Layering Financial Market Data

Erik van Zijst
January 20th, 2003

Vrije Universiteit,
Faculty of Sciences, Department of Mathematics and Computer Science,
Amsterdam, The Netherlands

erik@marketxs.com

1. Abstract

In this paper we address the problems related to the distribution of realtime, non-interactive data streams across heterogeneous, packet-switched networks that permanently or temporarily lack sufficient bandwidth to reliably transfer the whole data stream in realtime from sender to receiver. Examples of such realtime data streams include audio/video playback and realtime stock updates from a stock market. To prevent the receiver from lagging behind when the data stream temporarily or permanently exceeds the available network bandwidth, carefully chosen parts of the stream have to be discarded to ensure the timely delivery of the remaining packets. Although successfully applied to digital audio/video transmissions[1], this technique has so far not been used for realtime market data distribution. In this paper we will explain the realtime properties of market data and propose two algorithms for layering these streams, including an object model that offers a generic interface to financial applications.

2. Introduction

Using a heterogeneous, packet-switched computer network consisting of many individual links between sender and receiver to distribute a continuous stream of high-volume data only works well if the whole network has sufficient capacity at all times to deliver all of the data from the source to its destination. However, as soon as any part of the network gets congested, the sender will have to buffer pending data until the network becomes less congested again. This is similar to the way the reliable TCP protocol automatically reduces transfer rate to match the available bandwidth. Since network congestions do not necessarily disappear after a finite amount of time, this implies an infinite amount of storage capacity at the sending node. Aside from the storage problems, the data arriving at the destination gets more and more delayed, which often affects its relevance. Typical examples here are live audio, video or both. These data streams must always be played at realtime in order to make sense. We call these streams *isochronous* because the have both a minimum and maximum transfer rate. In order to not fall behind due to network congestion, the sending process must selectively discard certain packets and not wait for the network to process all previous packets, to ensure that the others arrive on time. A *layered data stream* can help the sending process decide which packets to drop without corrupting the content. In a layered stream, each layer provides additional, more detailed information about its parent. Together, all these layers provide content in the same resolution or detail as its original, but with one or more of the top layers missing, the remaining layers still contain a valid representation of the original content, but only in a lower resolution, frequency or bitrate. This way, quality is directly related to the available resources, which seems quite logical. We believe the same approach could be taken for financial market data distribution. In fact, it seems the only acceptable way to address network heterogeneity for realtime content distribution in general.

3. Network Model

The environment for the proposed layering technique is a computer network that consists of a mesh of interconnected point-to-point links. These links have individual properties and can fail independently. The performance of the network as a whole depends on the actual routing protocol that is being used, but may (and is likely to) vary over time. The environment as depicted above resembles the Internet and it is not unlikely the proposed layering architecture will even be deployed on the Internet. This network environment is opposed to the somewhat more traditional way of broadcasting realtime data through satellites. When using satellites to broadcast through a medium with a quality as homogeneous as the “ether”, there is usually little need for techniques to handle periodic changes in quality dynamically. This does imply a constant transfer rate however,

which might be acceptable for audio playback, but is less suitable for market data, that often comes in bursts and peaks. Unfortunately however, satellite content distribution comes with its own set of problems that include high latency, expensive bandwidth, special hardware at a client's location and no support for various QoS levels, such as retransmissions, as a result of its non-duplex nature. These issues are beyond the scope of this paper.

4. Desired Properties

Ideally, every individual client should be able to receive the realtime market data stream in the highest possible quality. If the network has no multi- or broadcast support, as is usually the case on wide area IP networks, every client will have a point-to-point connection to the sender. We want to be able to send a realtime stream of financial market data to each client individually in the highest possible quality, without having the sending process to compute specific streams with different quality for each connected client, analogue to the way a realtime audio/video stream is distributed in a RealAudio network. In order for this to work, the source will have to layer the data in as many layers as necessary, before multiplexing it to the connected clients. The sender will then be able to provide each client with the highest possible transfer rate, by uploading as many layers as the client's connection allows. In general, the stream can handle individual client resource problems more accurately as the data is split into more layers.

5. Object Model

To be able to offer financial applications a generic interface to the layering infrastructure and to be able to support different layering algorithms and implementations, we present an object model that holds the layering engines, interfaces with applications and has the ability to serve multiple client connections simultaneously, providing each client with the highest possible quality. The system works with objects that implement the `Layerable` interface. Each object is an update for a certain subject. In financial applications this subject is a security and the updates related to the subject are stock-quotes or individual trades. The key in a `Layerable` object indicates which subject an object is related to. The core of the system is an instance of the `ObjectLayerer` interface. This class implements a certain layering algorithm and may be configured to support one or more object layers.

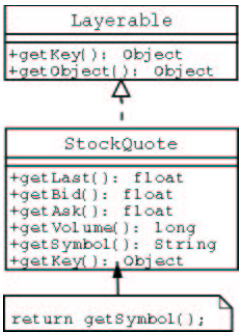

Illustration 1: The system works on objects that are layerable.

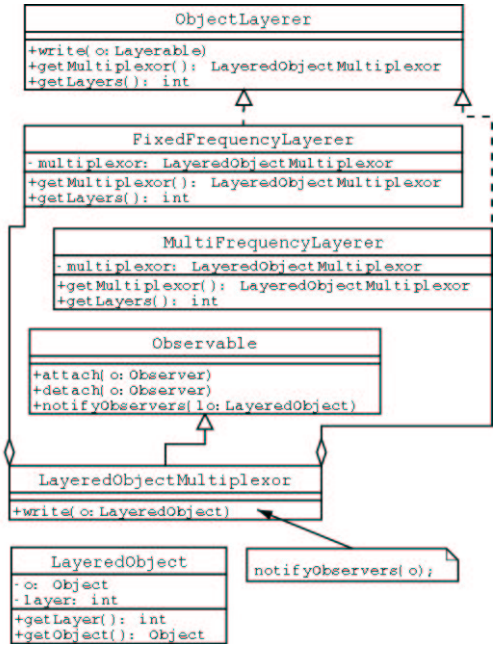

Illustration 2: The way objects are layered is defined by the implementation of the ObjectLayerer class. Distribution to the client connections is handled by the observer pattern.

An application that generates processes stock-quote updates, passes these updates to the layerer class as layerable objects. Output and distribution of the layered object stream that is generated by the layerer is supported by an instance of the

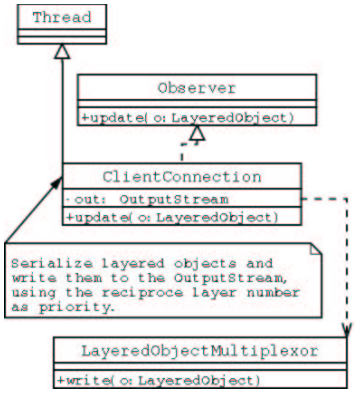

Illustration 3: A connection object receives the layered objects and sends them to the connected client.

`LayeredObjectMultiplexor` class that is implemented by the layerer. Distribution of the stream to multiple client connections is done by applying the Observer/Observable design pattern[2]. The multiplexor effectively is an observable object that can have multiple observers registered that receive a copy of the layered stream. Every time the layerer publishes a new layered object, the multiplexor invokes the update method of all observers, passing the new object along with it. A typical observer is an object that manages the socket of a specific remote client. It buffers a small number of updates and transmits them in serialized form over the network. Inside the buffer, the objects are ordered according to their layer number and age. A lower layer will always get priority over a higher layer. These objects typically have their own thread of control to handle the network communication. In the next two paragraphs, we will present two algorithms that can implement the ObjectLayerer interface. The first algorithm, Fixed Update Frequency, turns the stream of updates from the application into a single layer stream that has a fixed maximum frequency. When the application delivers more updates per second, the algorithm will discard them, while ensuring a maximum delay for each subject, regardless of the properties of the original stream. The second algorithm, Incremental Multi-Frequency, extends the first algorithm to create a true multi-layer stream in which each layer provides additional, more detailed information about its parent.

6. Fixed Update Frequency Algorithm

We think the fundamental issue in deciding whether or not a certain stock-quote update is important enough to forward, or just to drop it, is related to the volatility of the stock at hand. When there is not enough bandwidth available to forward all stock-quotes, the updates for the most volatile stocks should be dropped in favor of the less volatile ones. We propose a best-effort engine that can trim down a continuous high-volume stream of quote updates to a fixed bandwidth, without losing important updates for stocks that get only very few trades. The algorithm works by putting the individual stocks in a circular, linked list. Every item of the list represents the latest update for a certain stock. Every time a new update comes available, the current one is overwritten in memory. This way, the linked list always contains the latest trade of every stock. Next thing we do is to have a virtual token cycle the list at a fixed speed. If we want to shape the original stream down to one update per second, we will have the token visit one list item per second. When the token visits an item, it removes the quote update and transmits it, leaving an empty list item. Empty items will be skipped without delay. When one of the stocks has two updates per second in the original stream, the second update after the last visit of the token will overwrite the quote already in the list. This way the engine will always try to provide the client with the most recent update of every stock.

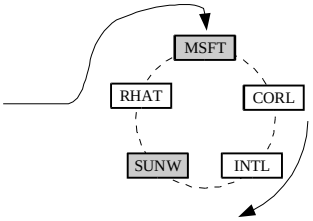

Illustration 4: The Fixed Update Frequency algorithm at work. The gray list items represent items with a recent quote update, while the transparent items are empty.

How the algorithm works is illustrated above. While the virtual token visits the items that have a recent quote update waiting to be sent out, the arrow on the left represents a thread that inserts new trades when the application calls the write method of the layerer. To sustain the maximum throughput rate of one quote per second, the token thread sleeps for one second after it sent and removed an update from the list. When sufficient bandwidth capacity is guaranteed, the token could run without any delay. If in that case the token thread is capable of completing at least one full circle between every incoming trade, there will be no data loss, nor any delay and the stream will be forwarded in the original order. Using the engine as illustrated, unfortunately introduces two issues. The first is that when the interval between two incoming updates is shorter than the rotation speed of the token, more than one updates are stored in the circular list at a time, causing the order of the updates in the output stream to reflect the fixed order of the items in the linked list. This is not necessarily equal to the input order. In case of independent stock-quotes that might not be an important issue though. The second, more severe issue is that of additional delay, as a result of the fixed period of time the token waits after every removed update. When an incoming update is inserted in the list, it will have to wait for the rotating token to reach it, before it gets processed. The time it takes for the token to reach the update depends on the number of waiting updates between the new update and the token and the configured update frequency. In the worst case scenario this equals the total list

items divided by the update frequency:

$$t_{max} = \frac{size(S)}{f_{update}}$$

Where t_{max} is the maximum delay in seconds that the algorithm could introduce, f_{update} the update frequency in updates per second and $size(S)$ the total number of elements in the circular list.

7. Incremental Multi-Frequency Streams

While the Fixed Update Frequency algorithm provides a good way of trimming a dynamic, potentially high-volume stream of quote updates to a fixed maximum rate, we need a more advanced algorithm to produce several of these fixed-rate streams that can be combined to one multi-layer market data stream.
A straightforward way of going about this problem might be to have several copies of the algorithm running in parallel, each at a different rate, tagging their outgoing objects with their unique number and combining all these packets into one big outgoing stream. Although the individual `ClientConnection` objects can successfully drop the most dense layers in case of limited bandwidth, it is not a very efficient way of tackling the problem since the individual layers contain a lot of unnecessary, redundant information. A more elegant way would be to have each layer contain only information additional to the previous one, much like the aforementioned audio/video schemes. In order to achieve this, we extend the Fixed Update Frequency algorithm to a multi-level circle. A virtual cylinder of vertically stacked circles with one circle for each layer of the output stream. Each circle will have its own token traversing the list at a fixed rate. A visualization of the virtual cylinder is depicted below in illustration 2.

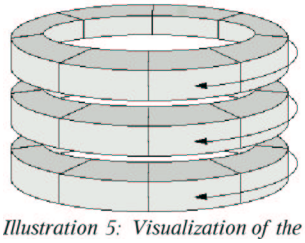

Illustration 5: Visualization of the individual circular linked lists representing the stream layers.

Every column, consisting of individual list items stacked on top of each other, can hold a number of quote updates for a certain stock, equal to the number of layers in the output stream. New stock-quotes are inserted in the lowest circle. When a second update for the same stock arrives before the token could remove the previous, the old update is pushed to the next layer while the new update takes its place in the lower ring. Once an update is pushed up, it can never return to a lower layer, even if all other layers are empty.

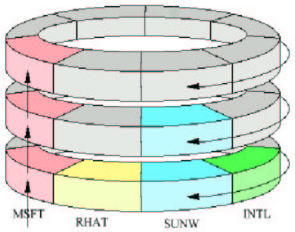

Illustration 6: The Incremental Multi-Frequency model. Colored items represent quote updates. Illustrated is how volatile stocks are spread across the layers.

Securities that have more than one trade during the time the token needs to complete one full rotation will be spread out over two or more layers. The more layers the network can distribute, the higher the update frequency for the security at the destination. Regardless of the total number of layers and their rotation speed, it is usually a good idea to let the token in the upper layer rotate without delay. This way the source itself will never introduce any data loss and every client can receive each stock-quote, provided there is enough bandwidth available.
Since the Incremental Multi-Frequency algorithm is based around the Fixed Update Frequency algorithm, it suffers from the same issues, namely out-of-order delivery and additional delays. Since the all of the rings can contain updates for the same stock at a certain time and since they rotate independently, it is unknown which ring will output its update for the specific stock first, who will be second and so on. This means that not only the original order of the independent updates are shuffled, but even the updates for a single stock are no longer guaranteed to arrive chronologically. While more recent updates come and go, an unfortunate quote update could loiter around in the system for quite some time. There could be a situation where an update that is just about to be removed by the rotating token, gets moved to a higher layer by a more recent update that entered the rings from below. In this example there is a big chance the new, most recent update gets removed by the token of its ring sooner than the

older update that has to wait for his token to make its way around the ring of his layer.
To address the out-of-order delivery, the system needs to provide a way of putting the outgoing, layered, updates back in the original, chronologic order. While there might be other ways of tackling this problem, we suggest to label incoming objects with sequential numbers, while having the tokens pass their updates to a transmit window that uses the sequence numbers to reconstruct the original order. If updates are forced out of the upper layer (and are therefore considered lost), the transmit window is notified not to wait for their sequence numbers. When applying order reconstruction to all updates, the transmit window must have a size equal to all list items of all layers together:

$$size(sw) = \sum_{n=1}^{j} size(n)$$

Where $size(sw)$ is the maximum size of the transmit window and $\sum_{n=1}^{j} size(n)$ is the sum of all list items in all layers from 1 to *j*, where *j* represents the upper layer. The potential size of the transmit window and the bursty behavior it adds to the output stream might justify a solution that applies order reconstruction to only achieve chronologic delivery of updates for the same stock.
Just like the Fixed Update Frequency algorithm, the Incremental Multi-Frequency algorithm can also introduce additional delay to quote updates. Since this model is essentially a collection of Fixed Update Frequency rings, its worst case delay is the sum of each ring's worst case delay:

$$t_{max} = \sum_{n=1}^{j} \frac{size(n)}{f_{update_n}}$$

Where t_{max} is the maximum delay in seconds that the algorithm could introduce for a single quote update and $\sum_{n=1}^{j} \frac{size(n)}{f_{update_n}}$ is the sum of each ring's maximum delay, defined as the list size $size(n)$, divided by the update frequency f_{update} of each ring *n* to *j* (*n = 1* being the lowest ring, *j* the highest).

8. Open Issues

Although an actual implementation has so far only been realized for the framework described in paragraph 5 with a layerer class implementing the Fixed Update Frequency algorithm, we might already be able to identify some potential bottlenecks and problematic issues for the Incremental Multi-Frequency algorithm. Since the architecture's overhead grows linearly with both the number of layers and the number of individual stocks, layering the data stream for a complete stock exchange with thousands of stocks might prove a challenging undertaking. Also, when using separate threads to cycle the rings independently, one might end up with complicated synchronization issues. Individual token-threads might be necessary when the rings have different rotation speeds.

9. Conclusions

In this paper we proposed techniques to layer a potentially high-volume, realtime market data stream incrementally, to make it suitable for one-to-many distribution over a heterogeneous computer network such as the Internet. Where realtime broadcasts cannot be delayed to address performance drops in the network, layered streams provide a way to drop parts of the data in an intelligent way, decreasing the resolution of the data, rather than corrupting the original stream.
We proposed a framework with two possible layering algorithms that is placed between a financial application that generates a stream of stock-quote updates and the network connections that connect the application to its remote clients. Of the two proposed algorithms, the first, Fixed Update Frequency, is limited to generating a single-layer, fixed rate stream, while the second, Incremental Multi-Frequency, is a full featured algorithm, capable of transforming a raw, monolithic object stream into a multi-layered stream that can dynamically adapt to the individual, changing network performance needs of the clients. Both algorithms take a best-effort approach to make sure all clients are always at least provided with recent updates for both volatile and less-volatile stocks.
Although there are a number of potential problems with the Incremental Multi-Frequency algorithm and we lack an implementation to test with, we have no reason to believe the algorithm would fail if deployed. In fact, we think the proposed solution is a good way of tackling the financial layering problem and we are not aware of any similar initiatives.

1 RTSP, Real Time Streaming Protocol, http://www.rtsp.org/2001/faq.html

2 Gamma, E., et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995