vic: A Flexible Framework for Packet Video

Steven McCanne
University of California, Berkeley
and Lawrence Berkeley Laboratory
mccanne@ee.lbl.gov

Van Jacobson
Network Research Group
Lawrence Berkeley Laboratory
van@ee.lbl.gov

ABSTRACT

The deployment of IP Multicast has fostered the development of a suite of applications, collectively known as the MBone tools, for real-time multimedia conferencing over the Internet. Two of these tools — *nv* from Xerox PARC and *ivs* from INRIA — provide video transmission using software-based codecs. We describe a new video tool, *vic*, that extends the groundbreaking work of nv and ivs with a more flexible system architecture. This flexibility is characterized by network layer independence, support for hardware-based codecs, a conference coordination model, an extensible user interface, and support for diverse compression algorithms. We also propose a novel compression scheme called "Intra-H.261". Created as a hybrid of the nv and ivs codecs, Intra-H.261 provides a factor of 2-3 improvement in compression gain over the nv encoder (6 dB of PSNR) as well as a substantial improvement in run-time performance over the ivs H.261 coder.

KEYWORDS

Conferencing protocols; digital video; image and video compression and processing; multicasting; networking and communication.

1 INTRODUCTION

Over the past few years, a collaborative effort in the network research community has produced a suite of tools for multimedia conferencing over the Internet [16, 25, 26, 39, 42]. The driving force behind these tools is Deering's IP Multicast [11], a technology which extends the traditional IP routing model for efficient multipoint packet delivery. The incremental deployment of IP Multicast has been realized by building a (temporary) virtual multicast network on top of the existing Internet, which Casner has dubbed the Multicast Backbone, or MBone [6].

The first applications to provide video over the MBone were the Xerox PARC Network Video tool, *nv*, and the INRIA Video Conferencing System, *ivs*. While these two systems share the goal of supporting low bit-rate multicast video over the Internet, their approaches are markedly different. Ivs is an integrated audio/video conferencing system that relies exclusively on H.261 [45] for video compression. By adopting a standardized algorithm, ivs can interoperate with a large installed base of H.320 video codecs as an H.320-compliant bit stream is easily generated from an H.261 packet stream by introducing H.221 framing in software [20].

In contrast, nv is a "video-only" application that utilizes a custom coding scheme tailored specifically for the Internet and targeted for efficient software implementation [17]. Because of its low computational complexity, the nv codec can run much faster than an H.261 codec. Even though H.261 has better compression performance than nv, nv is more often used by the MBone community because of its better run-time performance.

Inevitably, in pioneering work such as nv and ivs, restrictions must be imposed on the design process to facilitate experimentation. For example, the ivs design assumes video is represented as 4:1:1-decimated CCIR-601 YUV, while nv assumes 4:2:2 decimation. Extending nv to support H.261 or ivs to support nv-style compression would require non-trivial design changes. Also, since both systems are based on software compression, their video capture models were designed around uncompressed video. Extending either to support hardware-based compression engines would be relatively difficult.

In this paper, we describe a third model for a packet video application, realized in the UCB/LBL video conferencing tool, *vic*. Vic builds upon the lessons learned from ivs and nv by focusing on flexibility. It is an extensible, object-oriented, application framework that supports

- multiple network abstractions,
- hardware-based codecs,
- a conference coordination model,
- an extensible user interface, and
- diverse video compression algorithms.

Moreover, we have combined Frederick's insights from the nv codec with the compression advantages and standards compliance of H.261 in a novel scheme which we call *Intra-H.261*. Intra-H.261 gives significant gain in compression performance compared to nv and substantial improvement in both run-time performance and packet-loss tolerance compared to ivs.

Vic was originally conceived as an application to demonstrate the Tenet real-time networking protocols [14] and to simultaneously support the evolving "Lightweight Sessions" architecture [24] in the MBone. It has since driven the evolution of the Real-time Transport Protocol (RTP) [40]. As RTP evolved, we tracked and implemented protocol changes, and fed back implementation experience to the design process. Moreover, our experience implementing the RTP payload specification for H.261 led to an improved scheme based on macroblock-level fragmentation, which resulted in a revised protocol [44]. Finally, the RTP payload specification for JPEG [13] evolved from a vic implementation.

In the next section, we describe the design approach of the MBone tools. We then discuss the essentials of the vic network architecture. The network architecture shapes the software architecture, which is discussed in the following section. Finally, we discuss signal compression issues, deployment and implementation status.

2 COMPOSABLE TOOLS VS. TOOLKITS

A cornerstone of the Unix design philosophy was to avoid supplying a separate application for every possible user task. Instead, simple, one-function "filters" like *grep* and *sort* can be easily and dynamically combined via a "pipe" operator to perform arbitrarily complex tasks. Similarly, we use modular, configurable applications, each specialized to support a particular media, which can be easily composed via a *Conference Bus* to support the variety of conferencing styles needed to support effective human communication. This approach derives from the framework proposed by Ousterhout in [33], where he claims that large systems are easily composed from small tools that are glued together with a simple communication primitive (e.g., the Tk *send* command). We have simply replaced his *send* primitive with a well-defined (and more restrictive) Conference Bus protocol. Restricting the protocol prevents the evolution of sets of tools that rely on the specifics of each other's internal implementations. In addition to vic, our conferencing applications include the Visual Audio Tool (vat) for audio [26], a whiteboard (wb) for shared workspace and slide distribution [25, 15], and the Session Directory (sd) for session creation and advertisement [23].

This "composable tools" approach to networked multimedia contrasts with the more common "toolkit framework" adopted by other multimedia systems [10, 31, 37, 38]. Toolkits provide basic building blocks in the form of a code library with an application programming interface (API) to that library providing high-level abstractions for manipulating multimedia data flows. Each distinct conferencing style requires a different application but the applications are typically simple to write, consisting mostly of API calls with style-dependent glue and control logic.

The toolkit approach emphasizes the programming model and many elegant programming mechanisms have resulted from toolkit-related research. To simplify the programming model, toolkits usually assume that communication is application independent and offer a generic, least-common-denominator network interface built using traditional transport protocols.

In 1990 Clark and Tennenhouse [8] pointed out that multimedia applications could be simplified and both application and network performance enhanced if the network protocol reflected the application semantics. Their model, Application Level Framing (ALF), is difficult to implement with toolkits (where application semantics are deliberately "factored out") but is the natural way to implement "composable tools". And ALF-based, media-specific tools offer a simple solution to multimedia's biggest problem — high rate, high volume, continuous media data streams. Since the tools are directly involved in processing the multimedia data flows, we can use ALF to tune all the performance-critical multimedia data paths within the application and across the network.

In addition to performance, flexibility is gained by composing simple tools rather than using a monolithic application built on top of some API. Since each tool deals directly with its media stream and sends only low-rate reports like "X has started/stopped sending" on the Conference Bus, the coordination agent necessary to implement a particular conferencing scenario can be written in a simple interpreted language like *Tcl* [34]. This allows the most volatile part of the conferencing problem, the piece that melds audio, video, etc., into a coordinated unit that meets particular human needs and expectations, to be simple and easy to evolve. It also ensures that the coordination agents are designed orthogonal to the media agents, enforcing a mechanism/policy separation: media tools implement the mechanism by which coordination tools impose the policy structure appropriate for some particular conferencing scenario, e.g., open meeting, moderated meeting, class, seminar, etc.

3 NETWORK ARCHITECTURE

While the freedom to explore the communications protocol design space fosters innovation, it precludes interoperability. Since the MBone was created to study multicast scaling issues, interoperability is especially important. Multicast use at an interesting scale requires that a large group of people spread over a large geographic region have some reason to send and receive data from the group. One good way to achieve this is to develop interoperable applications that encourage widespread use.

3.1 RTP

To promote such interoperability, the Audio/Video Transport Working group of the Internet Engineering Task Force (IETF) has developed RTP as an application level protocol for multimedia transport. The goal is to provide a very thin transport layer without overly restricting the application designer. The protocol specification itself states that "RTP is intended to be malleable to provide the information required by a particular application and will often be integrated into the application processing rather than being implemented as a separate layer." In the ALF spirit, the semantics of sev-

RTP		
UDP	RMTP	AAL5
IP	RTIP	ATM

Figure 1: RTP and the Protocol Stack

eral of the fields in the RTP header are deferred to an "RTP Profile" document, which defines the semantics according to the given application. For example, the RTP header contains a generic "marker" bit that in an audio packet indicates the start of a talk spurt but in a video packet indicates the end of a frame. The interpretation of fields can be further refined by the "Payload Format Specification". For example, an audio payload might define the RTP timestamp as a audio sample counter while the MPEG/RTP specification [22] defines it as the "Presentation Time Stamp" from the MPEG system specification.

Because of its ALF-like model, RTP is a natural match to the "composable tools" framework and serves as the foundation for vic's network architecture. Since RTP is independent of the underlying network technology, vic can simultaneously support multiple network protocols. Figure 1 illustrates how RTP fits into several protocol stacks. For IP and IP Multicast, RTP is layered over UDP, while in the Tenet protocols, it runs over RMTP/RTIP [2]. Similarly, vic can run directly over an ATM Adaptation Layer. In all these cases, RTP is realized in the application itself.

RTP is divided into two components: the data delivery protocol, and the control protocol, RTCP. The data delivery protocol handles the actual media transport, while RTCP manages control information like sender identification, receiver feedback, and cross-media synchronization. Different media of the same conference-level session are distributed on distinct RTP sessions.

Complete details of the RTP specification are provided in [40]. We briefly mention one feature of the protocol relevant to the rest of the paper. Because media are distributed on independent RTP sessions (and because vic is implemented independently of other multimedia applications), the protocol must provide a mechanism for identifying relationships among media streams (e.g., for audio/video synchronization). Media sources are identified by a 32-bit RTP "source identifier" (SRCID), which is guaranteed to be unique only within a single session. Thus, RTP defines a canonical-name (CNAME) identifier that is globally unique across all sessions. The CNAME is a variable-length, ASCII string that can be algorithmically derived, e.g., from user and host names. RTCP control packets advertise the mapping between a given source's SRCID and variable-length CNAME. Thus, a receiver can group distinct RTP sources via their CNAME into a single, logical entity that represents a given session participant.

In summary, RTP provides a solid, well-defined protocol framework that promotes application interoperability, while its ALF philosophy does not overly restrict the application design and, in particular, lends itself to efficient implementation.

4 SOFTWARE ARCHITECTURE

The principles of ALF drove more than the vic network architecture; they also determined the overall software architecture. Our central goal was to achieve a flexible software framework which could be easily modified to explore new coding schemes, network models, compression hardware, and conference control abstractions. By basing the design on an objected-oriented ALF framework, we achieved this flexibility without compromising the efficiency of the implementation.

ALF leads to a design where data sources and sinks within the application are highly aware of how data must be represented for network transmission. For example, the software H.261 encoder does not produce a bit stream that is in turn packetized by an RTP agent. Instead, the encoder builds the packet stream fragmented at boundaries that are optimized for the semantics of H.261. In this way, the compressed bit stream can be made more robust to packet loss.

At the macroscopic level, the software architecture is built upon an event-driven model with highly optimized data paths glued together and controlled by a flexible Tcl/Tk [34] framework. A set of basic objects is implemented in C++ and are coordinated via Tcl/Tk. Portions of the C++ object hierarchy mirror a set of object-oriented Tcl commands. C++ base classes permit Tcl to manipulate objects and orchestrate data paths using a uniform interface, while derived subclasses support specific instances of capture devices, display types, decoder modules, etc. This division of low-overhead control functionality implemented in Tcl and performance critical data handling implemented in C++ allows for rapid prototyping without sacrifice of performance. A very similar approach was independently developed in the VuSystem [29].

4.1 Decode Path

Figure 2 roughly illustrates the receive/decode path. The elliptical nodes correspond to C++ base classes in the implementation, while the rectangular nodes represent output devices. A Tcl script is responsible for constructing the data paths and performing out-of-band control that might result from network events or local user interaction. Since Tcl/Tk also contains the user interface, it is easy to present control functionality to the user as a single interface element that might invoke several primitive control functions to implement its functionality.

The data flow through the receive path is indicated by the solid arrows. When a packet arrives from the network, the Network object dispatches it to the Demuxer which implements the bulk of the RTP processing. From there, the packet is demultiplexed to the appropriate Source object, which represents a specific, active transmitter in the multicast session. If no Source object exists for the incoming packet, an up-call into Tcl is made to instantiate a new data path for that source. Once the data path is established, packets flow from the source object to a decoder object. Hardware and soft-

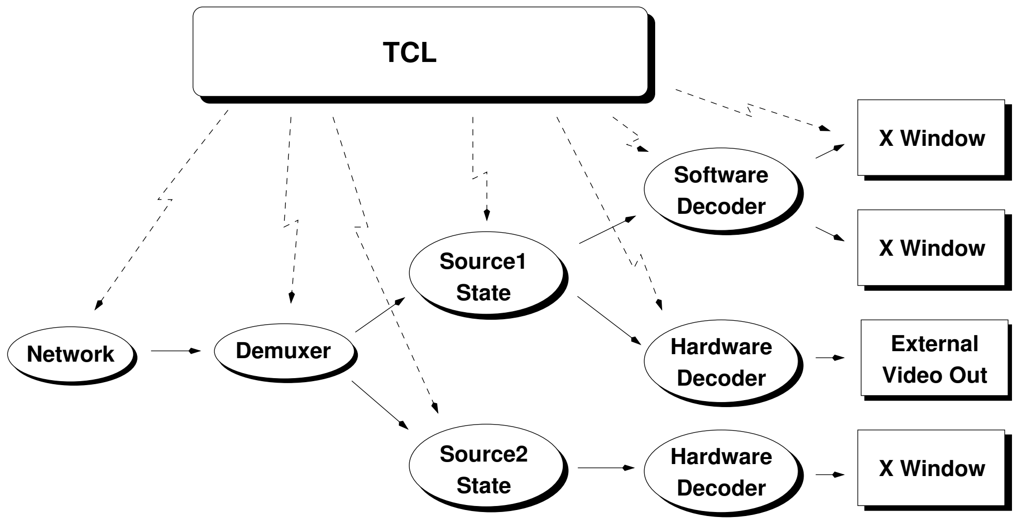


Figure 2: The receive/decode data path.

ware decoding, as well as multiple compression formats, are simultaneously supported via a C++ class hierarchy. When a decoder object decodes a complete frame, it invokes a rendering object to display the frame on the output device, either an X Window or external video output port.

Note that, in line with ALF, packets flow all the way to the decoder object more or less intact. The decoder modules are not isolated from the network issues. In fact, it is exactly these modules that know best what to do when faced with packet loss or reordering. C++ inheritance provides a convenient mechanism for implementing an ALF model without sacrificing software modularity.

While this architecture appears straightforward to implement, in practice the decode path has been one of the most challenging (and most revised) aspects of the design. The core difficulty is managing the combinatorics of all possible configurations. Many input compression formats are supported, and deciding the best way to decode any given stream depends on user input, the capabilities of the available hardware, and the parameters of the video stream. For example, DEC's J300 adaptor supports hardware decompression of 4:2:2-decimated JPEG, either to an X Window or an external output port. The board can be multiplexed between capture and decoding to a window but not between capture and decoding to the external port. Also, if the incoming JPEG stream is 4:1:1 rather than 4:2:2-decimated, the hardware cannot be used at all. Finally, only JPEG-compressed streams can be displayed on the video output port since the system software does not support a direct path for uncompressed video. Many other devices exhibit similar peculiarities.

Coping with all hardware peculiarities requires building a rule set describing legal data paths. Moreover, these rules depend intimately on how the application is being used, and therefore are complicated by user configuration. We have found that the Tcl/C++ combination provides a flexible solution for this problem. By implementing only the bare essentials in C++ and exporting a Tcl interface that allows easy creation, deletion, and configuration of C++ objects, the difficulty in managing the complexity of the data paths is greatly reduced.

4.2 Capture Path

We applied a similar architectural decomposition to the video capture/compression path. As with the decoder objects, encoder objects perform both compression and RTP packetization. For example, the H.261 encoder fragments its bit stream into packets on "macroblock" boundaries to minimize the impact of packet loss.

Different compression schemes require different video input formats. For instance, H.261 requires 4:1:1-decimated CIF format video while the nv encoder requires 4:2:2-decimated video of arbitrary geometry. One implementation approach would be for each capture device to export a common format that is subsequently converted to the format desired by the encoder. Unfortunately, manipulating video data in the uncompressed domain results in a substantial performance penalty, so we have optimized the capture path by supporting each format.

A further performance gain was realized by carrying out the "conditional replenishment" [32] step as early as possible. Most of the compression schemes utilize block-based conditional replenishment, where the input image is divided up into small (e.g., 8x8) blocks and only blocks that change are coded and sent. The send decision for a block depends on only a small (dynamically varying) selection of pixels of that block. Hence, if the send decision is folded in with the capture I/O process, most of the read memory traffic and all of the write memory traffic is avoided when a block is not coded.

4.3 Rendering

Another performance-critical operation is converting video from the YUV pixel representation used by most compression schemes to a format suitable for the output device. Since this rendering operation is performed after the decompression on uncompressed video, it can be a bottleneck and must be carefully implemented. Our profiles of vic match the experiences reported by Patel et al. [35], where image rendering sometimes accounts for 50% or more of the execution time.

Video output is rendered either through an output port on an external video device or to an X window. In the case of an X window, we might need to dither the output for a color-mapped display or simply convert YUV to RGB for a true-color display. Alternatively, HP's X server supports a "YUV visual" designed specifically for video and we can write YUV data directly to the X server. Again, we use a C++ class hierarchy to support all of these modes of operation and special-case the handling of 4:2:2 and 4:1:1-decimated video and scaling operations to maximize performance.

For color-mapped displays, vic supports several modes of dithering that trade off quality for computational efficiency. The default mode is a simple error-diffusion dither carried out in the YUV domain. Like the approach described in [35], we use table lookups for computing the error terms, but we use an improved algorithm for distributing color cells in the YUV color space. The color cells are chosen uniformly throughout the feasible set of colors in the YUV cube, rather than uniformly across the entire cube using saturation to find the closest feasible color. This approach effectively doubles the number of useful colors in the dither. Additionally, we add extra cells in the region of the color space that corresponds to flesh tones for better rendition of faces.

While the error-diffusion dither produces a relatively high quality image, it is computationally expensive. Hence, when performance is critical, a cheap, ordered dither is available. Vic's ordered dither is an optimized version of the ordered dither from nv.

An even cheaper approach is to use direct color quantization. Here, a color gamut is optimized to the statistics of the displayed video and each pixel is quantized to the nearest color in the gamut. While this approach can produce banding artifacts from quantization noise, the quality is reasonable when the color map is chosen appropriately. Vic computes this color map using a static optimization explicitly invoked by the user. When the user clicks a button, a histogram of colors computed across all active display windows is fed into Heckbert's median cut algorithm [21]. The resulting color map is then downloaded into the rendering module. Since median cut is a compute-intensive operation that can take several seconds, it runs asynchronously in a separate process. We have found that this approach is qualitatively well matched to LCD color displays found on laptop PCs. The Heckbert color map optimization can also be used in tandem with the error diffusion algorithm. By concentrating color cells according to the input distribution, the dither color variance is reduced and quality increased.

Finally, we optimized the true-color rendering case. Here, the problem is simply to convert pixels from the YUV color space to RGB. Typically, this involves a linear transformation requiring four scalar multiplications and six conditionals. Inspired by the approach in [35], vic uses an algorithm that gives full 24-bit resolution using a single table lookup on each U-V chrominance pair and performs all the saturation checks in parallel. The trick is to leverage off the fact that the three coefficients of the Y term are all 1 in the linear transform. Thus we can precompute all conversions for the tuple $(0, U, V)$ using a 64KB lookup table, T. Then, by linearity, the conversion is simply $(R, G, B) = (Y, Y, Y) + T(U, V)$.

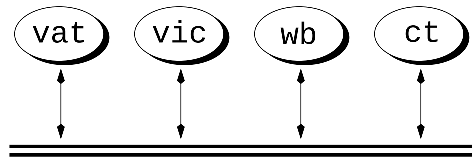


Figure 3: The Conference Bus.

A final rendering optimization is to dither only the regions of the image that change. Each decoder keeps track of the blocks that are updated in each frame and renders only those blocks. Pixels are rendered into a buffer shared between the X server and the application so that only a single data copy is needed to update the display with a new video frame. Moreover, this copy is optimized by limiting it to a bounding box computed across all the updated blocks of the new frame.

4.4 Privacy

To provide confidentiality to a session, vic implements end-to-end encryption per the RTP specification. Rather than rely on access controls (e.g., scope control in IP Multicast), the end-to-end model assumes that the network can be easily tapped and thus enlists encryption to prevent unwanted receivers from interpreting the transmission. In a private session, vic encrypts all packets as the last step in the transmission path, and decrypts everything as the first step in the reception path. The encryption key is specified to the session participants via some external, secure distribution mechanism.

Vic supports multiple encryption schemes with a C++ class hierarchy. By default, the Data Encryption Standard (DES) in cipher block chaining mode [1] is employed. While weaker forms of encryption could be used (e.g., those based on linear feedback shift registers), efficient implementations of the DES give good performance on current hardware (measurements are given in [27]). The computational requirements of compression/decompression far outweigh the cost of encryption/decryption.

4.5 The Conference Bus

Since the various media in a conference session are handled by separate applications, we need a mechanism to provide coordination among the separate processes. The "Conference Bus" abstraction, illustrated in Figure 3, provides this mechanism. The concept is simple. Each application can broadcast a typed message on the bus and all applications that are registered to receive that message type will get a copy. The figure depicts a single session composed of audio (vat), video (vic), and whiteboard (wb) media, orchestrated by a (yet to be developed) coordination tool (ct).

A complete description of the Conference Bus architecture is beyond the scope of this paper. Rather, we provide a brief overview of the mechanisms in vic that support this model.

Voice-switched Windows. A feature not present in the other MBone video tools is vic's voice-switched windows. A window in voice-switched mode uses cues from vat to focus on the current speaker. "Focus" messages are broadcast by vat over the Conference Bus, indicating the RTP CNAME of the current speaker. Vic monitors these messages and switches the viewing window to that person. If there are multiple voice-switched windows, the most recent speakers' video streams are shown. Because the focus messages are broadcast on the Conference Bus, other applications can use them for other purposes. For example, on a network that supports different qualities of service, a QoS tool might use the focus message to give more video bandwidth to the current speaker using dynamic RSVP filters [5].

Floor Control. All of the LBL MBone tools have the ability to "mute" or ignore a network media source, and the disposition of this mute control can be controlled via the Conference Bus. This very simple mechanism provides a means to implement floor control in an external application. One possible model is that each participant in the session follows the direction of a well-known (session-defined) moderator. The moderator can give the floor to a participant by multicasting a *takes-floor* directive with that participant's RTP CNAME. Locally, each receiver then mutes all participants except the one that holds the floor. Note that this model does not rely on cooperation among all the remote participants in a session. A misbehaving participant cannot cause problems because it will be muted by all participants that follow the protocol.

Synchronization. Cross-media synchronization can also be carried out over the Conference Bus. Each real-time application induces a buffering delay, called the *playback* point, to adapt to packet delay variations [24]. This playback point can be adjusted to synchronize across media. By broadcasting "synchronize" messages across the Conference Bus, the different media can compute the maximum of all advertised playout delays. This maximum is then used in the delay-adaptation algorithm. In order to assure accurate synchronization, the semantics of the advertised playback points must be the delay offset between the source timestamp and the time the media is actually transduced to the analog domain. The receiver buffering delay alone does not capture the local delay variability among codecs.

Device Access. Each active session has a separate conference bus to coordinate the media within that session. But some coordination operations like device access require interaction among different sessions. Thus we use a global conference bus shared among all media. Applications sharing a common device issue *claim-device* and *release-device* messages on the global bus to coordinate ownership of an exclusive-access device.

Conference Buses are implemented as multicast datagram sockets bound to the loopback interface. Local-machine IP multicast provides a simple, efficient way for one process to send information to an arbitrary set of processes without needing to have the destinations "wired in". Since one user may be participating in several conferences simultaneously, the transport address (UDP destination port) is used to create a separate bus for each active conference. This simplifies the communication model since a tool knows that everything it sends and receives over the bus refers to the conference it is participating in and also improves performance since tools are awakened only when there is activity in their conference. Each application in the conference is handed the address (port) of its bus via a startup command line argument. The global device access bus uses a reserved port known to all applications.

4.6 User Interface

A screen dump of vic's current user interface, illustrating the display of several active video streams, is shown in Figure 4. The main conference window is in the upper left hand corner. It shows a thumbnail view of each active source next to various identification and reception information. The three viewing windows were opened by clicking on their respective thumbnails. The control menu is shown at the lower right. Buttons in this window turn transmission on and off, select the encoding format and image size, and give access to capture device-specific features like the selection of several input ports. Bandwidth and frame rate sliders limit the rate of the transmission, while a generic quality slider trades off quality for bandwidth in a fashion dependent on the selected encoding format.

This user interface is implemented as a Tcl/Tk script embedded in vic. Therefore, it is easy to prototype changes and evolve the interface. Moreover, the interface is extensible since at run-time a user can include additional code to modify the core interface via a home directory "dot file".

A serious deficiency in our current approach is that vic's user interface is completely independent of the other media tools in the session. While this modularity is fundamental to our system architecture, it can be detrimental to the user interface and a more uniform presentation is needed. For example, vat, vic, and wb all employ their own user interface element to display the members of the session. A better model would be to have single instance of this list across all the tools. Each participant in the listing could be annotated to show which media are active (i.e., a participant may have audio but no video).

We intend to evolve our tools in this direction by merging the user interfaces of the different tools into an integrated interface. Each of the tools will be reduced to a bare application that performs only network and media processing. The integrated tool would orchestrate a given session structure by configuring the bare tools over the Conference Bus. Different application styles and arrangements could be easily implemented as separate programs, using the scripting language to realize the user interface and orchestration. Moreover, this decomposition makes the tools less dependent on the X environment. For example, the bare vic process would need only enough window system specific code to draw pixels into a display window. With the forthcoming port of Tcl/Tk to Windows95, our bare application framework should readily migrate to the PC environment.

Figure 4: Vic's User Interface

5 SIGNAL COMPRESSION

The flexible software architecture presented in the previous section is well suited for experimentation with new signal compression schemes. This is especially important since video compression algorithms have been traditionally designed for constant bit-rate channels and new approaches are required for transmission over heterogeneous packet networks [18, 41] like the Internet. In its current form, the Internet is a relatively harsh environment for compressed video signals. Packet loss rates are often significant and loss patterns are bursty [4]. Compression schemes that rely on low bit error rate or on channel coding techniques to effectively reduce the bit error rate do not operate well under these conditions.

For compression algorithms like MPEG and H.261, packet loss causes sustained degraded quality. This is due to their method of removing temporal redundancy: Both use a motion-compensating predictor to predict blocks in the current frame from previous (or future) frames then code the residual prediction error. Because the prediction is based on the decoded signal, the model assumes the decoder shares an identical state. But when packet loss occurs, the decoder state becomes mismatched and quality degrades with each new frame. MPEG and H.261 both rely on intra-mode updates to eventually resynchronize, but at low bit rates, the resynchronization intervals can be tens or hundreds of frames so the probability of error in any given interval is high enough that the decoded bit stream is virtually never error free. The solution is to reduce the resynchronization interval, in the extreme case, to a single frame. This is the model used in Motion-JPEG, where each frame is coded and transmitted independently of all others; but this approach results in low compression efficiency because much of the transmission is redundant.

Conditional replenishment. Another solution to this problem is to forego motion compensation, and instead employ a very rudimentary form of prediction. Frederick's insight in nv was to use an aggressive, block-based conditional replenishment scheme. In this model, each video frame is partitioned into small blocks and only the blocks that change (beyond some threshold) are transmitted. Furthermore, block updates are always intra-coded (i.e., dependent only on the current block) to avoid persistent errors in a predictor loop. At some low rate, a background process continuously refreshes all the blocks in the image to guarantee that lost blocks are eventually retransmitted.

Conditional replenishment works well in practice for several reasons. First, block updates are "self-correlated" in the sense that a block is usually transmitted because of motion in the scene. Therefore, that same block will likely be transmitted again in the next because of the spatial locality of the motion. Thus a lost block update is often retransmitted immediately as a side-effect of conditional replenishment. Second, the class of video currently sent over the Internet is primarily teleseminars and video conferences where large static backgrounds often dominate the scene and conditional replenishment is highly effective. Third, because the replenishment decision can be carried out in the pixel domain as the first step in processing, much of the encoder computation is shed by coding only small portions of the image. Finally, computational complexity is further reduced by the fact that a copy of the decoder need not be run at the encoder since there is no prediction carried out.

The Nv Codec. The high-level compression model utilized by nv is decomposed as follows [17]:

Here, 8x8 image blocks from the conditional replenishment stage are transformed using a Haar wavelet decomposition. A threshold is then applied to each coefficient such that coefficients with magnitude below the threshold are set to zero. This process creates runs of zeros, which are run-length coded in the last stage. Since the Haar transform requires only additions and subtractions and the threshold step requires only two conditionals, the algorithm has very low computational complexity. Unfortunately, compression performance suffers because the Haar transform provides relatively poor energy compaction of the signal [28] and the entropy coder is based exclusively on fixed size run-length codes.

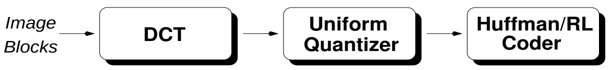
The performance of the nv coder can be substantially improved with the following changes:

(i) Replace the Haar transform with the discrete cosine transform (DCT), which has good energy compaction for images [28].

(ii) Replace the coefficient threshold stage with a uniform quantizer to reduce the entropy of quantized coefficients.

(iii) Follow the run-length coder with a Huffman coder to further compress the symbol stream.

The modified encoder structure then becomes:



Intra-H.261. These changes amount to applying the compression advantages of the DCT-based approaches (JPEG, MPEG, and H.261) to nv's coder. Since the scheme so closely matches H.261, it makes sense to create an H.261 variant that leverages off the ideas from nv. In fact, it turns out that one can get the advantages of aggressive conditional replenishment and intra-coded blocks with a fully-compliant H.261 syntax.

In vic, we use this technique, which we call "Intra-H.261". Intra-H.261 uses only intra-mode H.261 macroblock types and uses macroblock addressing to skip over unreplenished blocks. Because the encoder uses only a small, simple subset of the H.261 specification, the implementation is straightforward (a few hundred lines of C++).

We achieve reasonable computational performance by folding quantization into the DCT computation and by using an efficient 80-multiply 8x8 DCT [36]. We experimented with several vector-radix DCTs [7] but found that the separable row-column approach, though having asymptotically higher complexity, performed better in the 8x8 case because of reduced memory traffic. Furthermore, because Intra-H.261 never sends inter-coded blocks, the algorithm need not compute a prediction error signal. This prunes much of the computation because it eliminates the need to run a copy of the decoder within the encoder (i.e., it eliminates an inverse quantization and inverse DCT of every encoded block).

Performance. Because the Intra-H.261 and nv compression schemes use similar conditional replenishment algorithms, we can evaluate their relative compression performance simply by ignoring the temporal dimension and comparing only their 2D image compression performance. Figure 5 shows the performance of the two approaches for the canonical 8-bit, 512x512 grayscale "Lena" image. Both encoders were modified to omit block-addressing codes to allow the H.261 encoder to operate on a non-standard image size. This modification has little impact on the results since block-addressing accounts for a small fraction of the bit rate.

The peak signal-to-noise ratio (PSNR) is plotted against rate (in bits per pixel). Multiple points were obtained by varying the H.261 scalar quantizer and the nv dead-zone threshold. Note that the nv algorithm was intended to operate with a non-configurable, fixed threshold, but we explored other thresholds to complete a rate-distortion curve.

As seen in the graph, H.261 consistently outperforms the nv coder by 6-7dB. Since transmissions are typically rate-limited, we should consider a fixed distortion and compare the corresponding bit rates. From this perspective, the nv bit rate is two to three times that of H.261. For a rate-limited transmission, this translates into a factor of two to three decrease in frame rate.

Unfortunately, the compression advantages of H.261 come at the cost of increased computational complexity. Most modern workstations can handle the computational burden of H.261 for typical MBone rates (128kb/s). However, performance problems on lower end workstations have been reported. We believe these problems can be solved by gracefully shedding load to adapt to the available CPU resources. This work is currently underway.

We also compared the run-time performance of the H.261 encoders of vic and ivs. Vic version 2.6.2 and ivs version 3.4 were tested on an SGI Indy (133MHz MIPS R4600SC) with the built-in VINO video device. Both applications were compiled with gcc 2.6.2 using the -O2 flag. We ran the programs individually, giving each application a CIF-sized high-motion input and a low-motion input. In order to measure the maximum sustainable compression rate, we disabled the bandwidth controls in both tools and ran the test on an unloaded machine. We measured the resulting frame rates by decoding the streams on separate machines. The results are as follows:

	high-motion	*low-motion*	*cpu. util.*
ivs	3.5 f/s	20 f/s	100%
vic	8.5 f/s	30 f/s	40%

For the high motion case, almost all the blocks in each frame are coded, so this gives a worst-case performance bound. For the low-motion scene, vic ran at the full NTSC frame rate