Taming the Complexity of Distributed Multimedia Applications

Frank Stajano and Rob Walker

Olivetti Research Limited
24A Trumpington Street
Cambridge CB2 1QA
United Kingdom

{fstajano, rwalker}@cam-orl.co.uk
http://www.cam-orl.co.uk/
Phone: (+44 1223) 343.000
Fax: (+44 1223) 313.542

Abstract

The Medusa environment for networked multimedia uses Tcl to compose applications out of low-level processing blocks called modules. A medium-sized application such as a two way multistream videophone already uses around one hundred interworking modules, running in parallel on several host machines. This paper shows how we overcome the inherent complexity of such applications: to deal with parallelism we use a multi-threaded library hidden behind a single-threaded Tcl interpreter; to build higher order components than the modules we use the object oriented extension [incr Tcl]; and to exploit the variety of available input and output devices we adopt the Model-View-Controller paradigm.

1. Introduction

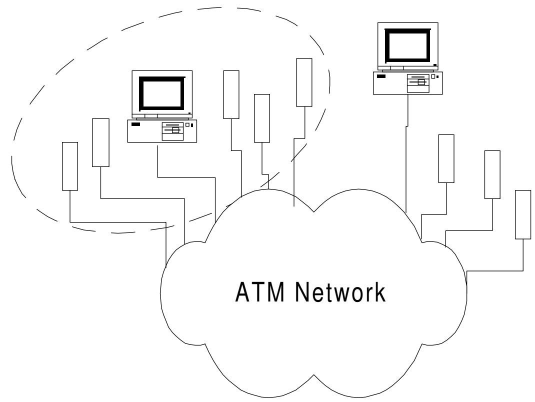
Medusa *[Medusa-ICMCS 94]* is an applications environment for distributed multimedia designed to support many simultaneous streams. Tens of parallel streams and over a hundred software modules are used in some of our applications and several such applications may be active in the system. Input devices in the broadest sense (from cameras and microphones to sensors and location equipment like Active Badges *[Badges 94]*) can all be used as sources of multimedia streams. The consumers of such streams may be human, for example someone watching a video, or computer-based, such as an analyser recognising a gesture made in front of a camera. It is not uncommon for the same camera stream to be simultaneously sent to a number of video windows on different X servers, a storage repository on the network and a video analyser module on a processor bank. Modules, the software units that generate, process and consume the data, are optimised for efficiency and written in C++. Applications, designed for flexibility and configurability, are created in Tcl/Tk: they instantiate the required modules on the appropriate networked hosts, they connect them together and control them according to the user's directives.

The first few applications were successful and they raised expectations for the subsequent ones. As soon as one aspect of the system became configurable in one program, all the other programs were required to offer the same flexibility. While the modules remained the same, the Tcl portion of the applications started to grow in size and complexity over the limits of what could easily be managed within the applications framework we had one year ago *[Medusa-Tcl 94]*. This paper presents some problems we had to face in three key areas (asynchronous programming, reusable components, multiple interfaces) as a consequence of this growing complexity, together with our solutions.

The inherent parallelism of the system called from the start for an asynchronous interface to the modules. It was not trivial, however, to schedule the various Medusa tasks and the Tcl interpreter in a way that would ensure consistency and efficiency. Maintaining a simple API from Tcl was also a high priority, and a challenging task. We finally implemented a multi-threaded interface to the lower levels of Medusa, but we presented it as single-threaded from Tcl. We also developed a new programming construct, the asynchronous context, to support efficient error checking in an asynchronous environment from within the single-threaded Tcl layer.

After some experience with the system it soon became clear that an extra level of abstraction was highly desirable between the low-level modules and the complete application. Some common patterns of modules emerged repeatedly and the obvious desire was to encapsulate such a medium-level structure into something as self-contained as a module, of which one could create new instances as required. We adopted [incr Tcl] for this *[incr Tcl 93]*.

The first Medusa applications had traditional mouse-based interfaces implemented in Tk. As our data analysers became more refined, we started investigating alternative input devices like pen, gesture and voice. An application could check for input on all these different channels, but it was preferable to find a way to abstract the input interface from the required action. The symmetrical problem was found on the output side, where the same message could be presented through a variety of media including text, video, recorded audio and synthetic voice. We used the Model-View-Controller paradigm for this *[Smalltalk 90]*.



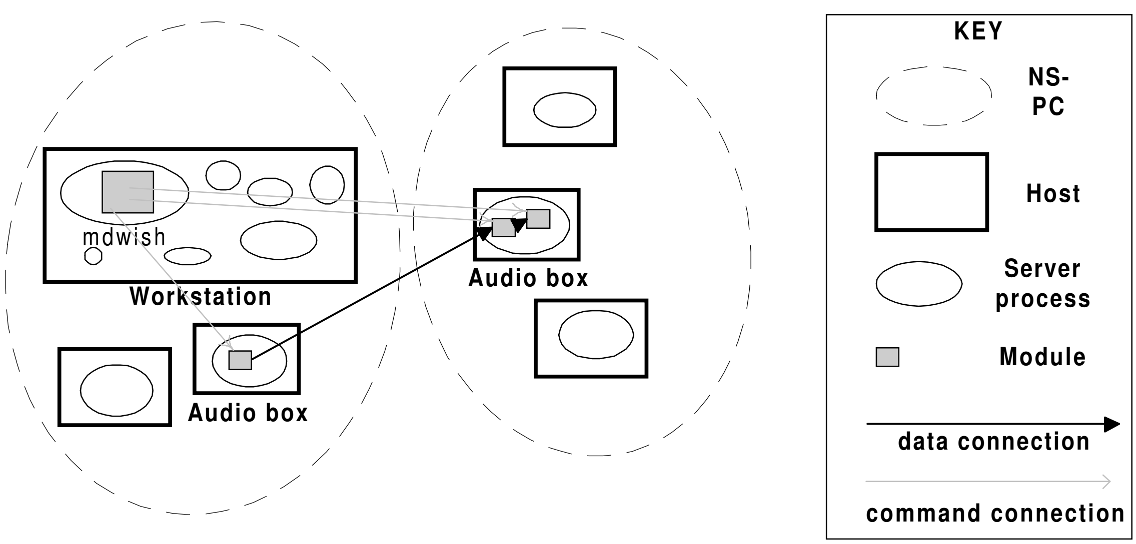
The Network Scalable Personal Computer

2. Medusa - a brief overview

To follow the rest of the discussion, an understanding of the basics of the Medusa architecture is required. An in-depth presentation of Medusa would be outside the scope of this paper, but a brief overview will be given here. More details may be found in [Medusa-ICMCS 94], [Medusa-Tcl 94] and [Medusa-video 95].

The hardware on which Medusa runs is based around the concept of the Network Scalable Personal Computer (NS-PC): the multimedia devices are not peripherals of the main computer but are first-class network citizens with their own ATM connection. This has several advantages. One of them is scalability: some NS-PCs may have one camera while others have six, without the need for an overdimensioned backplane; in some cases a Network Scalable PC might not even include a workstation — it might be, for example, the logical union of a video tile (a stand-alone LCD display) and an audio brick. Another benefit is the avoidance of undesired interference between streams: in a traditional architecture the digital audio data has to be routed through the workstation's bus, thus degrading the performance of both the workstation and the audio link itself.

From a software point of view, Medusa uses a peer-to-peer architecture in which modules, linked by



NS-PCs, hosts, servers and modules across the network.

connections, exchange data and commands. Modules reside in server processes distributed across the hosts of a heterogeneous network. However connections between modules have the same behaviour and properties regardless of the relative location of the modules they connect — to Medusa, a connection between modules in the same process is no different from one between modules on different machines. At this low level, even the controlling Tcl application is seen as a module; the application process contains a module with many command connections to all the modules it creates.

3. Parallelism, asynchronous execution and threads

Any Medusa process, both the servers containing the data processing modules and the application process containing the control module, must contain the Medusa scheduler, which manages the processing of messages sent and received over the connections. Because the application process is implemented as a Tcl interpreter containing, among others, the Tk and Medusa extensions, the interaction between the Medusa scheduler and the Tk event loop is particularly important.

Our original implementation of the Tcl interpreter for Medusa was single-threaded. While standard Tcl commands were executing, Medusa was locked out. This didn't affect the multimedia performance as the interpreter only dealt with the *control* of modules; the data could still flow through the worker modules, which resided in other processes. Still, this meant that a Medusa-triggered callback might have to wait for a while before being invoked. Because Medusa activity was considered to be high priority, whenever Medusa regained control (because Tcl invoked a Medusa command) it allowed its Tcl callbacks to be invoked. In retrospect this was a bad design because it rendered procedures interruptible by callbacks, whereas they aren't in plain Tcl/Tk.

Tcl does not have any synchronisation primitives to disable interruptions by callbacks scheduled by **`-command`** options, **`after`** events or **`bind`** commands. On the other hand, it doesn't need them because its mode of operation is such that these callbacks can only be invoked when the interpreter is idle or when a flush is explicitly requested via **`update`**. In other words, callbacks can't interrupt anything else by definition. The queuing of the pending callbacks is taken care of by the system and is transparent to the application programmer. We considered whether to introduce "disable/enable interrupts" primitives, but we soon decided that Tcl's original model was the one to adopt. Procedures must be uninterruptible unless they request an **`update`**. This method is very simple, sufficiently powerful and, most importantly, very natural for the programmer — so much so that in most cases the programmer doesn't even think about what could happen in the middle of what else, and yet Tcl does the right thing in the end. So we followed this philosophy in our new implementation.

The development of a threaded API to the Medusa module layer prompted a major change from the original implementation: we made the new Medusa-Tcl interpreter multithreaded. One thread runs the interpreter while other worker threads running in parallel carry out the individual Medusa commands. The commands for each module must be executed in strict sequence, but the interpreter need not be held up waiting for each one to finish before issuing the next, and commands for different modules can be executed in parallel.

Each Medusa command spawns its own thread and immediately returns control to the interpreter (unless synchronous execution is explicitly requested for that command) so that the Tcl flow of control can move on to the next instruction.

To sequence the commands pertaining to a given module, every module has a FIFO queue of worker threads. All threads, except the head of the queue, are blocked, and when a thread finishes it drops off the queue, allowing the one behind it to run.

Although this change involved a complete rewrite of the interpreter, we maintained complete compatibility with the previous Tcl-level API, so that existing scripts could run unmodified. We maintained, for example, the scheme by which Medusa commands can have optional **`-success`** and **`-failure`** callbacks associated with them for the management of asynchronous calls.

While the syntax described in *[Medusa-Tcl 94]* was retained for compatibility, some new alternative constructs were introduced to make the Tcl programming interface to the modules even more similar to that of Tk's widgets.

```
module .m \
     -host duck -server mgbvideo \
     -factory CameraDevice0 \
     -module video_source \
     -frame_rate 1/2
```

The module, once created, has a **`configure`** method through which its attributes can be inspected and changed:

```
.m configure -frame_rate
⇒ 1/2

.m configure -frame_rate 1/3
⇒ 1/3
```

This syntax implicitly uses the **getattributes** and **setattributes** methods of the module.
It is also possible to bind a Tcl variable to an attribute so that changes to one will be reflected in the other and vice versa. This implicitly uses the **watchattributes** method of the module to propagate changes from Medusa to Tcl, and the **trace** mechanism on the variable to propagate changes in the reverse direction. This is most useful for controlling an attribute with a Tk widget that can be linked to a variable. The variable becomes the contact point between the module and the widget.

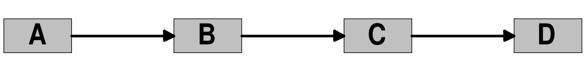
```
module bind attrName varName

.volumeSlider bind gain scalePos
scale .s -variable scalePos
```

As we shall see in greater detail later on, the variable can be viewed as the Model in a Model-View-Controller setup; the scale is then both a View and a Controller for the variable. The same goes for the attribute.

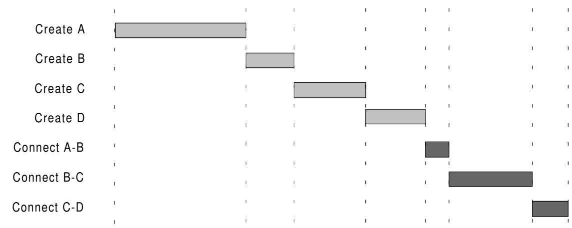
3.1 An example of asynchronous programming issues: building a pipeline

Because Medusa is a fully distributed system in which the multimedia peripherals are first-class networked computers, most connections from an ultimate source to an ultimate sink (such as from a camera to a video tile, possibly going through buffers, gates, format converters and so on) involve modules that are on different hosts from each other and from the host that the Tcl program is running on. The **connect** method is used to tell a module to connect to another module. Setting up the complete pipeline involves creating the *n* distributed modules and connecting up the *n*-1 pairs by issuing the **connect** method on one of the modules of each pair.

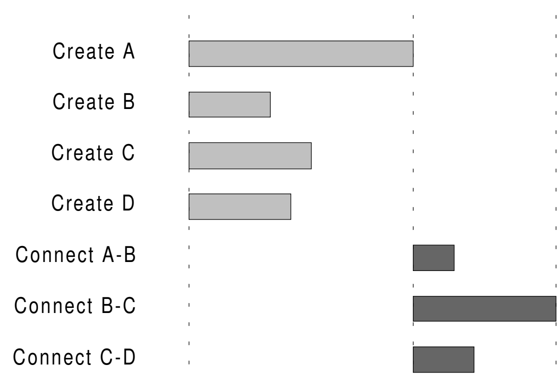


The pipeline of modules we want to create

Without the programming constructs to specify asynchronous execution, one would create all the modules sequentially (each time waiting for completion of the previous creation) and then connect all the pairs in turn. Neglecting, as a first approximation, the time spent issuing the commands, the situation can be represented as follows, with time on the horizontal axis:



This is clearly unacceptable and was never considered as a viable solution.
In our first implementation the command to create new modules could be invoked asynchronously but it did not support **-success** and **-failure** callbacks. The **MDsynch** *?module?* command (similar to **update**) blocked the interpreter until no more commands were pending on a given module, or on all the modules known to the interpreter. In this environment one would create all the modules concurrently and then, for each pair in turn, **MDsynch** the modules of the pair and then connect them before proceeding to the next pair. The resulting situation is:



This is a substantial improvement but still isn't fully optimised. Because at connect time the A-B pair is processed before the B-C pair, the commands "MDsynch A; MDsynch B; Connect A-B" precede "MDsynch C; Connect B-C" and the issuing of the "Connect B-C" command has to wait unnecessarily for the creation of A (specifically because of the "MDsynch A" in the treatment of the A-B pair), whereas it could have started earlier otherwise. This is because there is only one Tcl thread of execution and so **MDsynch**, like **update**, blocks the whole interpreter.

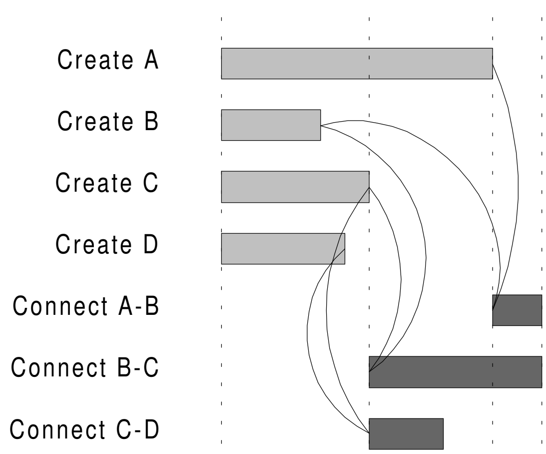
Note that the above example, for illustration purposes, shows a worst-case scenario in which the creation of the first module takes the longest time; on the other hand the method we outlined cannot optimise the order of creations in advance, because the creation times for the various modules are unknown before creating the modules.

It is interesting to note that even the availability of **-success** and **-failure** callbacks is not sufficient to produce the optimum version without resorting to global flags. The problem is that we can't schedule the "Connect A-B" in the **-success** callback for either A or B, because the other module might not be ready to accept a connection. We would have instead to schedule something like the following proc:

```
proc conditionallyConnect {A B} {
  # Invoked on successful creation
  # of A; tries to connect it to B.
  global created
  if {[info exists created($B)]} {
    $A connect $B
  } else {
    set created($A) 1
  }
}
```

Every module except the first and last must have two of these calls in its **-success** callback, one to attempt to connect it to the previous module and one to attempt to connect it to the next, because it is not known in advance which module of any given pair will be ready first.

The optimal solution, obtainable through the above strategy, has the following diagram:



But this is where our new threaded implementation proves its worth: its sequencing of asynchronous commands is such that this behaviour is obtained by default, with no effort required by the programmer. It is sufficient to issue all the creation commands and then all the connection commands, without having to register any callback or global flag. The create commands will start in parallel for the different modules, each being the head of its queue. The command to connect two modules is effectively present in two queues, but it doesn't hold the second queue up for any length of time — it is just there for synchronisation purposes, to ensure the module has been created. The actual connection operation starts as soon as both modules exist.

A key feature of this scheme is that the command that creates a module, even if it returns before having completed the creation, ensures that the thread queue for the module is ready before returning. This allows the system to accept and enqueue commands (such as **connect**) for that module even before the module actually exists.

3.2 Programming for robustness in an asynchronous environment

One of the main reasons why prototypes are so much quicker to write than full-fledged applications is that in writing a prototype one can afford to omit a great deal of detail about error checking. While in the prototype an action can simply be expressed by issuing the corresponding command, in a properly written application the command must be wrapped inside a layer of code that traps any potential errors, checks the return code and possibly recovers by attempting an alternative operation. It is well recognised *[Man-Month 75]* that this error catching in itself adds a significant layer of complexity to the application.

But the problem only becomes worse in an environment using asynchronous programming. To be able to check the return code, which is what Tcl's **catch** does, the command must have completed its execution, successfully or not. However the asynchronous strategy, which is fundamental to achieve the desired levels of efficiency especially in a distributed system, is to issue the command in the background and move to the next one at once, well before completion; so **catch** would not work here, because the command to be caught returns control to the interpreter immediately, before knowing whether it will succeed or fail. In a prototype which doesn't check return codes the asynchronous approach is easy to follow: as long as all the commands succeed, everything will work and go very fast; as soon as one command fails, though, the whole prototype will fall over. For an application that wants to be robust, checking the return codes is reasonable, if only tedious, when one can afford to wait until the commands complete before proceeding, i.e. when one has the luxury of using the **`catch`** approach; but having to support both error checking *and* asynchronous programming multiplies, instead of simply adding, the complexities of the two approaches.

It should be noted in passing that the fact of dealing with a distributed environment makes error checking a necessity for every application which is not a prototype. You wouldn't normally think of **`catch`**ing every button creation in your Tk program, because if the program's window comes up at all it is highly probable that it will be able to create all the buttons it needs. But when the items you create are modules running in external processes and often on external hosts, it would not be wise to assume that because the program has started on the local computer then all these remote operations will succeed too.

The way Medusa controls asynchronous programming is, as we've seen, by making the **`-success`** and **`-failure`** options available to every Medusa command. This allows the programmer to register two callbacks, one of which will be invoked when the command completes. While this construct is expressive and complete, at the Tcl level it forces a programming style where the code to perform a complex action, instead of being written as a self-contained procedure, has to be scattered among the nodes of a binary tree of callbacks. We can say from experience that this leads to programs of low readability that are quite hard to maintain and update.

To overcome this problem we developed a new programming construct, the *asynchronous context*, which allows several related actions to be launched as a group, registering **`-success`** and **`-failure`** callbacks for the whole group without having to follow the outcome of the individual actions.

Let's use the pipeline example from section 3.1 to show how this construct works. Assume that all the modules have been created and that we want to connect up all the pairs, as fast as possible and checking for any errors.

We create an asynchronous context object, say *c*, and register our **`-success`** and **`-failure`** callbacks with it; these callbacks will be relative to the success or failure of the entire operation, not of the individual commands. From within *c*, we launch all the required Medusa commands: this causes them to be spawned asynchronously, with hidden **`-success`** and **`-failure`** callbacks added by the context which will notify *c* of the completion of the launched actions. When we have launched all the required operations, we issue *c*'s **`commit`** method, which means that no more commands are to be launched from within this context. From now on, as soon as all the commands launched so far have completed, *c* will invoke one of its two completion callbacks.

The corresponding procedure can be written as follows:

```
proc pipe {args} {
# argument parsing omitted for
# brevity. The result of it is that
# the local variables success,
# failure and modules are created.

  set c [async_context #auto \
    -success $success \
    -failure $failure \
    -persistent 0]
  set l [llength $modules]
  for {set i [expr $l-1]} {$i>0} \
      {incr i -1} {
    set sink [lindex $modules $i]
    set source [lindex $modules \
        [expr $i-1]]
    $c launch $i-th_pair \
        $sink connect \
        up input   $source down
  }
  $c commit
}
```

You may have noticed the **`-persistent 0`** option in the command that creates the context. This instructs the context object to destroy itself after it has reached completion and invoked one of its two callbacks, relieving the programmer from having to keep track of the context herself. In other cases one may want the object to stay around even after completion, so as to be able to query its state or to invoke some of its methods: **`-persistent 1`** will then be used.

More features available for the context object are shown below:

The `-sync` option

If set to 1, makes the **`commit`** method block until completion. Also available as a **`sync`** method for persistent objects.

The `-cleanuponfailure` option

A failure of the context means that at least one of the launched operations failed; but other launched operations may have succeeded. If this option is set to 1, in case of failure for each launched operation which succeeded a matching cleanup command will be called. This requires the programmer to register a cleanup command for every command that is launched. In the example above, one would register a matching **`disconnect`** for every **`connect`**. Incidentally, this is the reason why the launched operations carry an identifier (**`$i-th_pair`** above).

The `interrupt` method

The typical use of this is to launch a set of operations from the main program while allowing an external event (typically a button press from the user) to abort the sequence. The **`interrupt`** method can be invoked at any time during the lifetime of the context, even before a **`commit`**. It will have effect if it is received before completion, in which case it will force a failure. If the interrupt comes in before commit, all requests to launch commands are silently ignored (this saves time: if you've already decided to abort the lot, any operation requested after this decision is not even started). If the interrupt comes after the commit but before completion, it won't affect pending operations although it will still force a failure on completion. If the context has already completed then the interrupt has no effect.

The composition properties

The asynchronous context has good composition properties. The construction of a composite object containing several modules (see 4.3) can be wrapped into a context, thus providing **`-success`** and **`-failure`** callbacks for the composite object. From then on, the construction of such a composite object becomes itself an operation that can be launched from within a higher level context.

The asynchronous context has been a very successful addition to our programming toolkit. It allows us to perform the important task of error checking without giving up the efficiency advantages of asynchronous programming, all with a clean programming interface to single-threaded Tcl.

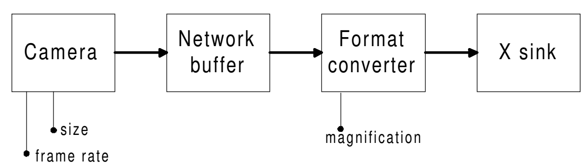
4. Composing modules into higher order objects

Just like the graphical side of a Tk application is built out of widgets such as buttons and listboxes, the multimedia side of a Medusa application is built out of modules such as cameras and speakers.

In both cases, after some practice in writing applications, it is frequently observed that common patterns emerge: groups of widgets or modules appear in the same arrangement from one program to the next. And, just like Tk programmers have often felt the need for mega-widgets, so Medusa programmers have often wanted mega-modules.

A typical example is the basic video pipeline that puts a video stream from a networked camera onto an X display. A camera module pumps data through a network buffer into a converter and an X sink. The converter translates the incoming video from the format produced by the source to the one accepted by the display (displays with different colour depths will represent pixels in different formats). The X sink puts the frames of video into an X window which may have been created by Tk.

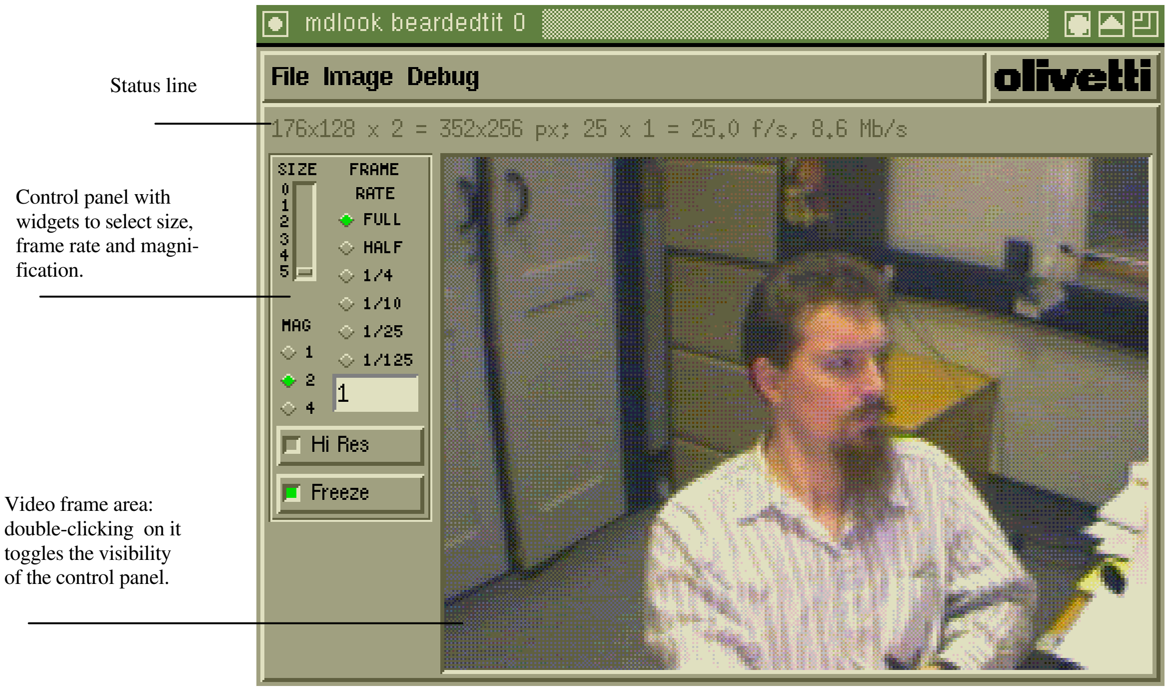
The basic parameters for such a pipeline are the



The modules of a video pipeline

frame rate, the size of the picture produced by the camera and the magnification factor (video frames can be resized in software by the converter module). Unfortunately, to change these parameters you must know which modules control them: the frame rate is set by the camera; the magnification factor, by the converter; the picture size, by the camera again; and all this without forgetting that, if you change the size or the magnification on the modules, you must also change the size of the Tk window containing the X sink. What is needed is a way to create the pipeline as an object which knows what internal operations to perform when requested to change those parameters.

This would allow Medusa programmers to develop a useful collection of parts for common tasks; new applications could then be written by concentrating on their new functionality instead of the perpetual recoding of the common portions. Ideally, for greater completeness, the object would also include the appropriate Tk widgets through which the parameters could be tuned by the user. Including the controlling widgets in the multimedia object itself also has the benefit that, whenever that object appears in an application, the user is always presented with a consistent interface that must be learnt only once.



A simple program incorporating a vwin (everything below the menu bar is part of the vwin)

One of the first requirements was for this level of grouping to be implemented in Tcl. Writing the objects in C++ was considered but had little to offer other than some potential efficiency gains; on the other hand it complicated the development process considerably, especially when embedding Tk widgets inside objects. We first tried using pure Tcl and one of the successful results was what we call a "vwin" object — the combination of a converter and X sink modules, a Tk window and some suitable controlling widgets.

The vwin was immediately popular and was adopted by practically all the video applications written after it — a practical demonstration of how much the need was felt for higher level reusable components. The drawback of this approach was that the vwin was not really an object but rather a group of modules, procedures and global variables. We therefore adopted the object oriented extension [incr Tcl], which provides language constructs for developing such groups as first-class objects and endowing them with a syntax similar to that of normal widgets and modules.

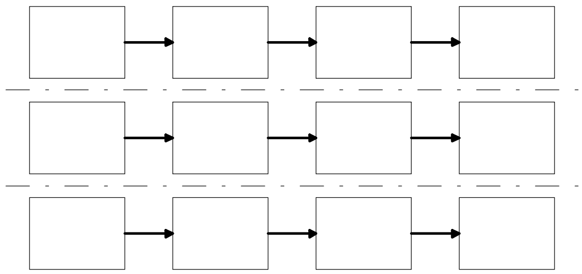
Of the three pillars of OOP — encapsulation, inheritance and polymorphism — what we needed most was encapsulation. With [incr Tcl], the vwin can at last become an object with its own methods and private data. We did not develop a class for modules in order to inherit from it, because our classes do not usually have an "is-a" relationship with modules: a class representing a pair of modules has both of them in it, but is neither; inheritance (even multiple) is generally inappropriate in our case. There is however scope for inheritance between objects. A basic video sink widget might simply consist of two Medusa modules and a frame widget, while a more complex one would inherit from this and have extra widgets to control the video parameters.

What we have described so far is similar to the experience of many other programmers in the field of Tk widgets: the "atoms" are too low level to build applications with, so we need a way of composing them into higher level "molecules" (with apologies to chemists!) with their own methods.

We do this by resorting to OOP, using mostly encapsulation and a bit of inheritance. In the field of distributed multimedia, however, there are new major complications, related to the issues of connections between objects and cardinality of connections.

4.1 Connections between objects

Modules, unlike widgets, must not only be grouped together — they must also be *connected* for data to flow. Grouping atoms into molecules is a hierarchical composition. But our atoms now also have connections between them. Can we allow connections to span molecule boundaries or not?



“Cutting horizontally” is our shorthand expression for “drawing molecule boundaries which are parallel to the data pipelines”.

If we go back to our video pipeline example it is easy to see that we would like to “cut horizontally” so as to have the camera, the converter and the Tk frame in the same molecule; this encapsulation would allow us to ask the molecule to change its magnification without us having to know which individual module to address. But on the other hand it is not unreasonable to want to cut vertically to obtain a source / sink arrangement, especially in the case of multiple parallel streams. It can be quite useful to be able to connect the multiple sources from NS-PC *A* to sinks on either *B* or *C*. But if we want to be able to cut both ways simultaneously, a hierarchical composition is no longer satisfactory. So we ask ourselves: should we look for a grouping strategy that is more flexible than the hierarchy?

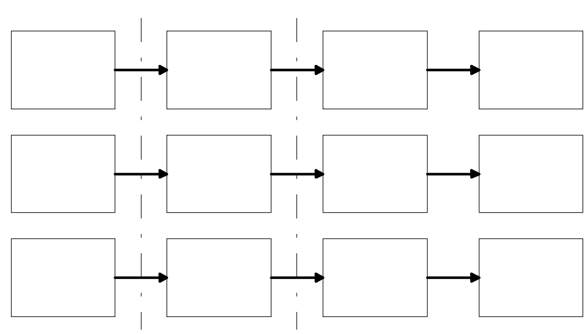
The problem of wanting to group atoms that have connections is found in other fields too, like digital circuit design, and we may draw some inspiration from there. The established practice in digital design is to allow connections to span molecule boundaries. An atom (a logic gate) has connections to the outside, and so has a molecule (a component like a counter). From the outside, atom and molecule are structurally alike: both have input and output ports accepting data and control information. The digital design point of view, compared to the multimedia point of view, does not give the same emphasis to sources and sinks of data, but nonetheless allows them to be represented in the model. In simple cases, like building a counter out of gates, it is the pattern of interconnections between the I/O ports of the atoms that defines the behaviour of the molecule. In more complex cases, where programmable logic is involved, the behaviour of the molecule may be defined by a set of instructions stored in a ROM.

This model, although purely hierarchical, has proved its worth in the hardware field. Does it work well with our multimedia modules? The answer is yes, as long as we are prepared to give up some flexibility. Because a hierarchy is involved, we cannot cut both horizontally and vertically: we must decide on one or the other beforehand, when we design the collection of reusable parts that we are going to build.

An important lesson to learn from the digital design field is that it is a good idea to make molecules that behave like atoms when seen from the outside. This means above all that they must be capable of being connected to atoms as well as to other molecules. Other features that would be desirable for uniformity but which are not essential to make the system work are the ability to export the internal modules’ attributes as if they were the molecule’s own, the ability to respond to the standard module methods (including for example **`watchattributes`** on the exported attributes) and the ability of being inspected with standard tools like **`MDpanel`** (a Tcl library procedure that pops up a window exposing the attributes of the module). The issue of making molecules look and behave like atoms is shared in the field of megawidgets *[incr Tk 94]*. We are still in the experimental stages at the time of writing, but it may well be the case that the solutions adopted by the Tk community to compose widgets may be fruitfully applied to composing modules.

An alternative solution is to provide the module look and feel not by emulation but through Medusa itself. We could build a special Medusa module that would do nothing except being a wrapper. This idea consists of recreating the module equivalent of what the frame is in the world of Tk widgets: a component that does nothing of its own but whose function is to group other components. Medusa already has facilities for module proxying and for handing off data connections to lower level implementation modules (see *[Medusa-ICMCS 94]*), so this strategy would save us from having to emulate a substantial amount of functionality.

The major piece of work would not go into creating



“Cutting vertically” in our language means drawing molecule boundaries that intersect the pipelines and consequently force the molecule to expose some I/O ports.

the wrapper module but rather into the equivalent of the **`pack`** command, say **`mdpack`**: the command that puts an existing module into the wrapper, connecting it with others in a user-specified pattern and exporting some of its attributes. This command should be exported to the Tcl level to allow molecules to be defined in Tcl.

Building a molecule amounts to creating a wrapper, creating the appropriate modules and plugging the modules together inside the wrapper with **`mdpack`**. [incr Tcl] steps in at this point, to allow the programmer to define molecule "types" instead of just instances. In the tradition of [incr Tcl], some renaming and aliasing acrobatics will become necessary to make the object respond to its module name as well as its object name, but this is an issue that has been dealt with before when building megawidgets.

In building hierarchies of molecules we believe that it will be advantageous to have a lower level where the molecules only contain modules and no Tk widgets. This allows the functionality to be separated from the user interface — a topic that will be treated in greater detail in section 5.

4.2 Cardinality of connections

As explained in section 2, the configuration of an NS-PC is completely flexible and the number of available devices varies from unit to unit.

We have talked of standard high level molecules and how to build them in the style of the digital circuit components, but there we implicitly assumed that a molecule had a definite structure and, for example, a given number of ports. If we want to design a molecule for a multi-headed video source, it must instead have as many output ports as there are available cameras on that given NS-PC; so, while the general structure will be the same, different instances will have different cardinalities. The constructor takes a list of the available cameras as a creation parameter.

On the sink side we have yet another situation. A multi-headed video sink is made of screen windows, but the NS-PC imposes no definite limit on the number of windows that can be created; it either does not have a screen, in which case it can't make any, or it has one, in which case it can make as many video windows as required. But if a multi-source is to be plugged into a multi-sink, the multi-sink must be ready to accept whatever the multi-source is ready to produce. Always over-dimensioning (and consequently under-using) is not a good solution. Over-dimensioning first and then killing off the dangling unused sources or sinks after the connection has been established is only slightly better.

This is our second major complication in grouping, which comes from the fact of dealing with a heterogeneous distributed system instead of a uniform configuration where every networked workstation has the same array of multimedia devices. In the digital design case, once a molecule containing five atoms of a kind has been defined, it is always possible to instantiate it; in the networked multimedia case, instead, the available resources are not known before runtime and so we need a more versatile molecule that, instead of five, has "as many atoms of that kind as it is possible to have on this host".

One solution is to give up completely on the multi-source and multi-sink arrangement and tend to cut horizontally, grouping one pipeline at a time. Later, all the parallel pipelines can be combined into a bigger molecule. This means giving up the advantages of having a multi-source or multi-sink as an object.

We have also been designing an alternative solution, based on delayed creation of the endpoints, which has the advantages of both the vertical and the horizontal styles of grouping: it sees the multiple endpoints as separate entities, but it can also address the multiple pipeline and the individual pipelines inside it. The key idea is that the multi-sink and multi-source must not be created in advance, but only when a specific multiconnection request is received. This way it becomes possible to work out the common subset beforehand and create two endpoints that match. This arrangement amounts to grouping vertically before the endpoints have been created and grouping horizontally once they exist. The main drawback is an increased setup time compared to the case where the endpoints are created in advance. The structure to implement this idea is rather elaborate and it is still part of our research to evaluate whether the benefits of this approach are worth the extra complication.

4.3 The current implementation

For most of the compositional approaches presented above we have built prototypes to evaluate their relative advantages and trade-offs. But we haven't yet developed any of them into a complete solution upon which to build an extensive library of components.

In the meantime, though, with or without a library, we had to produce some reliable applications to put our deployed hardware to good use, and for this task we built a few components as we went along, without a grand design behind them but with the practical goals of robustness and reusability.

These components are currently used in the supported multimedia applications that are in daily use at the lab: video mail, video phone, video peek and radio/tv/CD player. They are listed below.

The vwin

A video sink object with user controls for the stream parameters: picture size, magnification and frame rate.

The ringer

A complete audio pipeline from a file source to a speaker that can play a given sound sample and optionally repeat it at regular intervals until stopped.

The multicamera

A multi-headed video source combining many cameras in parallel and automatically adapting to the number of cameras available on the NS-PC where it is instantiated.

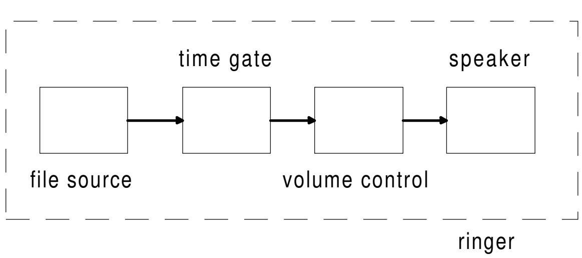
The multiwindow

A multi-headed video sink consisting of a dynamically variable number of parallel vwins; as the multiwindow is connected to a multicamera, it automatically creates or hides some of its vwins so as to match the number of channels of the multicamera.

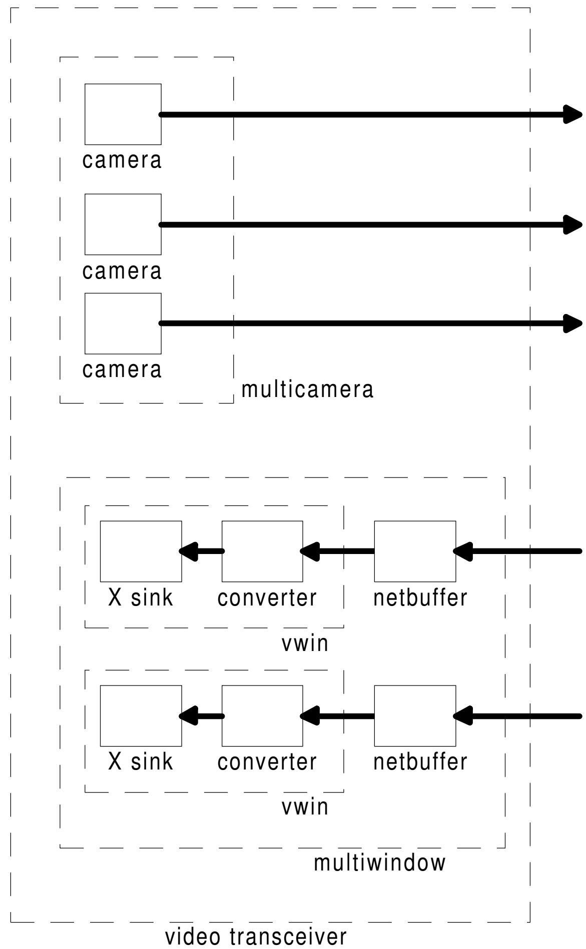
The video transceiver

The combination of a multicamera and a multiwindow. Can be connected to another transceiver for a bi-directional link, or it can be looped back on itself for a local view.

These components are built as [incr Tcl] objects containing Medusa modules and, where appropriate, Tk widgets. As can be seen we have performed the grouping both ways, depending on the circumstances. The ringer is built by "cutting horizontally", i.e. by creating an object which contains an entire pipeline and has no data connections to the external



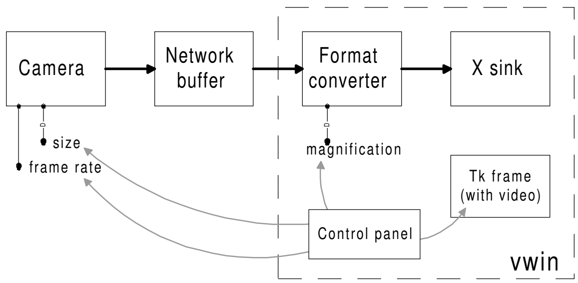
Internal structure of the ringer component



Internal structure of the video transceiver, showing a hierarchy of building blocks

world. The other objects are instead built by "cutting vertically", i.e. by running the object boundary around an endpoint of the pipeline and thus intersecting the data flow.

These components have been built by cutting either horizontally or vertically but not both, without resorting to the more complex "delayed creation" approach outlined earlier. The case of the vwin shows some of the problems of this limitation. This component is obviously an endpoint, i.e. an object that has been obtained by cutting vertically. However it has an element of "horizontal cutting" (which means "keeping the whole pipeline together") in that it contains user controls, like the picture size selector, that refer to parameters belonging to modules outside of itself, in this case the camera module. It is thus necessary to give the vwin a reference to the external module (the module name of the camera it is connected to) so that it can ask the camera to produce the appropriate size whenever the user acts on the vwin's controls.



The vwin's control panel has to act on modules outside the vwin's boundaries

This pointer going across the object boundary is the dirty part of this scheme. As new features are added to the vwin (this happened recently), the need arises for the vwin to talk to more modules in its pipeline that are outside of itself, and the proliferation of references to modules outside the vwin object is a bad thing.

The fact that these components can themselves be used as building blocks for other components, as exemplified by the multiwindow and the video transceiver, is a fundamental property. We would reject any composition method that only allowed one layer of grouping. The ability to make molecules out of other molecules, and not only out of atoms, allows us to build the library of components in a bottom-up style.

It must however be noted that the molecules we have built do not behave in all respects like atoms. The programming interface is similar for some aspects, like the availability of **`-success`** and **`-failure`** callbacks on most methods including creation, but the attributes, ports and capabilities mechanisms that all modules have are not supported nor emulated.

The molecules also have a richer set of methods than the atoms, with each component exposing new methods appropriate to the facilities it provides. This makes their programming interface more expressive, but it also means that they couldn't be easily treated by automatic tools like *Sticks and Boxes* (see *[Medusa-Tcl 94]*). It is debatable whether this is an asset or a liability; for the atoms, the ability to deal with them only through a limited set of standard commands is mostly an advantage, but for the molecules, given that they can encapsulate much higher level behaviour, it may not be appropriate to restrict the programming interface in a way that may not map naturally to the facilities they offer.

An important issue that comes out of this experience is the relevance of a strategy for connecting molecules that have connection ports with a composite structure. For the components presented above the problem was bypassed by designing the multiple endpoints in pairs and then giving one of them a **`connect`** method that knew how to talk to the other one. This works well as long as each type of component can only be connected to another given type; but as soon as a component can be connected to several other types of components, a clearly defined multi-connection interface becomes a necessity. Among the issues to be addressed (for which we do not have a complete solution yet) are the following.

Gender

Inputs and outputs must of course be distinguishable, and a port may only be plugged into a port of the opposite gender. But the case of a multiple port is less trivial than it sounds, as the subports composing the multiple port could be of different genders (as in the case of the transceiver). It all depends on whether one allows the connection between two transceivers to be seen as one fat bidirectional pipe, or whether one imposes the constraint that pipes can carry multiple streams but only in one direction.

Type

While there are good reasons for keeping the connections between modules as untyped, it is important to type the higher-level fat pipes that result from aggregation of many simple connections. Presumably this typing will have to be hierarchical, to respect the fact that the molecule out of which the fat pipe comes may be composed of several subpipes each coming from a submolecule and so on.

Cardinality

Suitable policies must be defined for the case when the two components to be connected match in gender and type but not in cardinality, e.g. one has five video inputs and another has three video outputs. In our multicamera / multiwindow implementation we arranged for the sink side to reconfigure itself to match the cardinality of the source side, but this will not be possible in every case. In such cases it must be decided whether to connect the common subset or to reject the connection request.

Self-description capacity

The multiport must encapsulate in itself a description of its gender, type etc. so that the connection operation can be made independent of the objects to be connected.

5. Designing multi-device interfaces

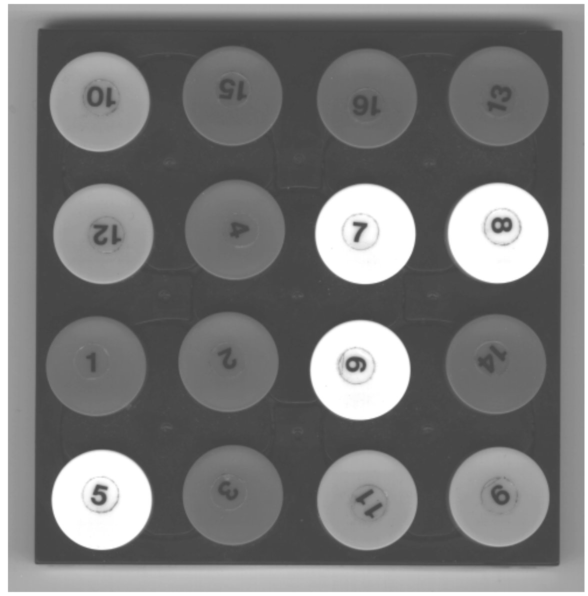
In what we call first-generation networked multimedia systems *[Pandora 90]*, the data streams are switched from one place to another in a “hands off” fashion. In second-generation systems like Medusa, however, the application can analyse and manipulate the data as it flows. This gives us the opportunity to use the multimedia streams as additional input and output user interfaces to control the application. Sound detection, speech recognition, motion detection, gesture recognition, speech synthesis are a few of the new input and output channels that an architecture like Medusa can support. Many “transducer” modules have been written which turn raw data into higher level events or vice versa.

From the point of view of the applications programmer, however, any one of these transducers is still a fairly low level component; it is inconvenient, for example, to have to accept input on several streams by redoing a similar programming job for each of the available input transducers.

We adopted the Model-View-Controller paradigm (MVC, see *[Smalltalk 90]*) as a higher level abstraction that shielded us from this problem. In this paradigm the unit of processing is the Model, which can perform several actions; the input devices are called Controllers: they may have very different ways of interfacing to the user, but they share a common interface to the Model; a similar arrangement exists for the output devices, called Views: they can show the state of the model in a variety of ways.

As a proof of concept, a demo application was written: Gripple, an electronic version of a commercial puzzle of the same name.

Gripple is rather similar to the classical “15 sliding tiles” puzzle; it consists of a set of sixteen numbered discs mounted on five interlocked rotating platforms. Each circular platform carries four discs. There is one platform for each of the corner quartets of discs and one platform in the centre of the board. By rotating the platforms, the discs move from one platform to another. The aim of the game is to bring the discs in the initial configuration after having scrambled the board.



The original Gripple puzzle, marketed by m-squared inc. in 1989

The Model contains a representation of the board describing the current disc positions. This is implemented simply as a list of sixteen elements.

The model accepts three commands:

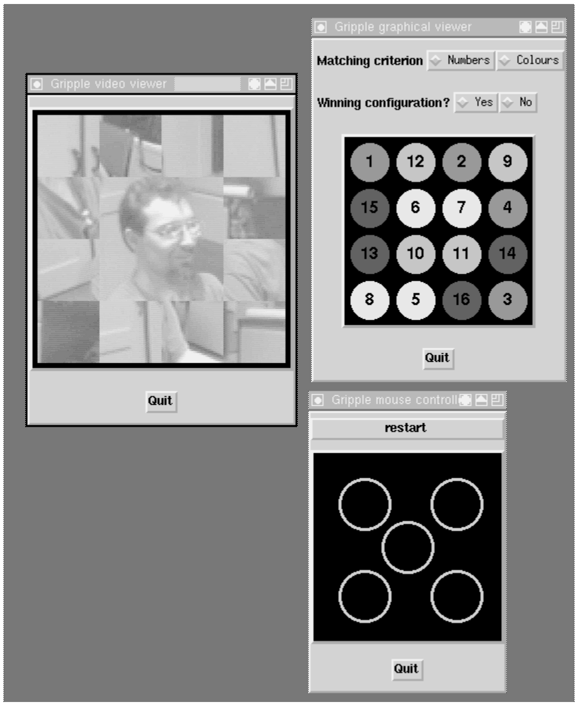
`rotate` *`platform direction`*
where *`platform`* is one of the five platforms and *`direction`* is either clockwise or anti-clockwise.

`restart` *`?difficulty?`*
where *`difficulty`* is a non-negative integer. If *`difficulty`* is omitted, the discs are positioned at random; otherwise, starting from the ordered configuration, that many random scrambling moves are performed (so lower difficulty settings correspond to easier configurations).

`reload` *`configuration`*
where *`configuration`* is a list of sixteen discs. This is used for studying moves.

As far as the model is concerned, all these commands ultimately have the effect of performing a specific permutation on the sixteen disks.

The simplest controller only has a textual interface through which the above commands can be sent to the model. The simplest view only shows the list of discs as text.



A screen shot showing the video view, the graphical view (note that the underlying model is the same, so the positions of the discs are coherent in the two views) and the mouse controller.

A more elaborate view draws coloured discs on a canvas and shows the rotation as an animation. Writing this component taught us that the model should send out “actions” to the views instead of its new state. For the view, performing the animation is much easier if the model says “rotate this platform that way” than if it says “the new configuration of my discs is the following”.

The mouse controller shows the five platforms. Clicking on a platform with the left button rotates it “to the left” (anticlockwise), and clicking with the right button rotates to the right.

To add to the challenge of the game, instead of using numbered discs another view uses video coming from a live television feed. The video is split into sixteen squares covering the same positions as the corresponding discs. Only when the puzzle is solved can one see the unscrambled picture. Because the target position of the squares is not immediately apparent, the puzzle is much harder if only this view is used. But, because many views can be attached to a model at the same time, one can always get help by temporarily looking at a “numbered discs” view.

The Hand Tracker controller uses an image processing algorithm *[Medusa-snakes 95]* which spots and tracks a hand in a scene from a camera. A drawing of the five platforms is superimposed on the scene; the player moves the hand to one of the platforms and rotates it in the required direction to trigger the corresponding action on the model. The tracking algorithm is based on active shape models.

The Voice controller *[Medusa-speech 95]* takes verbal instructions such as “rotate top left platform clockwise”. The speech recogniser is based on Hidden Markov Models. The use of a constrained grammar gives a high recognition rate — a fundamental requirement for usability.

From our Gripple experience we can abstract some general guidelines for designing applications that support multiple user interfaces.

The concerns are similar to those encountered when dealing with internationalisation of software applications. While in a single-language application it may be appropriate to embed messages to the user in strings within print statements, when the application has to be translated into many other languages it becomes necessary to add a level of indirection. Similarly, if direct programming of the I/O device may be sufficient for a basic keyboard-and-mouse application, a layer of indirection must be added if the same application is to support multiple user interfaces.

The application must be represented by a model with a well-defined set of methods. Controllers may be plugged into the model: they are the means through which the user can invoke the model’s methods. It is not required that every controller offer all the methods of the model: for example most of the controllers described above, except the text controller, do not offer the **`reload`** method and do not allow to specify a *`difficulty`* parameter to the **`restart`** method. Views show the state of the application. Having views and controllers that are not based on Tk allows us to run an application on a remote workstation and control it via local Medusa devices, which is useful for example in the case of a wall-mounted audio box in the corridor where there would be no workstation for traditional mouse-based interaction.

Using [incr Tcl] we have written Model, View and Controller classes that handle the communication between these elements and various management issues. For any given application, the programmer now designs specialised model, view and controller classes which inherit from the generic ones. If, some months later, a new I/O device is brought along, a new view or controller can be derived to make it interact with the application. Note that “device” should be taken in its widest sense: even a primitive image analysis module capable of discriminating between "no motion", "some motion" and "a lot of motion" can be considered as such a device; it is just a matter of mapping this three-way output onto some useful subset of the methods offered by the model.

An important development is the concept of using the MVC paradigm on a finer granularity than the whole application. It is quite useful, for example, to represent a communication link between two places with its own model and to be able to "view" this link in many ways: with audio and video for a workstation to workstation link, with audio only when one of the endpoints does not have a screen and so on. The application will always have to establish and shut down the communication link regardless of whether it includes video or not, so this abstraction helps us in isolating the internals of the application from the implementation details of how to address the various multimedia devices. The core of a conferencing application is independent of the nature of the links between the participants: ringing, accepting or rejecting calls, adding new participants and so on are actions that will always be present whatever the medium. According to the resources available at the different endpoints and to the users' preferences, each link will be "viewed" in the most appropriate way.

This novel way of applying the MVC paradigm appears to be quite promising and may well be even more useful to us than the ability to control an application with different user interfaces.

6. Conclusions

Most if not all of the multimedia environments based on Tcl/Tk, including Medusa, accept the wisdom that Tcl/Tk must only have the roles of glue, control and user interface, while the multimedia processing must be done elsewhere. This is a fundamental principle.

But we believe that delegating the media processing layer to a more efficient implementation language, while necessary, is not sufficient: distributed multimedia applications based on plain Tcl without any added structure will easily become too complex to manage and maintain.

We have shown three main areas in which we have experienced growing complexity and we have shown our adopted or planned solutions to these problems.

(1) It is not trivial to efficiently control parallel execution through a single-threaded process, but it helps to keep the Tcl side single-threaded for simplicity. The introduction of an appropriate programming construct solves some important control problems.

(2) Modules are a good first layer, but they are too low level for building large applications. There is a need for a library of reusable building blocks that are semantically high level and syntactically simple. The problems of building these are harder than those of composing Tk widgets. This is due to the data connections and the cardinality issues associated with a distributed heterogeneous system. [incr Tcl], while not in itself a complete solution, is a valuable tool which helps a lot in grouping atoms to create reusable components.

(3) The many media under which an application can present the same semantic contents to the user should not be handled on an ad-hoc basis without a unifying abstraction: the Model-View-Controller provides this.

By supporting our applications environment with the structure introduced by these abstractions we can tame the inherent complexity of distributed multimedia programs while reaping the benefits of reconfigurability and ease of prototyping that Tcl/Tk is rightly acclaimed for.

7. Acknowledgements

Many people at ORL have been working on Medusa bringing their essential contribution at many levels, from the design of the hardware and of the ATMos operating system to the networking, modules and applications layers. Everybody in the Medusa software team (Martin Brown, Paul Fidler, Tim Glauert, Alan Jones, Ferdi Samaria, Harold Syfrig and the authors) has been involved at some stage with the Tcl layer of Medusa.

Of specific relevance to this paper Tim Glauert, co-designer of the Medusa software architecture, wrote the threaded C++ API and Frazer Bennet, as well as tracking down some subtle threads bugs, patiently rebuilt our gigantic interpreter every time we needed to pull in a new release of our many extensions.

Many thanks are also due to the authors of the software that our interpreter is built out of: John Ousterhout for Tcl/Tk, Michael McLennan for [incr Tcl], Brian Smith and his team at Berkeley for Tcl-DP.

8. References

[Badges 94]
ANDY HARTER, ANDY HOPPER, A Distributed Location System for the Active Office, *IEEE Network*, Vol. 8, No. 1

[incr Tcl 93]
MICHAEL J. MCLENNAN, [incr Tcl] - Object-Oriented Programming in Tcl, *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11, 1993.

[incr Tk 94]
MICHAEL J. MCLENNAN, [incr Tk] - Building Extensible Widgets with [incr Tcl], *Proceedings of Tcl/Tk Workshop*, New Orleans, June 1994.

[Man-Month 75]
FREDERICK P. BROOKS, *The Mythical Man-Month*, Addison-Wesley, 1975, 1982

[Medusa-ICMCS 94]
STUART WRAY, TIM GLAUERT, ANDY HOPPER, The Medusa Applications Environment*, Proceedings of the International Conference on Multimedia Computing and Systems*, Boston MA, May 1994. An extended version appeared in *IEEE Multimedia*, Vol. 1 No. 4, Winter 1994. Also available as ORL Technical Report 94.3.

[Medusa-snakes 95]
TONY HEAP, FERDINANDO SAMARIA, Real-Time Hand Tracking and Gesture Recognition using Smart Snakes, ORL Technical Report 95.1

[Medusa-speech 95]
ROBERT WALKER, *An Application Framework For Speech Recognition in 'Medusa',* Third Year Project Report for the Computer Science Tripos, University of Cambridge, 1995

[Medusa-Tcl 94]
FRANK STAJANO, Writing Tcl Programs in the Medusa Applications Environment*, Proceedings of Tcl/Tk Workshop*, New Orleans, June 1994. Also available as ORL Technical Report 94.7.

[Medusa-video 95]
ANDY HOPPER, The Medusa Applications Environment, ORL Technical Report 94.12 (video). Also available as an MPEG-encoded movie that can be viewed online from our web server.

[Pandora 90]
ANDY HOPPER, Pandora — An Experimental System for Multimedia Applications*, ACM Operating Systems Review*, Vol 24, No.2 April 1990. Also available as ORL Technical Report 90.1.

[Smalltalk 90]
PHILIP D. GRAY, RAMZAN MOHAMED*, Smalltalk-80: A Practical Introduction,* Pitman, 1990.

The ORL Technical Reports are available for download on the Internet through Olivetti Research Limited's web server at
`http://www.cam-orl.co.uk/`
or via anonymous ftp at
`ftp://ftp.cam-orl.co.uk/pub/docs/ORL`