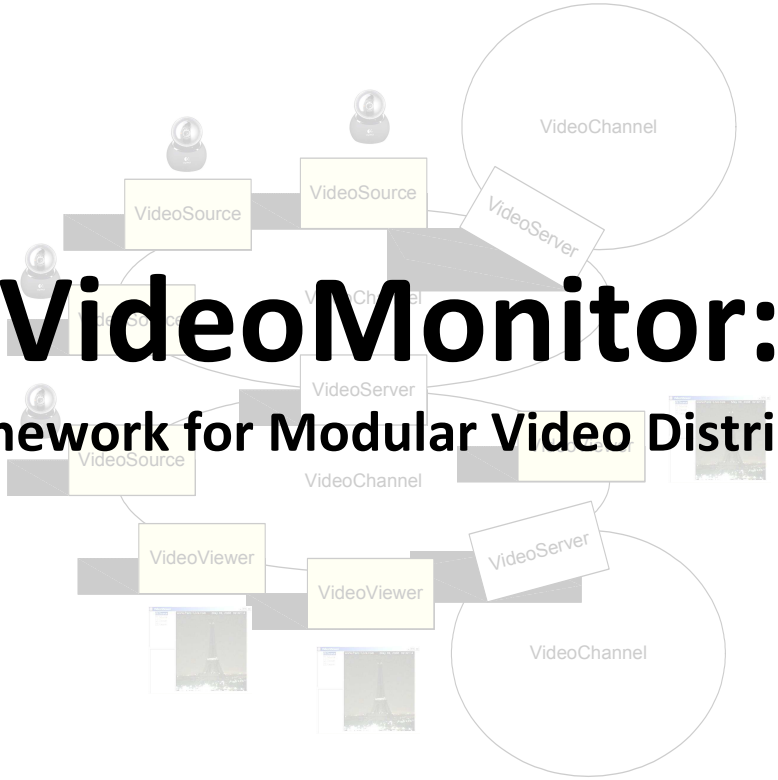


VideoMonitor:

A Framework for Modular Video Distribution

CS514 Assignment 3
Dan Gicklhorn (dpg62)
Phelps Williams (pww36)

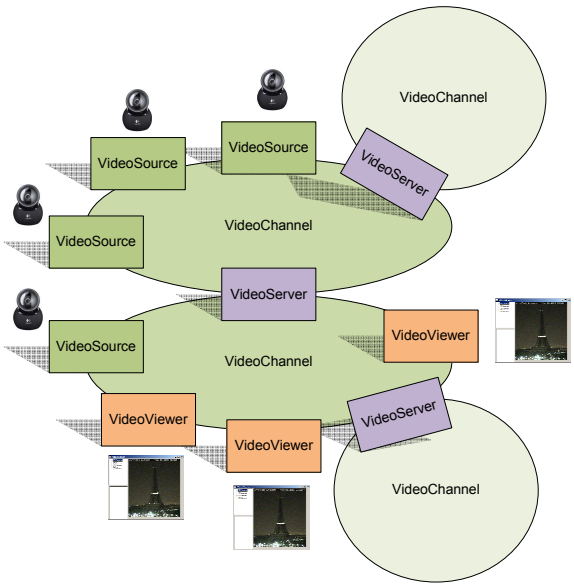
VideoMonitor:
A Framework for Modular Video Distribution

Overview of Features

- Highly modular service based system design
  Modular 3 component system to allow large, complex network topologies
- Self-Healing network model, robust to failures
  Network maintains a synchronized model and constantly performs integrity checks to test for failures as well as providing an dynamic active service list to components interfacing with the channel
- Componentized application model to allow for quick integration with the framework
  Component based LiveObjects modules to allow for easy interface with the network management object

Motivating Idea

The VideoMonitor system is designed around the idea of having a diverse collection of communication channels, on which many types of useful services can live. A generic packet format, dynamic addressing and network management, along with a structured service model can allow for this sort of amorphous collection-based network to run and provide immediate access to services deployed on it. This sort of network can be made to handle any sort of data, but our motivation comes from streaming video, for which there is no easily available support today and is an excellent service model to explore.



Design Overview

The Video Monitor system based on Liveobjects allows for multiple video sources to be dynamically viewed as best fits the user application. The system consists of three primary components, a Viewer, a Server, and a Source object. Multiple servers are interconnected with any number of Sources providing a video stream. The servers maintain a hierarchy of services, an index to the attached sources and servers currently active. A viewer component can then connect to a server and receive this index from which sources of interest can be selected. The registered viewer service on the server maintains a list of sources for each viewer, the stream from that selection of sources is then pushed to that viewer. Additionally, some sources are capable of connecting to streams with additional functionality such as resolution settings, pan-zoom-tilt control, etc. These additional features are classified as subservices and can be utilized by the server or viewer.

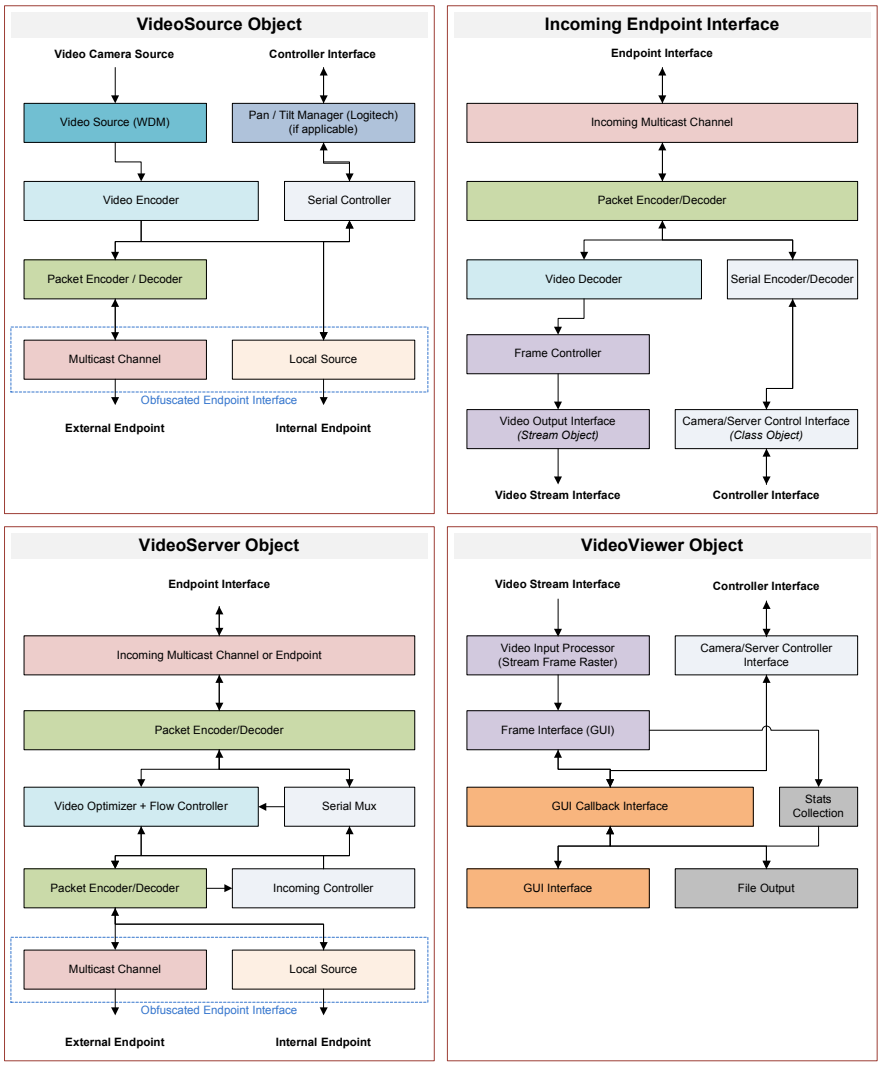


Figure 1: High-Level Component Design

The component roles can be classified as follows:

VideoSource

- Interface to an array of video sources (camera, internet based webcam, any dynamic image source, etc)
- Push video stream to connected server

Figure 2 Streaming Video Source

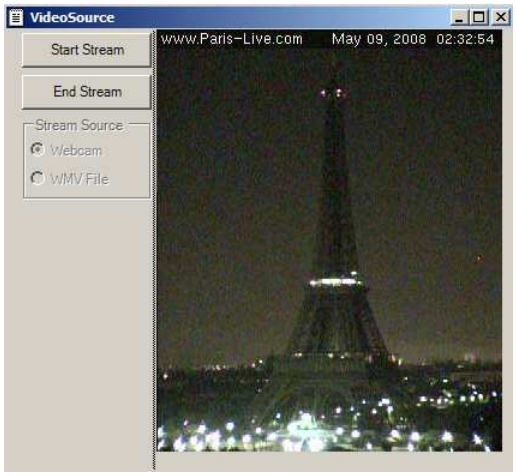


Figure 3 Streaming Webcam Source

VideoServer:

- Service Indexer
- Source Stream Distributor
- Additional features such as frame rate throttling

VideoViewer

- Interface which allows the user to select desired services
- Control which allows user to modify settings on the Server or Source services

Figure 4 Streaming Video Viewer

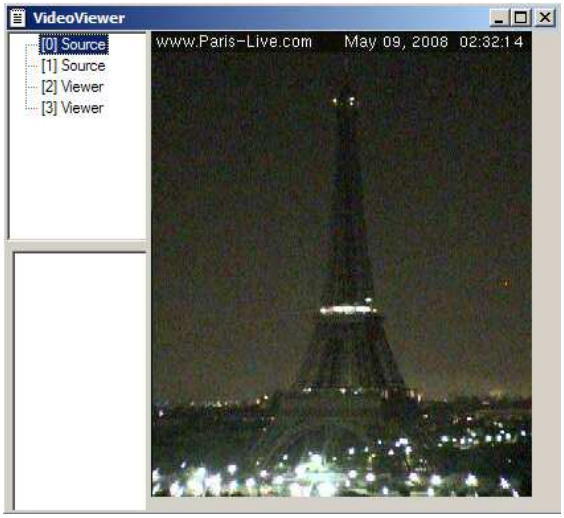


Figure 5 Streaming Webcam Viewer

VideoSource Overview

The VideoSource takes an arbitrary video stream such as a webcam or any image source off the internet samples the source and formats the sample as a serialized bitmap. The serialized bitmap is then pushed to the server. Other formats were also envisioned such as an MPEG encoder / decoder. The IP TV suggestion from Professor Birman was quite interesting although starting with a straightforward example such as a series of serialized bitmaps would be a good place to start. Protocol constraints could more easily be classified and corrected resulting in a more easily implemented system with a base functionality set.

The VideoSource also provides an interface to special functionality provided by the actual media source. For example, cameras such as the Logitech QuickCam Orbit MP provide a motorized pan-zoom-tilt control. Drivers for this functionality are provided through Logitech. Additional functionality such as face tracking is also accessible. The VideoSource classifies itself indicating these types of functionality are available. Alternate camera systems such as Axis Web Cams provide similar functionality though the control interface on the camera is different. The VideoSource abstracts these differences away.

The current set of functionality includes support for serialized bitmaps from either a usb based webcam or a url accessible source. Many open webcams were easily located and serve as useful example and test input for the system. The user interface is extremely simple providing a preview of what is sent from the source as well as essentially an on / off control to start or stop streaming. The VideoSource connects to the source channel on startup, the details of this process are covered in the architecture section below.

VideoServer Overview

The VideoServer serves as a service indexer and stream distributor. Viewer, Source and other

Servers are registered allowing for information routing via other servers or directly to endpoints such as the viewer or source. Each source is responsible for pushing frames to the source channel; these frames are captured by the server, addressed and retransmitted on the viewer channel to the viewers registered for a particular stream.

The VideoServer handles abstract frame and command objects, passing the information as necessary between channels. Modifications to command types, video streaming approach etc do not impact how the server operates.

VideoViewer Overview

The VideoViewer provides the user a list of available services determined through the connected VideoServer. The user is able to view several of these streams at once currently the number is four although this is easily changed. Viewing a stream is as simple as dragging the indexed Source to a viewing window. The Viewer indicates to the Server that it is now interested in a particular source and frames pushed from that source will now be additionally addressed to the new viewer.

The VideoViewer also provides a custom user interface for a variety of service types. An interface for pan-zoom-tilt control appears for sources providing this functionality. Frame throttling can either be auto-calculated based on packet loss rates or manually controlled. Currently the VideoViewer is functional minus the ability to issue commands.

Network Architecture

Features

- Self-healing design to account for failures
- Robust accounting for currently available services over the channel
- Low-overhead in message passing due to relaxed near-real-time requirement of network modeling

To effectively facilitate the transport of messages and maintain the network stability, a self-healing network model is needed. This model describes the interaction between “nodes” or LiveObjects objects interacting with the network. To make use of the LiveObjects componentized model as well as to provide obfuscated function to objects that wish to interact with the network, the main networking component is encapsulated into a StreamProcessor class which describes the entire interaction between Nodes. This Stream Processor class provides exported functionality to an application via an interface endpoint.

Design

The self-healing network designed for this implementation takes the form of a distributed, ring-monitored multi-cast channel. This design is used to both provide a comprehensive, near-live list of

available objects connecting to the channel, maintained with an ordered check system for node failures. The current network modeled locally in each node and ordered by node numbers. Any nodes joining the network will be given this network model object and then broadcast their local services to be added to everyone's model individually and resorted by node id. This list has 2 key values: root and tail. Root represents the node with the lowest numerical value (added longest ago), and tail, the node with the highest numerical value. The network is designed to provide the following aggregated membership functions:

Join When a new node joins the network, it sends out a broadcasted message reporting such and requesting a copy of the network model. Over all the nodes, as current network model is synchronized everywhere, the tail node (highest numerical number) responds with a copy of the network model. This new node is provisionally added to the tail node's network model, thus blocking it from responding to any new requests. The new node receives the model and sets his ID to the tail node's id + 1.

The new node now sets an internal reference to the tail node as it's "parent" node, and then transmits an ***expose*** message to the network with its service information so it can be replicated everywhere.

If a new node receives no response in request to asking for the network model, this implies one of two things. Either the node is the root to the network; in which case it will max out its number of times it sends the request message, or one has just added and will be available soon; and thus a subsequent request should be responded to.

Leave If a node is gracious enough to report leaving the network, it simply reports itself dead and will no-longer be considered for network duties.

Expose To expose a service (node) to the network allows the network to know about it. This can also be used to update the available services on a node or to replace a temporary listing in its parent. A node broadcasts a service object that represents its local services to the channel and is integrated into every node's network model. Once this is completed by a new node, it is fully initialized and ready to perform network functions. Upon any new (non-update) expose messages to the network, the root node assigns its parent reference to the new node, thus completing the ring structure of the network.

Test-Live This is a two-way request between a node and its parent. On a pre-set interval, a node sends this message to its parent, asking it to respond to make sure it still exists on the network. If the message times out, it is resent a specified number of times, upon exhausting which, the parent is assumed dead. The node then transmits a ***report-dead*** message to the network indicating that this node has gone offline and should be ignored.

Report-Dead If a node is reported dead, a message is broadcasted to the channel, and each node removes it from its network model. If a node is receives a message indicating itself has been reported dead, it introduces itself to the network the same way a new node would again, only changing its id if it believed itself to be the tail element before. When a node is reported dead, each node then re-checks its parental associations. If a node finds it has become the root node, it reorganizes accordingly and begins taking responsibility for the root's actions.

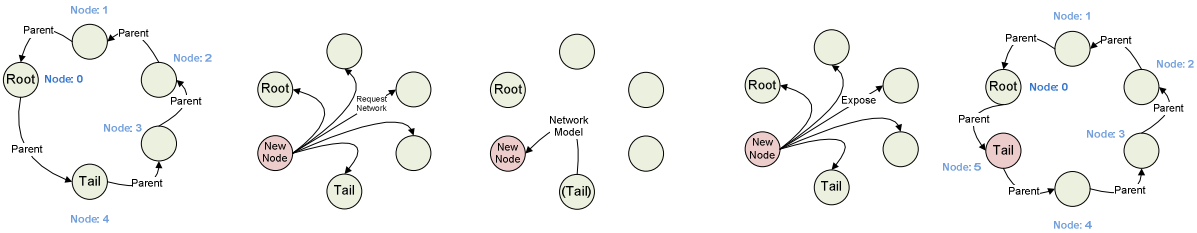
Checkup This consensus functionality begins with the root broadcasting a copy of it network model to the channel. Upon receiving a checkup message, each node responds with any discrepancies it finds between the root's network and its own, only sending services that exist locally but not in the root. Any services existing in the root but not locally are added locally. The root, upon receiving discrepancies, adds them to its own list and rebroadcasts the checkup message. This should settle quickly to any major discrepancies. This consensus may possibly add back dead nodes to the network but will include any live ones known by any node. Thus after local test-live functions are performed on each node to their parent; any dead nodes will be once again weeded out.

This model maintains a closed loop of node-parent relationships and monitors for failures therein. A near-real-time model of the current network is maintained at all times and can be made available to an application for service discovery and addressing.

Network Visualization

The following diagrams show the maintenance and procedure of the network model in adding new services and detecting failures.

The following describes the process of a node being added to the network.



Network Model(Global)

Network Layout	Parent (local assumed)
Node 0: Service	4
Node 1: Service	0
Node 2: Service	1
Node 3: Service	2
Node 4: Service	3

Global Assumptions
Root: Node 0
Tail: Node 4

Initially the network is setup with a complete ring of pointers. Root and Tail nodes are inferred from the header and footer rows of the network layout table. Parent pointers simply are inferred from the node above/below the node's entry in the table (or footer for root).

Network Model (Global)

Network Layout	Parent (local assumed)
Node 0: Service	4
Node 1: Service	0
Node 2: Service	1
Node 3: Service	2
Node 4: Service	3

Global Assumptions
Root: Node 0
Tail: Node 4 (Responds)

A new node looking to join the channel broadcasts a network request message to the channel. All nodes ignore other than the Tail node.

Network Model (Tail)

Network Layout	Parent (local assumed)
Node 0: Service	4
Node 1: Service	0
Node 2: Service	1
Node 3: Service	2
Node 4: Service	3
Node 5: Service (Provisional)	

Global Assumptions
Root: Node 0
Tail: Node 5 (Changed Locally)

The Tail node responds with it's network model object. It also adds a provisional entry for the node into it's network layout after it sends it to the new node. This entry will prevent the current Tail from responding to new requests while this new node adds itself to the network which will become the new tail.

Network Model (Global)

Network Layout	Parent (local assumed)
Node 0: Service	4
Node 1: Service	0
Node 2: Service	1
Node 3: Service	2
Node 4: Service	3
Node 5: Service (Upd. Globally)	4

Global Assumptions
Root: Node 0
Tail: Node 5 (Updated Globally)

The new node broadcasts to the channel an expose packet which makes the entire network aware of its services. This causes all nodes to update their tables to reflect this new node. This also implicitly updates it to the new tail node.

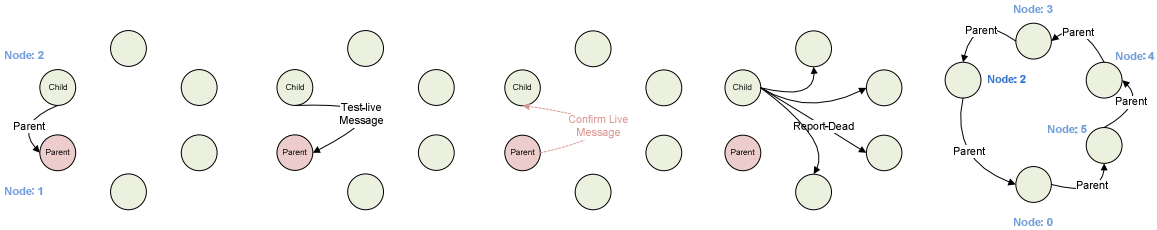
Network Model (Global)

Network Layout	Parent (local assumed)
Node 0: Service	5
Node 1: Service	0
Node 2: Service	1
Node 3: Service	2
Node 4: Service	3
Node 5: Service	4

Global Assumptions
Root: Node 0
Tail: Node 5

The new node is incorporated into the network and takes the place of the new tail element. This also implicitly associates it as the parent of the Root node, which will send a request-live check to it on it's next attempt. Any previously blocked requests now can receive a network model from it.

Likewise, detecting failures in the network can be described below.



Network Model (Global)

Network Layout	Parent (local assumed)
Node 0: Service	5
Node 1: Service	0 (FAIL)
Node 2: Service	1
Node 3: Service	2
Node 4: Service	3
Node 5: Service	4

Global Assumptions
Root: Node 0
Tail: Node 5

Initially before the check begins, the parent is assumed to be alive, no other check is performed prior to detection, thus the parent exists in the network model and is checked accordingly.

Network Model (Child)

Network Layout	Parent (local assumed)
Node 0: Service	5
Node 1: Service	0 (FAIL)
Node 2: Service >	1 (CHECK)
Node 3: Service	2
Node 4: Service	3
Node 5: Service	4

Global Assumptions
Root: Node 0
Tail: Node 5

A check message is sent by the node to its parent and is repeated a specified number of times on delay.

Network Model (Child)

Network Layout	Parent (local assumed)
Node 0: Service	5
Node 1: Service	0 (FAIL)
Node 2: Service >	0 (UPDATE)
Node 3: Service	2
Node 4: Service	3
Node 5: Service	4

Global Assumptions
Root: Node 0
Tail: Node 5

If no reply received after maximum number of attempts, the child believes his parent to be dead, and thus adjusts his parent accordingly.

Network Model (Global)

Network Layout	Parent (local assumed)
Node 0: Service	5
Node 1: Service >	0 (FAIL RCV)
Node 2: Service	1
Node 3: Service	2
Node 4: Service	3
Node 5: Service	4

Global Assumptions
Root: Node 0
Tail: Node 5

Thus the node reports his parent as dead to the channel to allow all of the other nodes to update.

Network Model(Global)

Network Layout	Parent (local assumed)
Node 0: Service	5
Node 2: Service	0
Node 3: Service	2
Node 4: Service	3
Node 5: Service	4

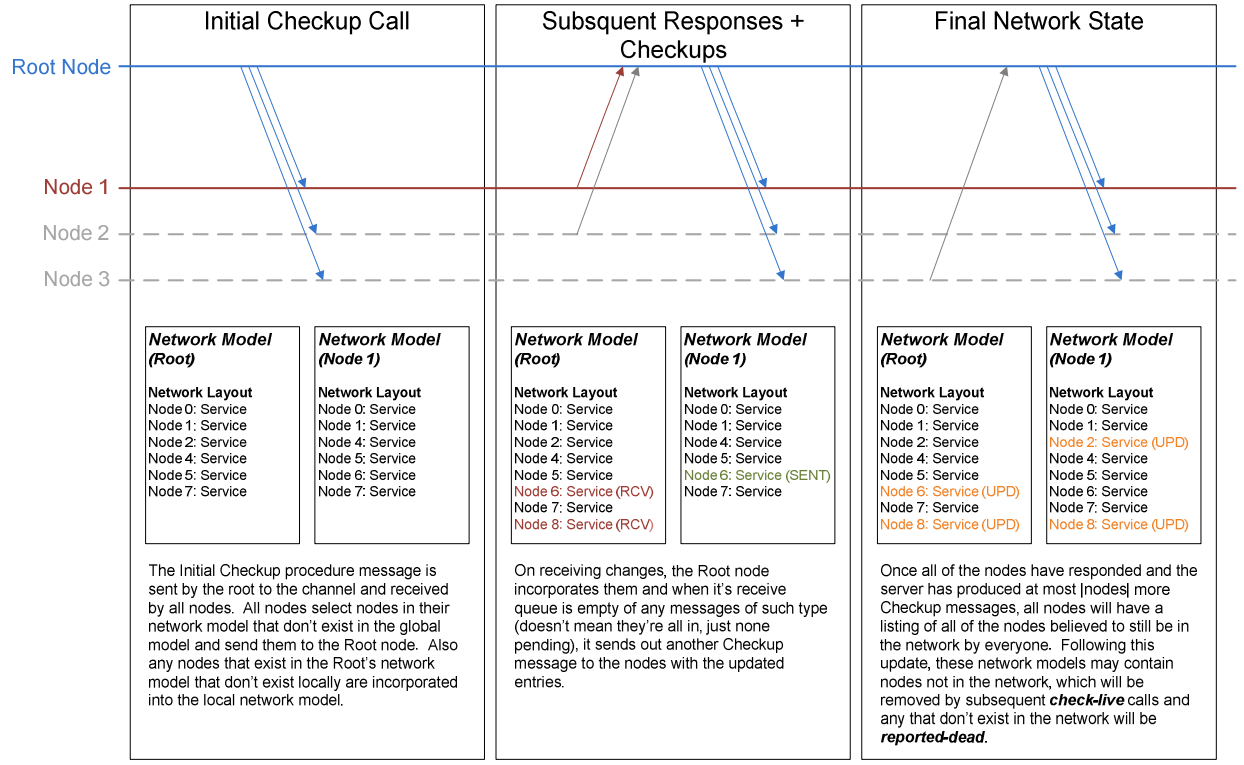
Global Assumptions
Root: Node 0
Tail: Node 4

Once the node is reported dead, the network now simply functions as before w/o the element.

In the case the node was deleted incorrectly, it will attempt to rejoin once it receives the report-dead message.

In the case that the root node is deleted, the node holding it as it's parent will assume the root node duties.

And the process of checking for errors in the network model is shown here.



Implementation Challenges

The completed design shares most of the elements presented above although several challenges were faced which forced modification of the initial objective. Initially we planned on using QSM which was presented to not provide ordering guarantees but would provide improved packet delivery times. As presented later, this would have been a nice addition based on the measured performance metrics. Because of configuration and general setup difficulties during development, it was decided to use the TCP based approach used in Assignment 2. We limited our design to not take advantage of the checkpoint capability and ordering guarantees however. This would simulate the limitations of QSM resulting in a more interesting design challenge. While this wasn't a significant issue for video frame transfer, this required additional consideration and implementation of our own state transfer approach described above. This was particularly relevant when a new node joined the network and address assignment had to be handled.

The Server object was dropped from the critical development path after it was decided that multiple channels would likely be difficult to use based on newsgroup postings. As a result the role of the server was trivialized. Initially the server was envisioned as a crossover between the two channels, acting as a filter reducing the network load particularly on the Viewer side. With a server object hierarchies of servers were envisioned potentially with additional unique channels between them and possibly supporting other network protocols allowing for Video Monitor bridges between distant locations. The servers would maintain a replicated representation of all nodes currently connected and

would provide this to the clients (Viewers and Sources) connected to them.

Webcam support (ie USB based cameras) was also an initial objective. It turns out that interfacing with these devices is not nearly as simple in C# as in many other languages familiar to the authors. Instead a remarkably convoluted and unsurprisingly difficult to work with wrapper to unmanaged C++ calls was required to interface with DirectX. After configuration difficulties and unreliable performance on new machines this functionality was dropped. Instead web based webcams accessible via a URL and updated at a much lower rates were used for development. It proved easy to integrate support for AVI files as well enabling streaming from this readily available media format. Inadvertently, this shows off the capability of Video Monitor decoupling the media displayed from the transport mechanism.

Future Considerations to Implementation

There are a few key components to our design that were intended to be completed, but simply gave way to more pressing needs, the following details the remaining steps to provide support for these featuers.

ServerObject

The ServerObject is intended at a base level to bridge two channels. This implementation proved more difficult than needed for the demonstration of our software and should be implementable with some focus on working with LiveObjects. The server also can provide any sort of support wanted by the user, such as frame rate management or other manipulations of the video streams.

Web Cam Support

While at the core of the original model, it seemed more pertinent to implement the underlying network and get images flowing through the network than dealing with the problems of connecting a camera. Though, support for cameras is easily implemented and should be designed to work with whatever video source a developer is looking for. On this same note, another feature that would aid in reducing network overhead is image compression/decompression, for which there are also many useful libraries available and would only require minimal changes to the implementation to integrate.

Full Network Implementation

Currently, features of the network implementation are deactivated to allow for frame rate testing and due to a few remaining bugs in the message management. These features could be quickly reactivated with some basic testing of message passing interfaces that were simply deactivated to debug other functionality.

Deployment

A Deployment of this system would present two challenges beyond the above features requiring implementation. The first of which being limited by the necessary point configuration of LiveObjects on each computer a component is installed on. And secondly, the system does not provide guarantee of message delivery, which our consideration for a video stream of images was decided as not critical, whereas if a MPEG stream or another stream dependant on progressive layering of frames were utilized,

this would become an issue. Other than these considerations, the system should deploy over a large system with a protocol such as QSM.

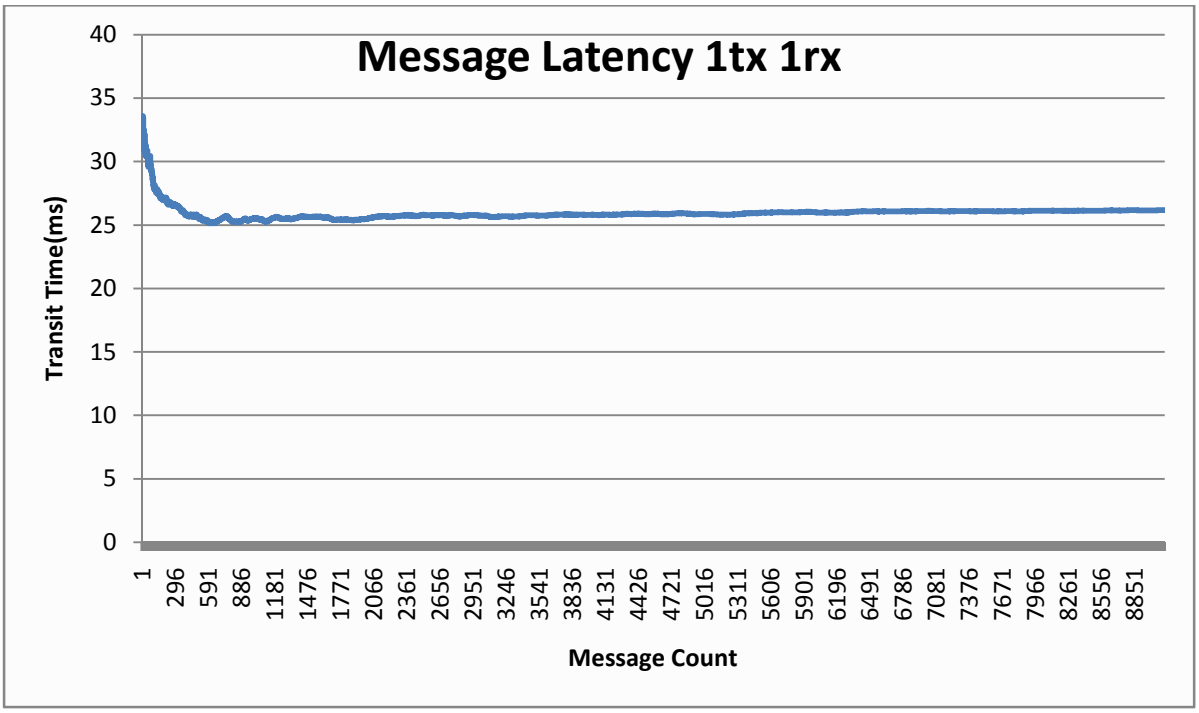
Configuration – Setup Guide

The setup process does not differ far from the sample examples provided. Additional dlls have been added though they are distributed automatically by the post-build batch script. It is recommended that the project is built allowing the post-build batch script to run and distribute files accordingly. Liveobjects is assumed to be installed at c:\liveobjects as suggested. Video Monitor has been tested both in localhost operation and over a local network between multiple computers.

The configuration process for running multiple computers follows the exact guide posted on the CS514 newsgroup. This worked as specified. Video Monitor creates its own communication channel (channel 100). To run a sample configuration, go to the liveobjects folder in the project directory. Open either videosource.liveobject or videoviewer.liveobject and wait several seconds. It is important to wait about 10 seconds for the addresses to be assigned properly. After ten seconds, open the other interface. If an active internet connection is present it is possible to start streaming from a webcam which is currently hardcoded to a stream showing the Eiffel Tower. It is also possible to stream AVI files (currently mislabeled WMV files). The AVI decoder being used is somewhat sensitive to different AVI codecs, the simpler the better. We found some classic footage of Charles Lindbergh nicely encoded available here http://www.charleslindbergh.com/movies/index.asp which worked well.

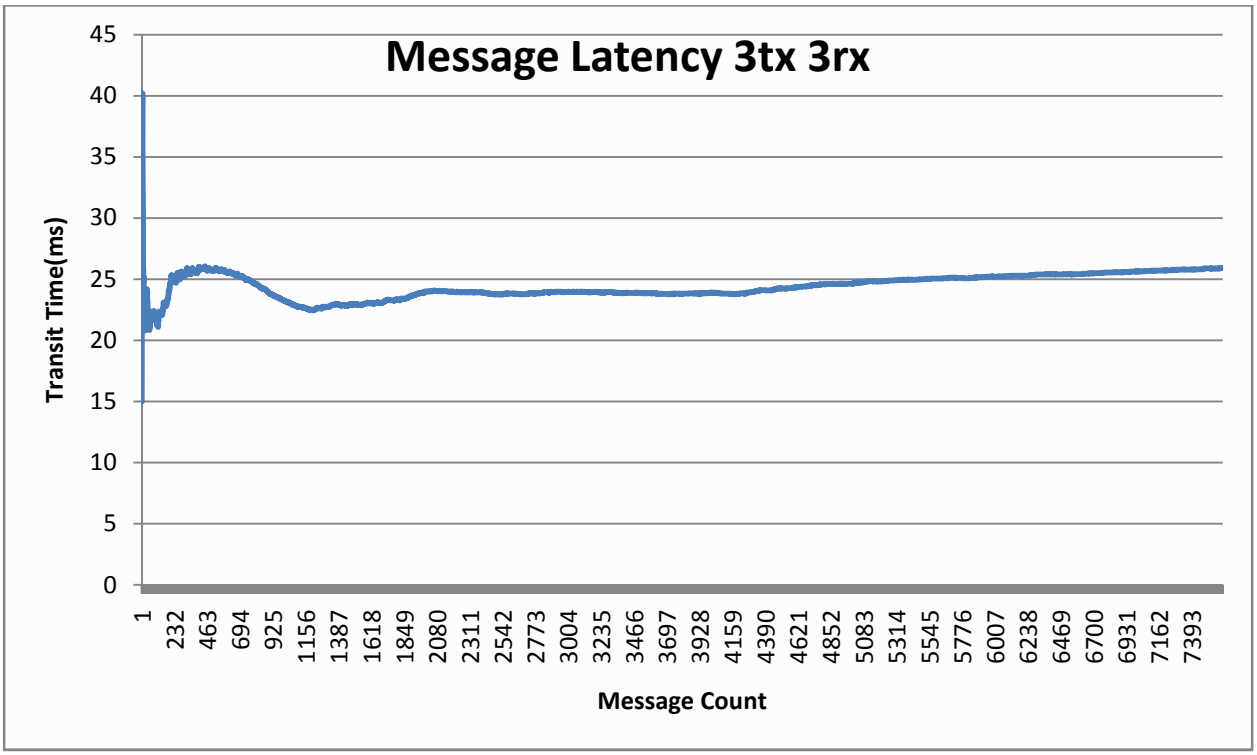
Performance Analysis

Data was gathered on end to end message latency under two sample configurations. Both tests were performed on a single machine. The first configuration served as a baseline for the second. The first used a single transmitter (Source) and single receiver (Viewer). This was meant to classify performance in a near ideal situation. Message latency under these conditions was expected to be very low showing the delay caused by the overhead provided by LiveObjects primarily.



Surprisingly this latency was quite high for a localhost test, a simple sockets level piece of software written in c++ reveled latency <1ms while our dataset revealed an average latency of 25ms, a rather significant reduction in performance. Without the tools to dig deeper our analysis of this issue must stop here, although this overhead must be noted particularly in designing real time applications where greater than 50ms of latency is considered unusable. With a video distribution system such as ours, buffers can be used to reduce the impact of this lack of performance. However if this were to be used as the basis for a voip or video conference type piece of software it may be necessary to take more aggressive steps to attain acceptable performance levels.

The second test configuration utilized three separate transmitters (Sources) and three separate receivers (Viewers). This was expected to result in notably more overhead than the previous example because each viewer would actively filter messages from the sources that were not selected. As described above a significantly more complex system was initially envisioned which would limit of the impact of these issue. However this test proved very interesting primarily because it performed at exactly the same level as the previous test. This would indicate any added latency from inefficiencies in the application were trivial. Seeing this information was encouraging and in looking forward at continued development would be an excellent indicator as to a strategy to allocate limited development resources.



The interesting transient performance through the first several hundred messages is very interesting. The settling time is rather long, on the order of minutes, which would seem to rule out hardware or software caching etc. Without better analysis tools of what the system is doing, it is very difficult to attribute a reason to this performance. The shape of the transient is notably different in each case, again it is difficult to draw any conclusions from the information available.

System Correctness

This system, while incomplete, can be defined as “correct”. From the original system design, the intent of the system was to provide streaming ability from a source to a viewer over a network that can provide a distributed network model to viewers, that they might be able to select a source an monitor it. With this high-level design in mind, the system does perform this functionality and provides an excellent framework for future development.

Conclusions

Working with Liveobjects as an educational tool for understanding distributed systems was an interesting experience. The Video Monitor project development was forced to reconsider system requirements repeatedly during the development process as schedules slipped due to difficulties with Liveobjects. However, with any difficult challenge, it serves as an excellent learning experience, exposing the authors to new technologies, new software development approaches, and most importantly in the context of this class, paradigms in distributed system architecture design.

Screenshots

The screenshot below shows 2 VideoSources broadcasting to 2 VideoViewers.