Layering Financial Market Data

Erik van Zijst
MarketXS
August 13th 2002
erik@marketxs.com

Abstract

Disseminating high-volume financial market data in a professional manner requires more than merely a lot of bandwidth. It requires one to be able to guarantee a certain quality of service (QoS) level that cannot be obtained by simply investing in network resources.
When broadcasting large quantities of continuous market events over a network to a large number of simultaneous, geographically-spread customers, one must be able to act upon a diverse number of problems that can occur within the network or at client locations in order to maintain quality of service.

Introduction

Using a heterogeneous computer network consisting of many individual links to distribute continuous streams 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 all destinations.
The typical example for high-volume data distribution over heterogeneous, packet switching networks is multimedia content such as live audio or video broadcasts over the Internet.
However, as soon as the network gets congested between the source and the destinations, the routers along the way will either have to store all incoming data from the source until the network becomes less congested again, or data will simply have to be dropped. Since network congestions do not necessarily disappear after a finite amount of time, this implies an infinite amount of storage capacity at the routers. Aside from the storage problems, the data arriving at the destinations gets more and more delayed, which often affects its relevance.
In case of a temporary drop in network performance during a continuous video broadcast, the best thing to do is probably to drop packets along the way in such a way that the resulting packets that do arrive at the destination still contain a realtime video stream, albeit in degraded mode. This approach is not new and is already widely taken by many multimedia protocols, including RTP/RTSP[i]. This technique is referred to as *data layering* where every data layer provides incremental information to take the data gathered by the previous layers to a higher resolution. 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.
In this paper we propose a technique to layer financial data, specifically stockquote updates, in a way that packet loss, introduced by network elements to tailor the stream to the available resources, results in a lower quality market data feed, without missing important, less frequent quote updates that typically tend to get lost when dropping packets (or quote updates for that matter) randomly.

Network Model

The environment for the proposed layering technique is a computer network that consists of a grid 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 true broadcasting through the use of 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.
Unfortunately however, satellite content distribution comes with its own set of problems that include high latency, expensive bandwidth, special hardware at the client locations and no support for various QoS levels as a result of a non-duplex upstream. These issues are beyond the scope of this paper.

Desired Properties

Ideally, every individual client should be able to receive the realtime data stream in the highest possible quality, without putting any additional stress on the source to format the data stream to all these specific properties.

The network elements themselves must be able to identify the layers within the stream and drop as many layers as necessary to trim the stream to the bandwidth available on each of its (physical or virtual) links.
In order for this to work, the source will have to layer the data in as many layers as necessary before transmitting it to the network. In general, the stream can handle individual client resource problems more accurately as the data is split into more layers.

Fixed Update Frequency

We think the fundamental issue in deciding whether or not a certain 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 stockquotes, 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 traverse 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 forwards 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.

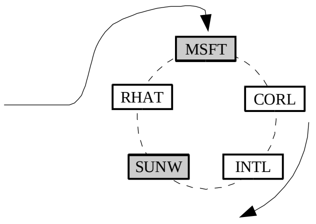

Illustration 1: The Fixed Update Frequency algorithm at work. The gray list items represent items with a recent quote update, while the transparent items are empty.

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 as they arrive from the original stream.
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 be equal to the fixed order of the items in the linked list. This is not necessarily equal to the input order. In case of independent stockquotes 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 transmitted 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 transmitted. 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.

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 packets with their unique number and combining all these packets into one big outgoing stream. Although routers 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 RTSP[i] protocol suite.
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. The higher the circle, the faster the token moves. A visualization of the virtual cylinder is depicted below in illustration 2.

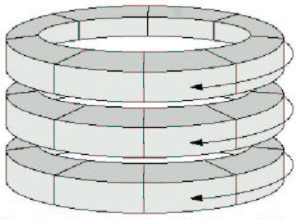
Illustration 2: Visualization of the individual circular linked lists representing the stream layers.

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 stockquotes are inserted in the lowest (and slowest) 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.

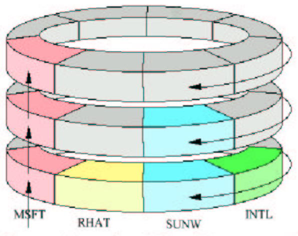

Illustration 3: The Incremental Multi-Frequency model. Colored items represent quote updates. Illustrated is how volatile stocks are spread across the layers.

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 individual update frequency, 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 packet loss and every client can receive each stockquote, 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 operate independently on their own speed, 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 updates 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).

Open Issues

Although an actual implementation has so far only been realized for 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 threads to cycle the rings independently as suggested, one might end up with complicated synchronization issues.

Conclusions

In this paper we proposed a technique to layer a 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 along the way intelligently, decreasing the resolution of the data, without corrupting the original stream.
We proposed two possible layering algorithms, specially designed to deal with financial market data: Fixed Update Frequency and Incremental Multi-Frequency. While the first is limited to generating a single-layer, fixed rate stream, the second is a full featured algorithm, capable of transforming a raw, monolithic data stream into a multi-layered stream that can dynamically adapt to multiple, simultaneous performance changes in the network. 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.

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