Terracast

Live Multicast on Wide Area IP Networks

Erik van Zijst

Fall 2004

Vrije Universiteit,

Faculty of Sciences, Departments of Mathematics and Computer Science,

Amsterdam, The Netherlands

erik@marketxs.com

1. Abstract

Contents

2. Introduction

3. Project Background

The Terracast project was started almost two years ago as a research effort at MarketXS to see if it was possible to use the ever expanding, public Internet as an efficient, scalable backbone for distributing live market data across the globe. At the time MarketXS operated a centralised financial *ticker plant* that was used by customers to obtain market data for use in their own backend systems or websites. Data was accessed using a programming API and exchanged over direct TCP connections. As the focus moved towards providing large volumes of live streaming market data to an increasing number of customers, MarketXS found itself distributing largely the same non-interactive content over the Internet to individual client applications using standard unicast TCP connections. The cost of both processing power at the distribution part of the ticker plant, as well as the necessary Internet bandwidth, increased linearly with the number of connected clients. This is different to how larger competitors handle data distribution. Coming from a less globally wired past, the major competitors have traditionally used satellite broadcast systems to get non-interactive live content to their customers. Although this comes with fairly high initial costs, it scales practically infinitely with the number of clients. MarketXS, being founded at the end of the Internet hype, has relied on the Internet as a flexible and cheap communications medium which appeared to be suitable for all but the most demanding institutions. However, as the amount of data and the number of clients grew, the need for a more scalable distribution mechanism emerged. Satellite was not favoured, as that would invalidate the service's selling point that no client side hardware was required. Ideally, market data would be published to the Internet just once with a fixed cost, where it would be multicast to all paying customers. Unfortunately, since the Internet does not allow for such distribution natively, a virtual network was designed to reduce the amount of required bandwidth at the ticker plant without abandoning the Internet.

Because the Internet offers a best-effort delivery service of which the quality and throughput can greatly vary, the new virtual distribution network should be very robust with no single points of failure and should be able to constantly deliver its data with minimal delay, regardless of network congestion.

4. Problem Description

The Internet has been tremendously popular and has seen phenomenal growth during its few decades of existence. Especially since the birth of the World Wide Web (footnote; http, mosaic) the amount of online content has been spectacular and steadily growing. Today the Internet (TODO: describe its sheer size with reference). 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 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. In this thesis we will study 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, we will introduce a software based system 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 – PSTN, most cable television networks and radio are 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 is reserved on the network and can never be used by other streams. This puts a hard limit on 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, often causing live video streams to appear jerky and frames to be skipped.

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. If today's Internet would globally support multicast traffic, we could use it to multicast a live video stream. However, since the stream is transmitted in a fixed (usually 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. This corrupts the data stream that is eventually received by one or more receivers further down the network. It also has a negative impact on communication sessions of other nodes using 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 favor one of yours when the overloaded link is ready to send another packet, implicitly rewarding heavy streams during congestion. If you add enough redundant packets to compensate for the loss of a significant part of them, you could effectively steal all link bandwidth 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 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. While audio and video data can usually withstand quite some 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 rely on if those lost packets are important trades. Under these circumstances the network will have to offer enough guarantees regarding delivery to satisfy applications that cannot withstand arbitrary packet loss introduced by overloaded routers and links.

Following from these requirements are the features that we think a packet-switched network will need to support in order to be a useable transport medium for dissemination of live, non-interactive content to large audiences.

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 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 the right packets to each receiver. Multicasting can also be done by letting the source encode the list of receivers 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 and one approach has even been patented [MAR04]. A third approach places all logic at the receivers by letting the network apply a broadcast mechanism whereby each packet is delivered to every connected node and the receivers filter out only those packets that are interesting. Although this is very wasteful, its simplicity can still make it attractive under certain circumstances. Even though it does not use packet-switching, conventional radio and cable television do exactly this.

Adequate Flow Control and Timely Delivery

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 by how much the network is overloaded by all combined streams 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, similar to when TCP connections are used. However, letting data streams 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 remains that live streams often lose their value when they are slowed down and delivered late.

Guaranteed Delivery

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 network has limited capacity, packet loss is hard to avoid. Even when there is sufficient bandwidth at all times, store-and-forward [footnote] 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, packets can also be dropped when routing policies change or packets are trapped in an occasional, temporary routing loop [footnote that explains temporary routing loops]. 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 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 has well-known applications in [1], [2] and [3], but may lead to aggressive storage and increased processing 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 is technique is commonly know as Forward Error Correction and is applied in several systems [this website marc sent earlier this year with the military FEC network] [find more]. The downside of FEC is that it comes 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 exacerbated as the retransmission requests use extra bandwidth, causing more data packets to be lost, leading to even more retransmission requests.

Having identified the requirements for live multicasts over packet-switched networks, we can understand why the largest publicly available network, the Internet, is less suited for this. Although IP natively recognises multicast packets by their class-D destination 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 [SAV96] 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 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 [why the MBone doesn't work]. Hence, native multicast routing is not widely available. 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.

Given the requirements for live multicast and the lack of support for them by the Internet, we think the problem of large-scale multicast is better dealt with in the application layer, rather than on the network layer. This leads us to the popular research area of application layer multicast where a virtual network is constructed from software based router nodes that 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 modest and incomplete list of solutions includes Scattercast [CHA00], Narada, TrafficCore, Overcast, etc, etc. The fact that new initiatives continue to emerge illustrates the difficulty of solving multicast on packet-switched networks in a generic way. In this text we will introduce another application layer multicast solution which we will call *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 for efficiently 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, means 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, 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 accept the random loss of one or two video frames, this type of behaviour can wreak havoc in financial data when the missed packet contains 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 sufficient capacity lead to the research area of 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, we will propose an enhanced form of layered multicast that guarantees complete delivery for certain layers to avoid random loss altogether, making it suitable for market data. We can isolate the two core activities of Terracast. The first is that of running and managing a robust and scalable overlay network that uses its own routing algorithms and supports multicast. The other core activity is that of managing flow control and congestion when live streams overload the network and ensuring layered multicast can be offered with guarantees.

Because Terracast combines two research subjects, it is difficult to find comparable, existing solutions. Therefore we will also divide the evaluation of Terracast in two and compare the part that manages the overlay network topology and multicast routing protocol to similar, existing overlay products, while comparing the flow control, bandwidth allocation scheme and layered multicast protocol to other existing layered multicast solutions from the recent literature.

5. Terracast Architecture

In this chapter we will describe in detail how the Terracast system works. However, before we discuss the inner workings of the various components, we shall give a high level overview of the system as a whole. In this overview we shall explain 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 supports single-sourced multicast. Although the section is quite lengthy, it does not explain the theory behind routing or packet discarding algorithms.

5.1. General Overview and Software Components

Being an overlay network, the Terracast network is essentially a number of software router daemons, interconnected by normal TCP connections. Together this forms an intricate web of routers and virtual links like the Internet itself, but different because it operates on the application layer. Each of the 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. 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 being 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 using the router's unique address. This mechanism is somewhat comparable to TIBCO's RendezVous product [ref], 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 to a logical IP port. 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 similarly in that only router daemons have a Terracast address, whereas user applications connected to a router daemon are identified by logical ports[1]. Illustration 1 shows how Terracast routers and client applications are connected. The diagram shows a Terracast overlay network of four router

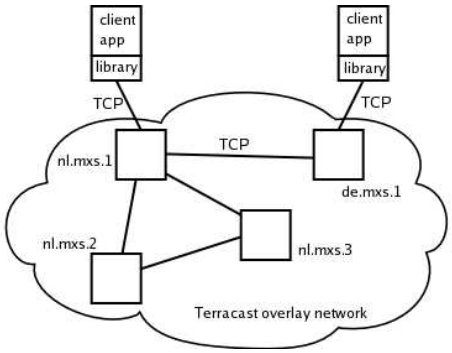

Illustration 1: A Terracast network is formed by software router daemons, interconnected by TCP connections.

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.

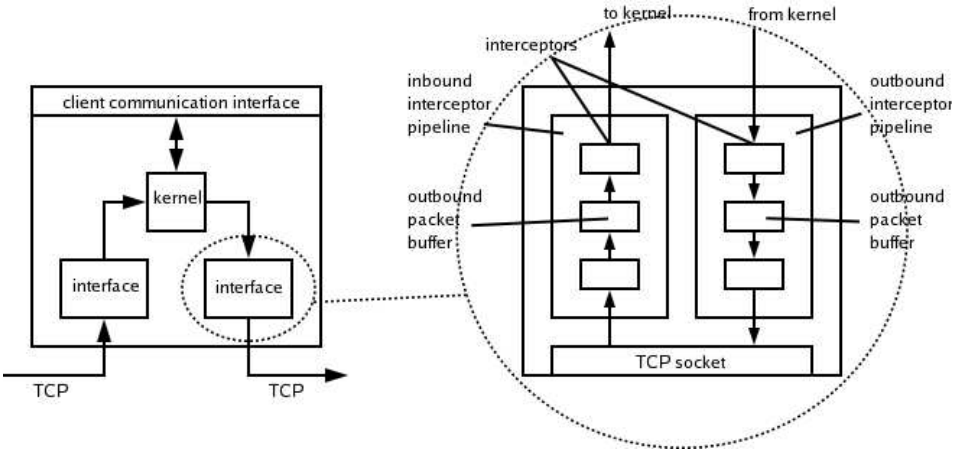

Illustration 2: A Terracast router daemon as shown on the left consists of a switching core or kernel and one or more interface modules that connect to neighbor routers. Using the unicast routing table, packets are forwarded from one interface to the other. The right hand side of the illustration shows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

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.

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. In principle 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 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 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 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 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, if the new information lead to changed routing entries, the router sends its own routing entry to all its neighbors. A similar thing happens when a 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. 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 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 shown that this means it needs storage capacity to temporarily buffer packets. The same issue exists in the inbound pipeline where a thread is continuously reading packets form 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 packets is done in a special interceptor. This interceptor 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 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 can only be dropped inside the

buffer interceptors of the interface modules. Therefore this interceptor plays a key role in making packet loss deterministic[2]. 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 interceptor that measures the actual outgoing traffic rate. When incoming bandwidth is to 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 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 allows clients to use their services. This mechanism of accessing the router's services was already mentioned in the previous chapter. When a user application wants to send messages over the Terracast network, it uses the router's client library. This library offers methods for sending and receiving packets. In the background it uses regular TCP sockets or native inter-process communication to proxy these method invocations to the 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. On the right we see 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 we will refer to a port in Terracast 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

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 an 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.

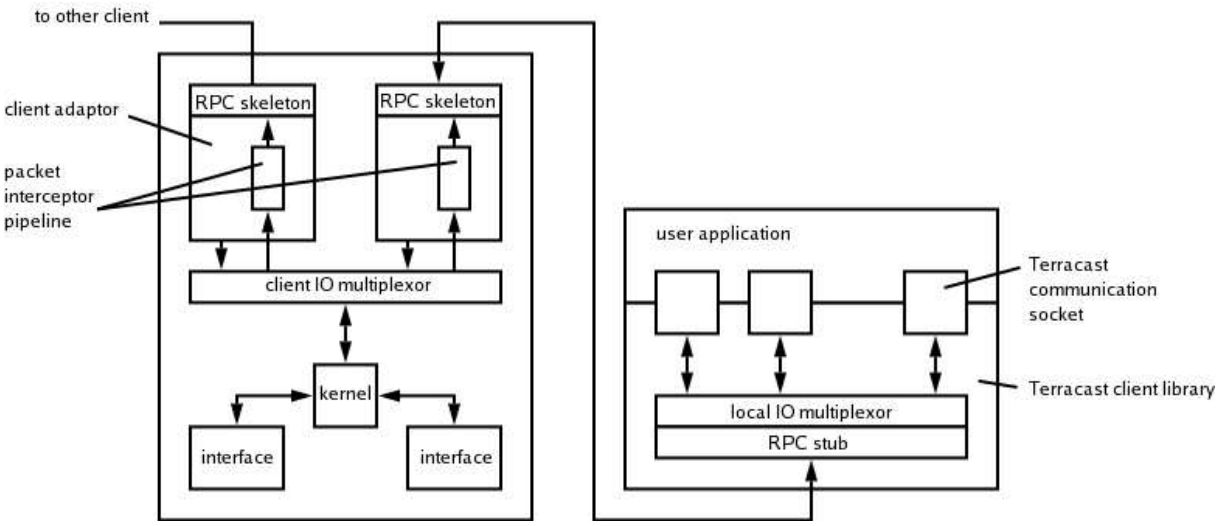

Illustration 3: User applications use the Terracast network through RPC using a client library. For each connected client, the router runs a client adaptor. When packets arrive addressed to a connected client, the router's client IO multiplexor passes it to the corresponding client adaptor where the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

for a router daemon is **`nl.ams.mxs.gw`**. The same rules apply to tport identifiers[3]. The motivation behind string-based addresses and the meaning of the dots is explained in chapter [x] 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 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 **`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 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

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.

existing multicast networks can offer multi sourced distribution, only few applications really require this functionality. In single-source multicast networks, the data stream is published by once 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 multi-sourced 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 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 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 it 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 never **`q:mygroup`**. If no other application connected to the same router has already bound the group earlier, the source application will succeed in binding and be ready to start its multicasting. 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 the virtual multicast distribution 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. In the current system it is not possible for subscribers to tell the difference between a multicast group that publishes no packets and a multicast group whose source has died. 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 the tsocket of a publish-mode group cannot automatically receive the data

packets it published. If the source 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 equal, each data packet carries a special flag that indicates 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 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 tport that was bound by the source application. It does not contain any other addressing information. The exact wire-level packet layout of the data packets can be found in appendix [X], together with the definition of all other packets used in the Terracast network.