one peer that still represent old addresses in the other peer. When the translation table is full, entries are best removed according to a last recently used scheme.

Conclusions

In this text I have studied the application of summarization on a location-based addressing structure for the Extended Bellman-Ford routing protocol. The head-of-path information used for preemptive routing loop detection is maintained and the use of wildcards is demonstrated. Additionally I have proposed a simple and somewhat naïve mechanism for reducing the size of the address headers in data packets. All in all this should contribute to a scalable and relatively bandwidth efficient dynamic routing protocol with a flexible and easy to understand addressing allocation structure.

MAC PAS605195.1-* -01/26/05 7:06 PM

Address Summarization for Distance Vector Protocols

Erik van Zijst

Abstract

Adaptive routing protocols such as distance vector¹ and link-state protocols maintain routing tables that contain a routing entry to each destination in the complete network. This way each packet can be forwarded directly to the neighbor that is in the shortest path toward the destination. When something changes in the network topology or available bandwidth, this may trigger changes to certain routing entries. In link-state protocols these changes need to be propagated to all nodes in the network as every host maintains an image of the complete network topology. With distance vector protocols the situation is usually slightly better, but some changes also need to be propagated throughout the entire network. The obvious problem is the limited scalability of these routing tables. When the network grows in size, so do the routing tables. This requires more bandwidth to exchange the necessary routing data between hosts. Also, the more links there are, the more changes occur in the network. These factors severely limit the scalability of these protocols. In this text I propose a technique that can significantly increase the scalability of large networks by substituting logical ranges of hosts in the routing table by one single, condensed entry. The farther a destination host is away, the more efficient it can be condensed together with other remote hosts. This is called address summarization. Because this text is targeted to the new MarketXS multicasting network that uses a distance vector protocol at its base, the proposed technique will only be discussed in the context of such protocol, although it can also be applied to link-state protocols.

Location Based Addressing

Traditional distance vector protocols only require host addresses to be unique. This results in routing tables similar to the one below.

<i>dest</i>	<i>Hop</i>	<i>cost</i>
P	R	135
Q	R	137
A	A	12

Illustration 1: Routing table fragment of a typical distance vector protocol.

Given the addresses of the other nodes, there is no way to tell approximately where these hosts are located, although looking at both the distance (cost) together with the direction (hop or neighbor) might reveal that both P and Q are located close to each other at the far end of the network. If we consider the above routing table to be a fragment of the routing table of node C in the network depicted below, we can see that P, Q and R belong to the same, remote subnet.

¹ Note that a network will typically have a hierarchy of multiple nested domains. This is much like the real world where a street is a domain inside a city and a city is a domain inside a country, etc.

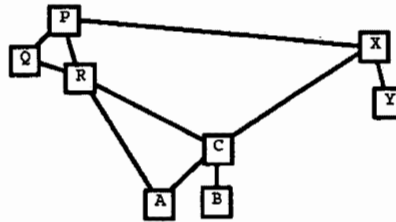
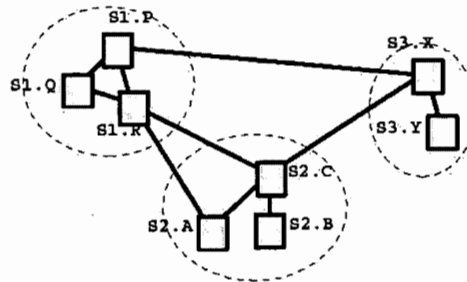


Illustration 2: Example network topology with abstract node addresses.

The paths to the individual nodes in the subnet largely overlap (for a large part of the total cost if we consider the link between C and R to be an expensive, long-distance connection). To capture the notion of subnets, is assigned a name to every subnet and prepend it to the host address. The subnets are called S1, S2 and S3 and as a result the fully qualified name of node C becomes S2.C. This makes the addresses location based.



Address Summarization

Given the logical domains that now group clusters of nearby hosts, each host can treat domains other than its own as a single entity and use a wildcard address that matches every host inside the domain. This reduces the size of the routing table, as well as the amount of routing information that needs to be exchanged between the nodes when the topology changes inside a domain.

When a new host is added to domain S1, there is no need to propagate that information to the other domains, as they already have a wildcard entry that will match the new address. In illustration 4 is applied this condensation or summarization process to a fragment of the routing table of node S2.C.

Dest	hop	cost
S1.*	S1.R	132
S3.*	S3.X	157
S2.A	S2.A	12

Illustration 4: Routing table fragment with address summarization applied.

The cost metric that is listed in the wildcard entries is the shortest path to any host inside the domain. In the present example that equals the cost to reach S1.R from S2.C. If we assume that both inter-domain links S2.C – S1.R and S2.A – S1.R have the same weight or cost, S2.C will route all traffic for domain S1 through neighbor S1.R.

When nodes exchange distance vectors, either because of a link change or as a normal, scheduled, periodic exchange, the receiving node first summarizes the destination address of each distance vector relative to its own address. It matches the destination address with its own address field by field and when a field does not match, the rest of the address is substituted by a wildcard. When S2.C receives a distance vector from S1.R that contains a path to destination S1.P, it is changed to S1.* upon arrival at node S2.C. This is because the first field differs from the first field of the local address and as such the rest of the fields are replaced by a wildcard. This wildcard entry is then fed to the distance vector core that checks whether the new cost (path length) is shorter than the cost of the entry that was already in the routing table. In the present example there already is an S1.* wildcard entry in the routing table that was derived from neighbor destination S1.R. Since the path to S1.P runs through S1.R, the path to S1.R will always be shorter than the path to S1.P, so the entry in the routing table will not be changed. In general, when the local address is 1.2.3.4 and an incoming distance vector advertises destination 1.2.2.2, it will be summarized to 1.2.2.*. Destination 2.1.6 becomes 2.*, destination 1.2.3.4.5.6 becomes 1.2.3.4.5.*, destination 1.2.3.5 stays 1.2.3.5 and destination 1.2.3.4.5 also stays 1.2.3.4.5.

Note that 1.2.3 and 1.2.3.* are two different addresses. The first only matches the exact address 1.2.3 while the second is a wildcard that matches everything that starts with 1.2.3 and has at least 4 fields. This includes 1.2.3.4 and 1.2.3.4.5.6 but not 1.2.3.

Increased Stretch Factor

Summarization means we maintain only a single path is maintained for a range of remote nodes. The consequence of this is that packets will not always be routed according to the real shortest path between source and destination. Consider the example network of illustration 3. Node S1.R receives the distance vectors of both S2.A and S2.C and after summarization learns that both neighbors offer a route for wildcard S2.*. If we assume that both inter-domain links (S1.R – S2.A and S1.R – S2.C) have equal costs, then S1.R will choose S2.A to be the preferred hop for S2.* because of the fact that its address logically comes first. When S1.R now needs to forward a packet to S2.B, it uses the S2.* wildcard entry and forwards the packet to neighbor S2.A. Unfortunately this is not the shortest path to S2.B as S2.A first has to route the packet through S2.C. Instead, S1.R should have sent the packet directly through neighbor S2.C. This ratio between the length of the actual path and the distance between the endpoints is called the stretch factor. The path between S1.R and S2.B stretches as a consequence of the address summarization

Summarization Levels

Path stretching can occur when packets are forwarded between nodes that are in different logical domains. An entire subnet is treated as a single node with several outgoing links. Because the virtual node often contains a large number of nodes, connected by its own internal network, it is sometimes better to choose a different inter-domain link when sending packets to the domain.

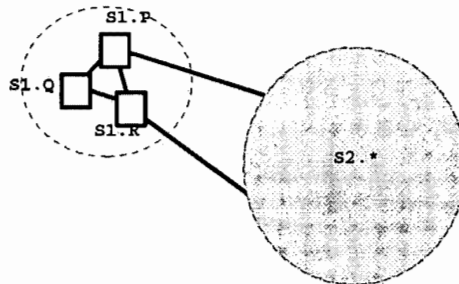


Illustration 5: With address summarization a different domain is seen as a single wildcard entity.

Consider illustration 5. When node S1.P needs to forward a packet for destination S2.F, it will send the packet directly down its inter-domain link to the S2 domain. In this example we assume both inter-domain links to have equal costs. Illustration 6 shows the inner network of the S2 domain and it becomes clear the stretch factor is quite high for packets from S1.P to S2.F, as the shortest path runs through S1.R.

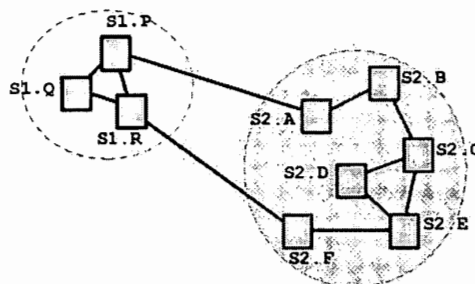


Illustration 6: With summarization level 2, nodes can look one level deep into a foreign domain. This reduces the stretch factor.

The stretch factor can be manipulated by changing the level of summarization. By default, an address is summarized after the first field that differs from the local address. However, if we make that the second field, we can look inside other domains for one level. A host with address 1.2.3.4 will then summarize 2.3.4.5 into 2.3.* and 1.3.4.5 into 1.3.4.*. Doing this at least on the border nodes that have the inter-domain links might be a good way to reduce the stretch factor. As the network was designed to be administrated by independent parties, administrators are free to experiment with different levels of summarization without jeopardizing the other subnets or domains.

Addressing Structure

In this article dotted strings are used as node addresses. The advantage of such a scheme is the large address space and flexible naming. New sub-domains can be added at any time as there is no need for a predefined or fixed number of hierarchical levels. Also, if individual address fields are not bound to a limited number of characters, they can contain geographical locations or company names. An example might be nl.amsterdam.level3.marketxs.pc3, or nl.ams.l3.mxs.pc3 to keep the address relatively short. The drawback of using this kind of unrestricted addresses is the amount of bytes they require. Because every packet always contains a source and a destination address, the amount of overhead per packet can be excessive. Early demo versions of the software could use these kind of addresses by letting the first byte of each address header specify the length of the address in bytes, followed by the address itself as a normal 8 bit ASCII string with a maximum of, say, 256 bytes. Null-terminated strings might be an alternative. For the final product it will probably be desirable to come up with a different, more lightweight way of representing hierarchical addresses. A straightforward way of assigning addresses could be to use a sufficiently large address space and divide it into logical subsets, much like IPv4 and IPv6 addresses. If we use a 64 bit address space and divide it into 8 bit fields (identical to IPv6) used to identify logical subnets, we get addresses like 120-23-61-201-43-146-128-132. To make distinct addresses, IPv4 uses the dot "." for separation, IPv6 uses the colon ":" and the present system uses a hyphen "-". When each address field represents exactly one logical subnet level, this scheme provides a simple, but at the same time somewhat limited way of addressing. It means that the complete network can address as many individual hosts as IPv6, while grouping them in at most 8 levels of sub domains, where each sub domain can contain a maximum of 256 hosts or sub domains.

The numerical addressing scheme is clarified with illustration 7.

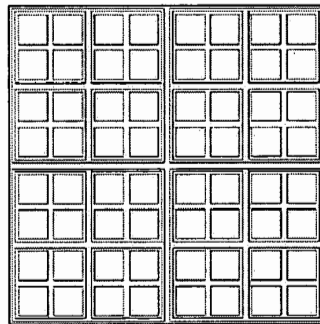


Illustration 7: Dividing addresses in logical domains and sub domains.

In this figure the whole address range has been divided it in 4 subnets or domains. Each of these domains have again been divided into 4 smaller domains, until every individual host address has its own private domain. If we do this for 4 levels deep, we can address a total of $4^4 = 256$ hosts. If normal address summarization is applied on a host's routing table, that would lead to a maximum of 13 routing entries in total if we include an entry to ourselves. This is 3 entries to identify the 3 remote subnets on the highest level, another 3 to identify the remote subnets on the next level, etc. The deepest level includes our own address, which makes up for 4 entries. The number of routing entries is thus a function of the size of each domain and the total depth of the hierarchy:

$$(ds-1) * hd + 1 = se$$

while

$$ds^{hd} = h$$

Where ds represents the number of hosts or sub domains in a domain, hd stands for the maximum depth of the domain hierarchy and se is the total number of routing entries after summarization. In the second formula, h stands for the total number of hosts. In the present example se would be $(4 - 1) * 4 + 1 = 13$.

If we increase the domain size ds to 16, while decreasing the hierarchy depth hd to 2, then the total number of hosts h still adds up to 256 ($16^2 = 256$), but the number of routing entries se increases to $(16 - 1) * 2 + 1 = 31$. If we work the other way around and decrease ds to 2, while increasing the hierarchy depth hd to 8, we can still address the same 256 hosts, but we would need only $(2 - 1) * 8 + 1 = 9$ routing entries. Obviously the stretch factor is inversely proportional to the size of the routing table, so a small se is not necessarily a good thing. With a 64 bit address space and 8 bit domains, the largest routing table a host would ever have to manage is $(256 - 1) * 8 + 1 = 2041$ which seems like a reasonable maximum. If we make the domain size 4 bit (16 hosts) and the depth 16 levels, the maximum routing table size is only $(16 - 1) * 16 + 1 = 241$ while we can still reach 264 individual hosts.

Conclusions

In this text the scalability problem of standard distance vector protocols is identified. Because each host maintains some information on every other host in the entire network, the overhead introduced to keep all routing tables in sync grows proportionally to the number of hosts in the network. This includes increased memory and processor capacity to store and search the larger routing tables and an increased number of routing updates to propagate all network changes to relevant hosts. When the network grows beyond thousands or maybe even millions of hosts, this overhead quickly grows out of proportions, making the network unusable.

I have proposed a technique called address summarization in which ranges of addresses are grouped into logical domains and sub domains, while using wildcard addresses to refer to domains instead of their individual hosts. The advantage of this scheme is the increased scalability. I have proposed two hierarchical addressing structures, a flexible one that uses human readable dotted strings but suffers from a high overhead factor and a numerical addressing scheme that is more likely to be useable in a production environment.

Depending on the configuration of the numerical scheme, one could let the distance vector protocol dynamically manage 264 hosts with only hundreds to thousands routing table entries. With appropriate sorting and search algorithms, together with modern computers, routing tables with these sizes are considered manageable. Also identified are the main drawbacks of the present approach. Increased scalability means that packets often do not travel according to their real shortest path. This deficiency is called the stretch factor. Another downside is the less flexible address allocation for new hosts and domains.

MAC PAS605213.1-01/26/05 6:43 PM

Technical Overview of the TerraCast Network

Introduction

TerraCast is a bunch of software router daemons, interconnected by normal TCP connections. Together this forms an intricate web of routers and links. Much like the Internet itself, but fundamentally different because it works on a higher level.

The Internet is a physical network, built from physical router devices and copper or fiber optic wires. Inside the router devices, advanced circuits and software figure out the structure of the network and ensure you can send individual messages to any other physical host in the network. The routers, being the junctions between multiple links, know where to send each message, so that all data eventually arrives at its destination. This gives every connected host the ability to receive data, as well as distribute it.

TerraCast effectively offers the same services, but in the form of a "virtual" network. At the heart are the software routers that, analogue to those physical routers, use advanced software algorithms to understand the network's topology and use that information to forward every data packet to its destination as quickly and efficient as possible. These software routers are interconnected by regular TCP connections. Just like on the Internet itself, programs connected to the TerraCast network can receive data as well as publish their own.

Features

Obviously the idea behind TerraCast was not to copy the Internet's protocols. Rather, it was designed to work around some of its deficiencies and be able to provide a network just as versatile and generic as the Internet, but with some important additional features.

One of these features is multicast. Where the Internet in principle only allows traffic to be exchanged between a single pair of hostsⁱ (unicast), TerraCast allows hosts to send a data stream to anyone on the network who is interested in receiving it. This is comparable to a TV station that is broadcast through satellites. Everyone who

knows the station's frequency can tune into it (provided it is not encrypted like many stations today). On the Internet of today that is not possible. It can only be simulated by sending the data stream to each individual receiver separately. While possible, this approach does not scale. It's a bit like broadcasting a TV channel by first running separate copper wires between the station and each viewer's home forming a huge physical star-network of cables with the station as its center.

TerraCast allows each user to broadcast a data stream to potentially thousands of receivers without the tremendous amount of stress that TCP/IP would cause. Being a generic, decentralized network, any host connected to the TerraCast network can multicast data streams.

Another feature that is missing in conventional networks including the Internet is that TerraCast allows for data streams to be delivered in degraded mode to receivers with insufficient bandwidth capacities. This is useful for sending realtime data streams to receivers on slow network links. Where the Internet reduces transmission speed to match the receiver's bandwidth, TerraCast remains sending at original speed (required for realtime data), but reduces the quality of the data stream insteadⁱⁱ. This way, the throughput is no longer held back by the slowest receiver. A feature that is very important when multicasting data to thousands of simultaneous receivers. Instead, each receiver will receive the realtime stream in the highest possible quality.

TerraCast can be seen as an "overlay network". A virtual network that operates independent of the characteristics of the underlying physical network (IP in most cases). Because of this, TerraCast is able to offer full multicast features where IP can't. To the underlying network, TerraCast is nothing more than a collection of applications that send data between static pairs. Beyond each router pair (identifiable by their TCP connection) there is no relation between the individual daemons. As such, the network can easily work through firewalls, NAT (IP masquerading), proxies and VPN's. Firewalls cannot control which TerraCast POPs can talk to each other. To understand this, look at how a HTTP proxy server tunnels traffic to and

from web browsers. Consider an intranet with two web browser machines. They both browse the Internet using a proxy server also on the local intranet. Although the individual web browser machines can surf the net individually, they never make a direct connection with a remote webserver. Only with the local proxy server that acts like a relay point. As such, the firewall that sits between the proxy server and the remote webserver can only choose to either allow the proxy server to talk to the webserver or deny it. It can in no way enforce unique policies for individual browser machines. It cannot even tell on behalf of which client the proxy server is fetching a webpageⁱⁱⁱ. TerraCast builds on the same basic principle. Because each software router is a relay point that tunnels data traffic for different senders and receivers much like a HTTP proxy server does, firewalls have no fine-grained control over TerraCast's communication. As soon as a firewall allows for only a single TCP connection between an internal and an external POP, all TerraCast routers connected to the internal one can talk to all POPs connected to the external one, and vice versa, without restriction. While this may make the network more susceptible to malicious applications, it offers a flexibility far beyond that of the current Internet.

IP Multicast

This section highlights the advantages of a TerraCast application level multicast network over a conventional network layer multicast.

Incrementally deployable. This is especially important when building a network that spans more than one administrative domain. Unlike IP multicast, where all networks must support it end-to-end, an application level solution can be deployed incrementally, one node at a time, while the service is available right from the start. The efficiency and performance of the service increases while the overlay network is expanded. This makes running an overlay service more flexible (another reason for incremental rollout could be lack of knowledge among network operators).

Possibly better robustness. Since the TerraCast overlay network runs its own adaptive routing algorithms, the content streams through the network are constantly rerouted to avoid the network's hot spots and congestion. This can be particularly useful on wide area networks that is used for very different types of applications and

hot spots are dynamic (i.e. moving around).

Another important advantage of having your own routing algorithms is the freedom of substituting them with others in the future. A very interesting development, especially in the face of multicasting non-interactive streams to many consumers, is that of load-balancing inside the network. Traditionally, routing algorithms seek for "best paths" through the network and send datastreams over these. However, it is sometimes much more interesting not to just send streams over this optimal path, but to also select several sub-optimal paths and divide the stream over all of them. This also avoids the optimal path from getting congested when the stream requires more bandwidth than this single path can provide. While TerraCast currently does not feature such routing algorithm, it is easily added at a later stage without requiring modifications of the underlying physical IP network.

Ability to incorporate application level intelligence inside the distribution network.

This is the motivation behind many application level middleware products. In the case of TerraCast, it allows the network to anticipate on situations of congestion where no possible path through the network can provide enough bandwidth. In these situations, the application specific intelligence reduces the quality of the congested data streams so that that the reduced streams can still be sent to the users downstream. Without such intelligence, a stream would be corrupted when a significant and unpredictable part of its packets are dropped. With conventional multicasting, one determines the available bandwidth of the slowest link and multicast the stream in that quality. This becomes less attractive when the network is large. A workaround is so-called *simulcasting*, where a stream is multicast in different qualities. This allows viewers to (either automatically or manually) select the stream that best fits their link speed. Simulcasting effectively wastes lots of bandwidth due to the inherent redundancy among the different streams. Also, when a receiver's bandwidth capacity temporarily drops below the critical level (possibly due to other link usage of background processes), the stream can easily get corrupted.

Fine-Grained Permissioning. The main concern of IP multicast is to get a single, non-interactive data stream to as many receivers as possible with the least possible overhead. It does not natively offer provisions for tracking the consumers, nor for authorizing them. All such features will have to be dealt with in higher layers

if they are required. TerraCast, being a higher level service that does not need IP multicast, does have features to track and even visualize data streams through the network. This is also easily used for entitling data streams to ensure only those consumers can receive it that are authorized.

Ability to bridge the Internet or third party networks. For collections of networks (especially those that are connected over the Internet or networks administered by third parties), overlay networks offer more advantages over IP multicast. These advantages include eased firewall traversal and the critical fact that incremental deployment allows a multicast service to be operated even when not all participating parties are capable (or willing) to use IP multicast. In these networks, overlay multicasting is the only viable solution.

Conclusion

The TerraCast network provides a network that is as versatile (and possibly as widely deployed) as the Internet, but manages to resolve some fundamental limitations of the old IP protocol. TerraCast can be deployed incrementally without requiring specialized hardware components. TerraCast manages this by offering a modern, scalable, packet-switched network as an infrastructure service, often referred to as an "overlay network".

MAC PAS605216.1-*--01/26/05 8:58 PM

i
.
.

Reliable Layered Multicast with Congestion Control

Guaranteed Partial Delivery in Congested Terracast Networks with Unrestricted Sender Speed

Erik van Zijst

Introduction

Having multicast functionality in networks is one thing. Many networks support it already in some form. The most wide-spread multicast architecture no doubt is IP multicast. Unfortunately, due to its many shortcomings that range from administration to scalability, it is disabled in most of the Internet. Aside from the Internet, there are other multicast solutions such as the MBone. However, very large, general purpose networks are scarce. One reason for this is the general lack of reliability in multicast delivery. Without guaranteed delivery, a transport service has only limited applicability. Over time, many mechanisms for reliable multicast have been proposed in the literature. PGM is an important example. Others are MTCP (a scalable TCP-like congestion control for reliable multicast in Java applications), TRAM and protocols that rely on Forward Error Correction such as RMDP and others. Essentially these protocols either slow down the source to a speed all receivers can handle (TRAM and MTCP), or they signal the receiving applications individually when they can't receive all data due to bandwidth constraints or excessive packet loss. The latter allows the sender to publish at a fixed rate and are more suitable for realtime transmissions. The former are useful only for bulk (file) transfers or networks with abundant capacity. Unfortunately, performance of both techniques is inversely related to the size of the network¹. When the source is being slowed down to match the slowest receiver, data will be delivered to all receivers, but this is not suited very well for very large number of subscribers. The larger this group, the larger the chance that one will be severely congested. When the source publishes at fixed speed, the number of receivers unable to keep up will increase as the total number of receivers increases, leaving those receivers with a corrupted data stream. In this text I propose a mechanism that allows the source to publish at a fixed rate to many receivers. However, to avoid the aforementioned problems, the source encodes its data stream in logical layers. These layers are given priorities and are used by the network's routers to decide which packets to discard in the face of congestion. Together with a tree based negative acknowledge mechanism, this allows the network to guarantee delivery of one or more of these layers to the receiving applications. When enough layers are used to encode the data in, with the lowest layer being thin enough to be received by all possible receivers, the network can guarantee delivery to all receivers, where each receiver will receive an uncorrupted set of data layers, regardless of its bandwidth.

¹ And the number of subscribers. The chance of encountering network problems such as congestion is related to both the average length of the distribution tree and the number of subscribers.

Types of Reliable Multicast Applications

It is important to make a distinction between different types of multicast applications, as each will require different characteristics from the underlying network.

Bulk Data

Bulk data applications are concerned with transferring files or other large blocks of data. Examples include distribution of corporate data, updating software, loading Web caches, or updating data warehouses. Such applications typically require: one sender per channel, no late joins, no real-time requirement, file level consistency, ordering, and full reliability.

Live Data

Live data applications typically come in two flavors. Those that require full reliability and those that don't.

Examples of the first are event logging and certain forms of stock quotes (tick-by-tick trade information) and basically every stream that cannot be delayed and whose packets require the contents of the previous packet. Full reliability and realtime delivery make this one of the more demanding applications. They typically require a network of which each link can match the bandwidth required by the source's publishing rate.

Note that many forms of live data applications do not require full reliability. An audio or video stream for example can often accept a certain amount of packet loss without becoming useless. While such applications may use unreliable multicast, certain forms of reliable multicast can often provide selective data repair and congestion control.

Collaborative

Collaborative applications are concerned with sharing data within a group. Examples include a shared whiteboard, collaborative editing, or controlling a multimedia conference. Such applications typically require: many senders per channel, late join with full data recovery, moderate real-time requirements, various consistency requirements, various ordering requirements, and full reliability.

Hybrid Applications

Hybrid applications are some combination of the above. Examples include Distributed Interactive Simulation (DIS) and other shared virtual reality environments. These applications often have many different requirements for different data streams or at different times. Some of them may be implemented as a combination of the above application types. Others may have special requirements that require special solutions (such as rapid joining and leaving of channels).

This text is written in the face of MarketXS and will therefore focus on the most relevant problem in this context: delivery of large quantities of stock quotes to many customers over wide area networks. Stock quotes fall into the category of resilient, live data. Under certain circumstances it is acceptable (and often even highly preferred) to omit certain stock quotes. Examples are desktop applications that display realtime updates for hundreds or sometimes even thousands of instruments. If all quote updates were delivered to these applications in realtime, this would require a staggering amount of bandwidth and would cause some of the more volatile instruments to refresh so many times per second, that it would be impossible for a human to read. At the time of writing most of the applications of this type work with these "thinned" data streams. Aside from end-user products, at least one major data vendor (S&P Comstock's delayed satellite feed) even "thins" the streams before broadcasting them to their customers.

Before running these applications over unreliable multicast networks, it must be noted that "thinning" market data streams cannot be done by discarding just a random percentage of the quote updates. How updates can be omitted without corrupting the stream is described in previous work .

In this text a mechanism is introduced that can be used to reliably deliver near-realtime financial information to many concurrent receivers using multicast over wide area networks. To this end the overlay network facilities provided by the Terracast network are used. A description of the Terracast network is found in "Technical Overview of the Terracast Network" and "Live Multicast on Wide Area IP Networks." The Terracast network is essentially a packet switched network in overlay mode that supports unreliable multi- and unicast. The multicast subscription facilities are offered by the sparse mode multicast tree protocol that was built into the Terracast routers as described in, "Computing Sparse Distribution Trees for Extended Bellman-Ford." To allow flexible congestion control, the Terracast network also got mechanisms for equally dividing a link's bandwidth over all data streams and prioritization of packets within data streams. The latter allows source applications to divide their data streams into logical layers. When congestion occurs somewhere in the network, only the layers with the highest priority are forwarded. This is especially useful when multicasting a data stream with a high data rate to a very large number of receivers. Receivers that are on congested parts of the network will then receive the most important parts of the data only. This mechanism of packet prioritization is described in "Fair Bandwidth Allocation and Packet Prioritization for Terracast." Packet prioritization eliminates the problem of lowering the publishing rate to match the slowest receiver, while still being able to offer uncorrupted, partial data streams to clients suffering from a (temporal or permanent) lack of sufficient bandwidth.

The concept of dividing a (multicast) data stream in logical layers and allowing receivers to receive as many layers as bandwidth capacities permit, so as to avoid congestion, is an active area of research. One of the pioneers of layered multicast schemes was Van Jacobson who divided a data stream in layers that were sent to individual IP multicast addresses. Aside from layering network protocols, there is also research into applications that would benefit greatly from networks with functionality such as proposed by Van Jacobson. Examples here are audio and video codecs that divide the data over layers. In

LVMR, a simple technique is shown that encodes any MPEG 2 stream in 3 layers. As long as bandwidth permits, Van Jacobson's receivers would subscribe to more and more layers until packet-loss is detected which is then interpreted as congestion. Van Jacobson's solution lacks guarantees and reliability. Van Jacobson's clients periodically add subscriptions to new layers, actively causing congestion. This congestion causes packet-loss also in the other layers and because plain IP multicast is a best effort service, the packets cannot be recovered. Hence, every time Van Jacobson's client take a peak at network availability, they corrupt their own content. Terracast's packet prioritization cannot guarantee delivery of all packets in all layers either. As it is provided as a best-effort service at the router level, there is no hard guarantee that no high priority packet is occasionally lost. Like IP multicast, Terracast at this point has no way of repairing lost packets.

Objectives

For Terracast to be useable for multicasting high volume, realtime data streams to very large numbers of receivers, even when parts of the network are temporarily or permanently congested, the layered data streams and packet prioritization scheme needs to be extended with packet repairing mechanisms, so there can be a guarantee that subscribers will receive one or more layers of the stream that are completely intact. When an occasional packet of a high priority layer is missed, it should be repaired. However, the receiver should not attempt to repair lost packets of low priority layers that are incomplete due to network congestion. Trying to repair these layers would require even more bandwidth, resulting in more packet loss. Yet, as more end-to-end bandwidth becomes available, the receiver should detect this and repair additional layers and forward them to the user application. When layers are dropped due to congestion, the programming API that is exposed to user applications should notify the user by means of an exception or alternative return value when it is processing packets. These notifications should be non fatal.

Localized Packet Repair

I propose a receiver-side component that receives the raw packets from the network and is in charge of repairing damaged stream layers before forwarding them to the application. To this end I introduce a retransmission request packet. When the receiver detects a missed packet in a layer that is to be repaired, it sends a retransmission request for this packet to the source. Detecting packet loss is done in the conventional way by sequencing each packet with an incrementing number. In the present case however, each layer has its own sequence number so all packets in one layer are sequenced independent of the other layers. This way packet loss can be detected in every layer, regardless of the conditions of the other layers.

To avoid a cascade of retransmission requests when a packet is dropped close to the source, a tree-based negative acknowledgment protocol is used similar to those used in PGM and TRAM. This technique can provide local packet repair without communication with the source. Illustration 2 visualized this. Local repair works by having each router store a window of transmitted packets. Packets are either stored for a fixed period of time

after which they are discarded, or a fixed amount of buffer space is reserved to store the most recently transmitted packets. In the case of Terracast, a slightly different approach is taken. Since Terracast by default uses reliable TCP connections between nodes, packets will never get lost while in transit between nodes. The only place where packets are discarded is in the outbound interface buffers before they are transmitted². Hence, there is no use in buffering what has actually been sent, as those packets are guaranteed to have reached the neighbor node. Instead, packets *received* by the interfaces are stored in the retransmission buffer immediately. This is visualized in the illustration below. However, storing incoming packets at this point, before they are being processed by the router core, could lead to packets whose time-to-live field has not yet been decremented (depending on the location in the inbound packet pipeline of the ttl-decrementing module). It is also possible to store packets immediately when they are passed to an outbound interface

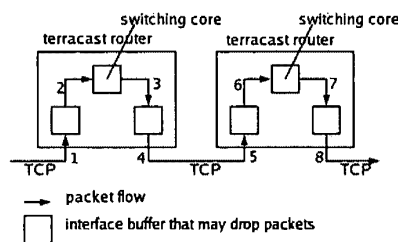


Illustration 1: When maintaining a packet buffer to locally service retransmission requests, packets should be collected before they enter a lossy component. Points 1 and 3 are useful (5 and 7 respectively), 4 is not

(position 3 and 7 in the illustration). This is before the packets reach the internal interface buffers where packets can be discarded during congestion. The downside of giving interfaces individual packet buffers is that duplicate multicast packets will often end up in two or more packet buffers when the router acts as a branch point in a multicast distribution tree. A solution may be to combine all packet buffers into a single data structure that stores only unique packets. Which approach is best also depends on storage prices. When a router that continuously transmits 10 megabits per second on each of its 3 links, it would require 225 Megabyte of memory space to store an outgoing window of 60 seconds for each interface. In fact, things could be even worse when the outgoing links are actually congested and the actual outgoing traffic rate offered to the interfaces was in fact 20Mbit/s. In that case the router's interfaces have already discarded 50% of all

² Note that it is crucial for Terracast's packet prioritization scheme to use reliable communication between peers. Only then the network is fully capable of deciding which packets should be dropped and which should be forwarded.

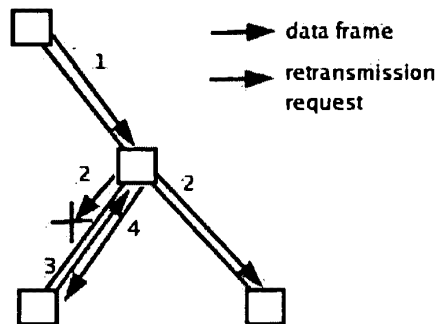


Illustration 2: *Reliable multicast with tree-based acknowledgements. (1) Data frames are sent by the source. (2) The branch point sends a copy of the frame to both neighbors, but one packet is lost. (3) The receiver detects the packet loss and sends a retransmission request to the source. (4) The branch point has a copy of the data frame buffered and responds.*

outgoing traffic, but since that traffic did get buffered in the retransmission buffer, 450MB of memory is now required. A practical way of going about the storage capacities is probably to reserve a fixed amount of combined buffer space and always use all of it. This implies that the time each packet is kept for retransmission requests depends on the total outgoing traffic rate of the router³. The busier a router gets, the smaller the chance it can respond to a local repair. When a retransmission request cannot be answered locally, it is simply forwarded to the next hop in the shortest path to the source of the data. The closer the request gets to the source, the bigger the chance it can be answered. A potential problem of limited packet storage capacity in routers, especially near the center of the network is that because they will be able to answer to fewer retransmission requests, they will forward more of those requests, resulting in more outgoing traffic. Also, when the packet is eventually repaired by a router closer to the source, the new packet passes the overloaded router once more, increasing both incoming and outgoing bandwidth. When the router is causing congestion on its link because they are too slow to transmit all packets, these retransmissions will cause other packets to be dropped. On their turn, these may lead to more retransmissions.

³ It is important to understand that storing copies of messages in the network can significantly increase the time a packet can survive in the network. However, it is crucial that all previously stored packets are purged by the time the sequence numbers are reused to avoid echos of very old packets to end up in new streams. For this reason TCP uses the TIME_WAIT interval to wait for all old packets to die before reusing network addresses.

Selective Packet Retransmissions

The algorithm monitors the status of the total stream and from this information derives how many layers of the stream can be reliably delivered without stressing the network. It will then mark these layers as intact and will request retransmissions when an occasional packet is lost. In principle the algorithm will only deliver packets from the intact-layers to the application, but not before they have been put back into their original global order. The process of global packet restoring is described in the next paragraph. In order for the algorithm to decide which layers can safely be marked as intact, it needs status information. Every packet comes with a list of packets from the other layers to which it is causally related. This means that aside from bearing its own sequence number, each packet contains the current sequence number of all the other layers too. Now by inspecting these sequence numbers each time a packet comes in, the receiver can see if it has missed any packets from the reliable layers. If that is the case, these will have to be repaired. Every time a missed packet is detected, a countdown timer is scheduled for it. If the packet is recovered before the timer expires, the timer is canceled and removed. If however the timer manages to expire, the layer is considered impossible to repair and is, together with all higher reliable layers, removed from the list of reliable layers. How long the time-out interval should be can be application dependent. Essentially all it determines is the how much data may be delayed. Setting it to a very low value of a few seconds means a temporary congestion can potentially corrupt a layer long enough to cause the receivers drop layers. Note that since packets can occasionally also get lost when routing tables converge and change the shape of distribution trees or when a router crashes, appropriate time-out values will probably have to be found experimentally. Also, it is important to realize that out of order delivery essentially means one or more packets are delayed. In a Terracast network with packet prioritization, routers explicitly introduce out of order delivery when a burst of packets queues in front of a busy router-to-router connection. Hence, the time-out should be large enough to allow for this. Setting it to a very long period makes the stream much more resilient to congestion, but also increases the time for the receiver to discover a layer must be dropped due to bandwidth constraints. Until congestion is finally detected, delivery of the previously received packets that depend on expired packets is postponed. Note that the system of setting time-outs on missing packets allows for congestion detection, it does not guarantee low latency. As long as no packets are lost and all are delivered in global ordering, the receiver cannot measure the total transmission delay. As such, a receiver cannot natively distinguish between an idle and a crashed source. Of course this applies to most best-effort packet switched networks. Aside from removing layers from the reliable list, the algorithm must also be able to detect when a new higher layer can be added to the repair list. This is done by passively monitoring the lowest layer that is not under repair. Every time a packet is received for this layer, the fact that it is received is stored for a period, equal to the repair time-out described earlier. When a packet from a different layer is received that causally depends on a packet from our layer that wasn't yet received, then the fact that we are now waiting for this packet is also stored for a period equal to the repair time-out. If during this time-out the packet is received after all, possibly because it was simply delivered out of order, the stored record is marked as received. The time-out is not altered. When a whole bunch of packets is missing (either because one of the

packets came in whose sequence number has a gap with the last packet received, then one timer for each missing sequence number is scheduled. Each time a time-out expires, the corresponding sequence number is inspected for the received flag. The packet slot is then removed. If a slot that was marked as lost has been removed because its timer expired, it's gone. If the packet still arrives after that, it is simply discarded. What this scheme gives is an in order list of sequence numbers marked either as received or lost. This history of packet arrival information is then used to determine whether the reception quality was high enough to add this layer to the repair list. For example, the reception information can be used to calculate a moving average that describes the average packet loss over the last x seconds. It seems natural to use the same time-out period as for the repair an reception interval. With a small time-out, say 10 seconds, the receiver aggressively anticipates congestion, dropping layers even during short bandwidth problems to keep the the delay of the delivered data minimal. When the time-out is high, the receiver is more tolerant to bandwidth fluctuations, dropping layers later, and adding new ones only after the network proved to have sufficient capacity for a longer time. As an optimization to make the algorithm add consecutive layers more quickly, the receiver can passively monitor all layers not currently under repair, rather than monitoring only the next layer. This way when the network suddenly becomes fast enough to receive all layers, the receiver will immediately add all higher layers, rather than one at a time. The way the receiver keeps track of the status of the other layers is through the list of sequence numbers in each packet. Unfortunately this causality information adds overhead to each packet which size is related to the number of layers in the stream. When sequence numbers are 32 bits and the publisher uses all 256 available layers, each packet comes with a kilobyte of causality information, which contributes to at least 12.5% of a 8kb packet that is filled to the brim. There is one straightforward optimization that can be made by only including sequence numbers that have changed since the previous packet. This means that the source publishes two packets with the layer in a row, the second one will not contain any status information, as the only sequence number that has changed is its own, not any of the other layers. However, when packet layers are cycled, sending one packet per layer each time, then each packet will reflect the change in each layer, still causing maximum overhead. To ease layer administration, a stream only supports a static number of layers. When a publisher binds a multicast address for reliable, layered communication, it explicitly specifies the number of layers that will be used.

Restoring Global Packet Ordering

With default operation, packets are passed to the application in the original order. Not only are the packets in the individual layers restored to their natural order using their sequence numbers, but the total ordering across the layers is also restored. When the source sent three packets with different priorities (each in a different layer), they are all three received by the receiving application in the exact same order. To restore the global packet order between layers, each packet carries the sequence number of each layer. A prioritized, layered packet is described as $P_{2(9, 4, 6)}$ where 2 represents the packet's priority (layer), 9 the last packet of layer 1 that was sent at the time this packet got published, 4 the sequence number of this packet and 6 the last packet from layer 3. Hence this means

that our packet causally depends on packet 9 from layer 1 and packet 6 from layer 3 and is only delivered to the receiving application after those have been delivered. This mechanism of recording causality in sequence vectors and piggybacking them with the messages is somewhat inspired by the application of logical clocks and vector timestamps used in communication of some distributed systems. The illustration below shows the process of selective repair and reordering.

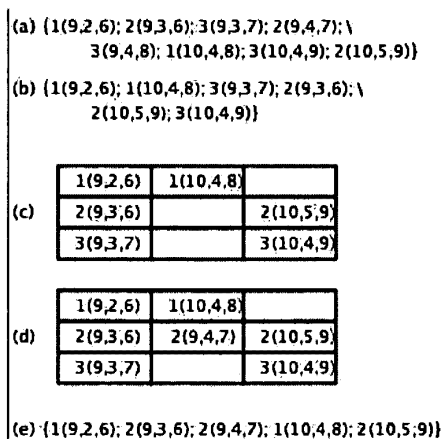


Illustration 3: The process of selective recovery and reordering. (a) A packet stream as sent by a source. (b) The packet stream as received by a subscriber. (c) Packets as stored in the subscriber's receive buffer. (d) Packets after selective repair. (e) Reordered packet stream as delivered to the application.

The notation {1(9,2,6); 2(9,3,6); ...} is used to describe a sequence of packets where $P_{1(9,2,6)}$ was sent prior to $P_{2(9,3,6)}$, etc. The first line labeled (a) thus shows the sequence of packets originally published by the source. It reads from left to right, so $P_{1(9,2,6)}$ was published first, followed by $P_{2(9,3,6)}$, etc. In this example the source has published 8 packets, divided over 3 priority layers (1, 2 and 3, where 1 is the highest priority). The second line (b) shows the packet stream as received by one of the subscribers. It shows 2 lost packets and an incorrect ordering. Before the packets are delivered to the user application, they are stored in internal buffers used during the repair and reordering process. This state is depicted in (c). Here the packets are in separate buffers, each representing a priority. The packets are also ordered inside the buffers. The illustration shows the missing packets $P_{2(9,4,7)}$ and $P_{3(9,4,8)}$. If the receiver currently has only layers 1 and 2 under repair, it will attempt to repair the hole in layer 2 by sending a retransmission request. Note that if the given sequence was just a snapshot of a running stream, the receiver would already have detected the missing packet $P_{2(9,4,2)}$ by the time $P_{1(10,4,8)}$ was received. After all, this packet says it depends on packet #4 from layer 2. So even before

a layer 2 packet was received in our snapshot, the receiver already detected packet loss in that layer and immediately scheduled a time-out for the missing packet. In fact, if we assume the receiver had received packet #2 from layer 2 prior to our snapshot, then the conclusion would be that both packet #4 as well as #3 were lost. An individual time-out is scheduled for both and a retransmission request is sent. Shortly after packet $P_{2(9,3,6)}$ is received simply because it got reordered in the network. Hence, the receiver places the packet in the receive buffer and cancels the time-out. This is illustrated in (d). Note that the hole in layer 3 is also detected when $P_{1(10,4,8)}$ is received. This packet claims to be sent after packet #8 of layer 3, so this must either have got lost or delayed. Since layer 3 is not under repair, a retransmission request is not sent. However, as for all layer 3 packets, a time-out is scheduled for sequence number 4 of layer 3. When the user application is ready to read packets, the algorithm will return only those packets from the layers marked for repair. Even though some of the layer 3 packets were received, they are discarded and not delivered. The resulting stream of delivered packets is a copy of the one published by the source, with layer 3 removed, despite the fact that the network had insufficient bandwidth to deliver the whole stream, had dropped packets from various layers and messed up the packet ordering.

In most network protocols that deal with retransmissions and packet reordering, packets are stored in a receive buffer that has a maximum size. In TCP for example, the receiver no longer accepts packets if its receiver buffer is full. For TCP this is not a problem, as the source is notified of this event and stops sending more packets. The receive buffer can become full either if a packet retransmission takes very long, while the set of pending packets builds up, or when the user application simply doesn't pick up the packets. In Terracast the first problem does not really apply. Since Terracast sources are realtime streams that will not slow down or pause while at the same time delivery of packets to the user application should not be delayed for too long, receivers attempt to recover lost packets only for a limited period of time. Whether a packet is received, repaired or lost, its slot will be freed after this time-out, keeping a natural size limit on the receive buffer. How big this can get is related to the average speed of the stream and the repair time-out used, but in principle the receiving algorithm should not impose a hard size limit during packet repair. However, a different matter is when the user application reads data from network slower than it is received. When this happens the user application apparently causes congestion in the receive buffer. A safe way of handling this could be to remove a layer, decreasing the amount of data that is delivered, while remaining near-realtime. If a clean feedback algorithm that can keep the number of layers in balance with the application's reading speed is not used, I suggest throwing a fatal exception to the user application and closing the stream when the receive buffer exceeds a certain maximum size, or when the total time between arrival of packets and actual delivery reaches a threshold.

Ignoring All Packet Ordering

Although restoring the original global packet ordering before delivering the data to the user application is assumed to be useful for most types of data, there is realtime content

where each update invalidates all previous data. This is especially true for stock quotes. When a new stock quote is received for a financial ticker, it renders the previous quote useless. For most applications that process realtime financial data, only the most recent information is interesting. When global ordering is restored, the algorithm will postpone the delivery of the most recent updates until all prior updates have been received also. For a typical application that displays stock quotes such as so-called terminals, this postponing adds little value. When the burst of updates is eventually delivered by the network layer, the application will update the ticker's quote on the screen, thereby leaving only the last and most recent update visible. The others were overwritten immediately. In fact, because the reordering process adds additional delay to the data delivery, it can be switched off by applications that don't benefit from it. Without reordering, global (causal relations between layers) and local ordering (ordering of individual packets of a single layer) is ignored and packets are delivered to the application immediately after they have been received. This does not disable reliability however. Packet loss in those layers marked for delivery is still repaired, only the repaired packets will be delivered with additional delay. Whether or not this makes them useless is up to the application to decide. To determine whether a packet delivered by the network is older or newer than other already received, the source may want to add some application specific timestamp in the packets. Alternatively, when ordering does matter to some extent (as in stock quotes where a newer one may override an older one, but not the other way around), the receiving application could inspect the packet's sequence vector. However, since these numbers are used by the network layer, it is generally not recommended to also rely on them in a higher layer. In principle it is possible to take advantage of the lack of ordering by omitting the sequence vectors from the packets. Obviously a packet will still need to bear at least a sequence number that represents its location inside its layer so packet loss can be detected. While this may appear as an efficient tradeoff (gaining bandwidth efficiency by giving away ordering), it does introduce another problem. With the sequence vectors gone, packets only maintain state information of their own layer. When suddenly no packets are received for a certain layer, the receiver is unable to tell whether it is simply idle (and doesn't require retransmissions), or whether all its packet are getting lost for some reason. As a result, the receiver cannot decide whether the layer should be dropped (because it is corrupted), or kept (because there is nothing wrong). Hence, it appears the packets should ship the same state information regardless of the receiver's needs.

Ignoring Global Packet Ordering

Aside from ignoring all forms of packet ordering, the receiver can also be configured to only ignore the global ordering of packets from different layers, but to keep the ordering of packets within layers intact. Whether or not this is useful will depend on the type of application and content, but it offers the advantage that delivery of packets from lower (high priority) layers are not delayed while higher layers are under repair.

Applying Reliable Layered Multicast to Market Data

As an example the preferred way of applying reliable layered multicast to financial data distribution is discussed in the context of MarketXS. The main objective is to use the Terracast multicast features to deliver a single realtime stream of quote updates to as many receivers as possible. In this example the stream contains updates for all tickers on a single stock exchange. As the required bandwidth to distribute each and every update in realtime is highly depended on the time of day, the type of exchange and environmental conditions⁴, it may not be possible to always deliver everything as is⁵ as that would require one to dimension the entire distribution infrastructure for the highest possible peak, which is usually not economically feasible. Instead a source application publishes the data at a fairly high rate of say 8 Mbps. This could be encoded in an even higher rate if necessary and useful, but since the physical distribution network that will usually be a mix of private and public links doesn't have infinite bandwidth, 8 Mbps seems a reasonable maximum for this example. Before the stream is packetized, the quote updates are categorized by ticker and by timestamp. The algorithm used to do this is the Incremental Multi-Frequency algorithm that was described in previous work. When quote updates from a particular layer are ready to be transmitted, they are combined inside one packet. Sending each update as a separate packet is inefficient in the network, so packets are filled with as many quote updates as possible. It is important however that quotes with different priorities (from different layers) are never mixed in packets. Packets submitted to the network are then given their local and global sequence number and are injected into the network. When the packets travel through the network, they follow the path of the sparse multicast distribution tree that is maintained by the interested receivers. When the stream encounters a congestion, only the most important ones are forwarded. Note that congestion can be the result of a slow physical connection between two routers, or an underpowered CPU of a router that is not capable of processing the incoming packets fast enough, or because the available physical bandwidth on the link must be shared with other data streams running between other nodes. As the Terracast network is a best-effort packet switching network, bandwidth is shared and never guaranteed.

When the stream finally reaches all its receivers after being branched and thinned at various places, the subscribers use the Selective Packet Retransmissions algorithm to restore repair those layers that should be delivered. Again, note that missing packets from layers that are not under repair are not recovered to avoid additional network congestion. Since the stream conveys stock quote updates and since quote updates invalidate their previous ones, it is undesirable for the network layer to postpone delivery of new updates because its successors have not yet arrived. Hence, the receiver disables all packet reordering and instead uses an application specific mechanism to track which update to process and which to discard. When a packet is received, its quote updates are extracted and stored in an array. In this array, each position represents a financial ticker symbol (tickerId in MarketXS terminology). Each symbol can thus store one quote update. After having stored the updates, a copy is forwarded for further processing (often merely

4 Environmental conditions in this context refer to events in the real world that may influence trading behaviour. Examples of bankruptcies of competing companies, accounting fraud or war often cause peaks in trading activity.

5 Which is also the reason why at least one major data vendor continuously thins its delayed data feed.

displaying it to the screen). When a quote update cannot be stored because its place in the array is already occupied by a previous update for that ticker symbol, their timestamps are compared. If the timestamp of the new update is higher than the one already stored (the new update is more recent), it overwrites the old update and is forwarded for processing. When the new update is in fact older than the one in the array, it must have been delayed in the network and has become useless. It is immediately discarded. Because the network guarantees the delivery of all packets from the subset of layers that it selected based on bandwidth capacities, not a single crucial quote update is ever missed. This is similar to the features offered by the MarketXS SOAP services.

Implementation

In TerraCast, the retransmission mechanism is realized through normal unicast UserData messages addresses to the source node of the multicast group. Their own source address is that of the receiving node. To this end, the communication endpoint implementation of a reliable, ordered multicast session on the receiving side binds both the target multicast group in read-only mode and binds an anonymous unicast port that it uses to send retransmission requests. The endpoint of the source of the multicast group also binds two ports. It binds the local multicast group in publish-mode and the unicast port that has the same name as the multicast group. When a receiver notices it missed a packet, it sends a unicast retransmission request towards the source of the multicast group. This retransmission packet is a subclass of the UserData class and carries the layer number of the missed packet, as well as its sequence local number. It does not explicitly carry the name of the multicast group as that information is already implicitly contained in the destination address of the retransmission packet. Without localized packet repair from the intermediate routers, the unicast packet will eventually reach the source's endpoint. To be able to respond to retransmission requests, the source's endpoint maintains a history of packets that were sent. Messages are kept in this buffer only for a limited period of time during which packet repair is possible. If the requested packet is still in the packet store, the endpoint implementation reinserts the multicast packet in the outgoing stream once more. It does not update the packet's timestamp in the temporary packet store. When a retransmission request is received for a multicast packet that was already expunged from the packet store, an error is returned to the initiator of the retransmission in the form of a NotifyData message with code MESSAGE_EXPIRED, indicating that the requested message does no longer exist. When the multicast receiver eventually receives this notification through its unicast port, it knows it cannot repair the packet loss, it will inform its user application and continue processing the rest of the received, pending multicast messages. When either the retransmission request, the error notification, or the actual multicast packet retransmission fails, the multicast receiver will detect this through a timeout and send another retransmission request. Without help from the intermediate

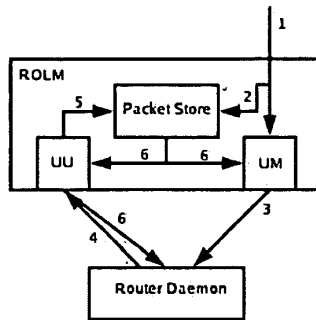


Illustration 4: Implementation of the ROLM communication endpoint used for publishing multicast packets. (1) A packet is received from the user application. (2) The packet is stored in the retransmission buffer. (3) The multicast packet is sent. (4) A retransmission request comes in. (5) The lost packet is looked up in the packet store. (6) Depending on whether the packet was found in the packet store, it is either published again to the multicast stream, or an error notification is unicasted as response.

routers, each packet retransmission will be handled by the multicast source. The localized packet repair mechanism is realized through interceptors in all interface interceptor pipelines. This system is illustrated in illustration 5 where two adjacent TerraCast routers are displayed. The picture shows the two interceptor pipelines inside the interfaces. Each interface contains both an inbound interceptor pipeline that processes messages received from the network and an outbound interceptor pipeline that gets a chance to process the messages just before they leave the router daemon. The pipelines show two different interceptors, called CCI (Congestion Control Interceptor) and PRI (Packet Retransmission Interceptor). A real-life router usually contains more interceptor instances per pipeline, but the most important one that is always present is the CCI (Congestion Control Interceptor) which is the unit that implements TerraCast's bandwidth allocation rules bandwidth allocation rules and is responsible for dropping packets. During normal operation what happens is that the source publishes its packet stream to the router

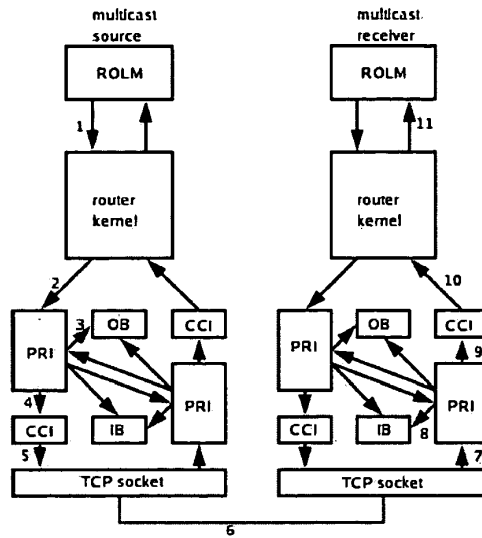


Illustration 5: Normal flow of packets in a reliable, ordered multicast transmission without packet loss.

daemon that sends the packets to all interfaces that lead to interested receivers. In our example that means the packets are passed to the first outbound interceptor (PRI) of the interface. The PRI is responsible for storing a copy of each recoverable (sequenced)

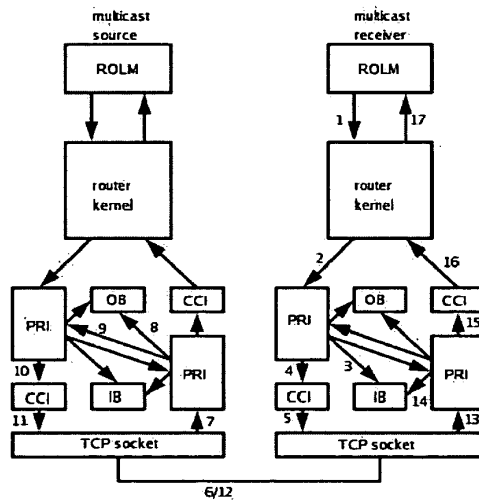


Illustration 6: Normal flow of packets in a reliable, ordered multicast transmission when a packet that was dropped by the source router is recovered by the receiver.

UserData packet that is transmitted. It stores a copy of the packet in the private temporary packet store of its interface. Then the interceptor passes the message to the next interceptor (CCI) where it is buffered until the TCP connection with the adjacent router has time to transfer it. If the packet is not dropped in this CCI, it is received by the neighbor router and sent through that router's inbound interceptor pipeline. It first enters the PRI, which stores a copy of it in its inbound buffer (IB in the illustration) and then passes it to the next interceptor where it is temporarily parked until the router's kernel thread picks it up and delivers it to the receiving multicast endpoint. When a packet is lost in the outbound message interceptor pipeline of the source router because the TCP connection between the two routers was not fast enough to transmit the packet stream, the receiver will notice the loss and send a retransmission request. This process is depicted in illustration 6. This unicast request message's destination address is that of the multicast source. Through the router's kernel it reaches the outbound message interceptor pipeline and goes into the PRI (2). This interceptor scans specifically for retransmission requests and tries to process them locally. To this end it inspects the message, looks up the multicast group, the stream layer identifier and the layer's sequence number and then checks the inbound message buffer (IB) to see if it contains that message (3). In our example that isn't the case, so the message is passed on to the rest of the interceptors (4) and eventually transmitted to the adjacent router (5,6) where it is fed to the inbound interceptor pipeline (7). The PRI checks the interface's outbound packet buffer (OB) (8) and finds the multicast message that was lost in the outbound CCI earlier. It is then re-inserted into the outbound message stream via the outbound PRI (9). The retransmission request message is then dropped and not forwarded any further. Assuming the message is not dropped again, it travels the normal way towards the receiver (10-17). At (14) the receiver's PRI sees the multicast message for the first time and stores a copy in its inbound packet buffer. This mechanism of unicast retransmission messages and interceptors that offer localized repair is flexible in the way that not all routers need to support local packet repair. A router that does not support it simply forwards the requests and lets the upstream routers handle it. Of course the network as a whole becomes more efficient when more routers support it.

The MESSAGE_EXPIRED notification that is sent when a retransmission is requested for a message that had timed out has the source's node address and unicast tport as sender address. It is important that the PRI's remember the fact that when multicast message is expunged from the in- and outbound packet buffers, it had expired. They must use this information to autonomously generate MESSAGE_EXPIRED notifications when receivers ask for it. If the interceptors do not do this, they would propagate retransmission requests for expired messages to the source. This could lead to a so-called nack-implosion when a multicast message is rexpunged from the buffer, while receivers continue to ask for it. After some time, the buffer should also remove the expired record from the buffer to allow the multicast group and sequence numbers to be safely reused in the future.

The retransmission NotifyData message is a special one in that it will never cause a reply. For example, when the notification is sent by an intermediate router because the requested message was expired, but the notification cannot be delivered because for

example the message's TTL is too short, the router that drops the message will not inform the multicast source (the official source of the notification message) of this fact. Again, this is to prevent nack-implosions under certain circumstances.

Merging NACKs

Aside from intercepting retransmission requests and storing recoverable multicast messages, the PRI's also need to merge duplicate retransmission request packets and make sure only one request is sent to the upstream router. This is to prevent a NACK-implosion when a packet is dropped in a router that is in the distribution tree to thousands of receivers. Without NACK-merging, all receivers will detect the lost packet at the same time, causing the router to be buried in thousands of requests for the same data packet and re-inserting the same data packet in the multicast stream equally often. In TerraCast, the PRI in the outbound interceptor pipeline of an interface remembers that it has seen a certain retransmission request message for a short time. If a duplicate request (usually sent by a different receiver) reaches the outbound PRI, it is simply dropped and ignored.

Conclusions

In this text I have proposed a mechanism that can add reliable, guaranteed delivery of partial multicast streams over a Terracast network. The mechanism differs from many reliable multicast solutions proposed in the literature by keeping delivery realtime, without the need to slow down the source when network congestion occurs. The scalable packet recovering scheme makes delivery of partial streams entirely deterministic. I have demonstrated an example application that offers a very flexible, efficient and above all scalable way of distributing large quantities of near-realtime market data to large numbers of subscribers using the Terracast overlay network infrastructure. The network's congestion control mechanism ensures that data streams are not delayed or buffered and remain near-realtime even if one or more links in the network are lacking sufficient capacity. By letting the source decide which packets are more important than others, the network is able to deliver a stream that is tailored to each receiver's capacity in near-realtime. In addition to the congestion control mechanism, the network uses a scalable algorithm with localized packet repair to ensure guaranteed delivery of well-defined subsets of the stream. This partial reliability in combination with the source's data encoding ensures that crucial quote updates are never missed and that given the performance of the network, the maximum possible delay of each received stock quote can be calculated. Note that these are characteristics comparable to those of the MarketXS SOAP services that have successfully used similar stream encoding techniques to keep market data delivery near-realtime and deterministic, even during severe network congestion. What the Terracast network adds is full multipoint distribution over existing wide-area network infrastructures that lack multicast functionality themselves. Combined this should offer a distribution network much more flexible, scalable and possibly even more economical than what is currently operated by MarketXS.

FRS PAS605212.1-*01/26/05 7:58 PM

PATENT APPLICATION SERIAL NO. _____

U.S. DEPARTMENT OF COMMERCE
PATENT AND TRADEMARK OFFICE
FEE RECORD SHEET

02/03/2005 EABUBAK1 00000033 60647601

01 FC:2005	100.00 OP
02 FC:2085	125.00 OP

PTO-1556
(5/87)

*U.S. Government Printing Office: 2002 — 489-287/69033