Plug-And-Play with Wires

Maximilian Ott and John Hearn
C&C Research Laboratories, NEC USA, Inc.

Plug-And-Play with Wires

Maximilian Ott *John Hearn*

C&C Research Laboratories, NEC USA, Inc.
4 Independence Way
Princeton, NJ 08540
max|jph@ccrl.nj.nec.com

Abstract --- To allow for processing of data-streams in interactive multimedia-based applications, we propose a data-flow framework. Individual processing modules are connected through ''wire'' objects. The wire assumes all responsibilities for data transfer and scheduling, which drastically reduces the complexity of the processing modules. We describe the overall architecture, module API, and design of the wire. We further discuss integration into event-driven systems, and demonstrate the flexibility of this approach with tiny, but operational application scripts.

INTRODUCTION

Data and compute intensive applications can often be divided into two distinct components: data-driven and event-driven. A data-driven component processes many operations and/or large data quantities. An event-driven component processes external control over an application such as a user interface.

Data-driven components are mainly concerned with speed and efficiency, while event-driven components are structured for flexibility and handling exceptions. Unfortunately, data-driven and event-driven components often conflict with each other by competing for compute resources. While an application is busy processing data, event driven information is not getting handled. Conversely, while an application is busy handling event information, the data is not getting processed.

Separating the components into two separate concurrent threads of execution resolves much of the delay issues. However, separation is only suitable for applications which have a loose coupling between the data-driven and event-drive components. Applications, such as a video decoder, which require a tight coupling between the data-driven and event-drive components do not generally benefit from separation.

This paper describes a framework for building data processing components which allows applications to handle large data streams while minimizing the impact on the event handling side.

BACKGROUND

Over the last two years we have developed a collection of data-driven components which is divided into three categories: producers, consumers, and processors [1]. Producer components include cameras, microphones, and VCRs. Consumer components include video display (motion JPEG), image display (extended TkPhoto), and audio playback. (Specialized producer and consumer components have also been developed to support various multimedia data storage formats.) Processing components include network transfer elements utilizing various protocols (TCP, UDP, MTP) allowing data streams to flow transparently between machines. (Network transfer elements aid research efforts targeted at distributed multimedia.) Each instance of a data-driven component is called a module.

Data streams originate at producer modules, are piped through processing modules, and terminate at consumer modules. We designed a separate unique object called a wire which controls the data exchange between connected modules (Figure 1). This has considerably reduced the inherent complexity involved when streaming data through various types of connected modules. The following code listing illustrates the use of a wire for displaying a video clip from a movie file; the multimedia version of “Hello World”:

```
set output [video .v]
pack . .v
set input [[movie "HelloWorld.moov"] \
           track -video]
wire -from $input -to $output
```

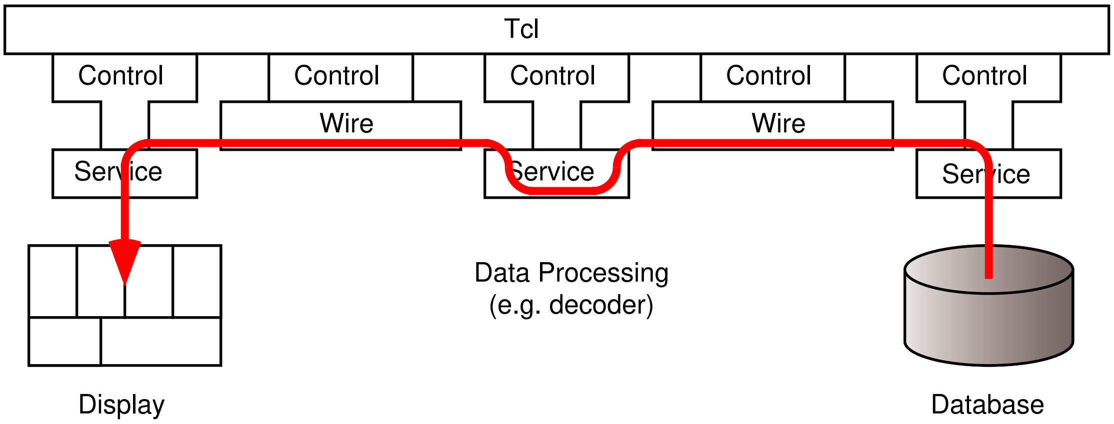

Figure 1: A wire connects modules through standard interfaces.

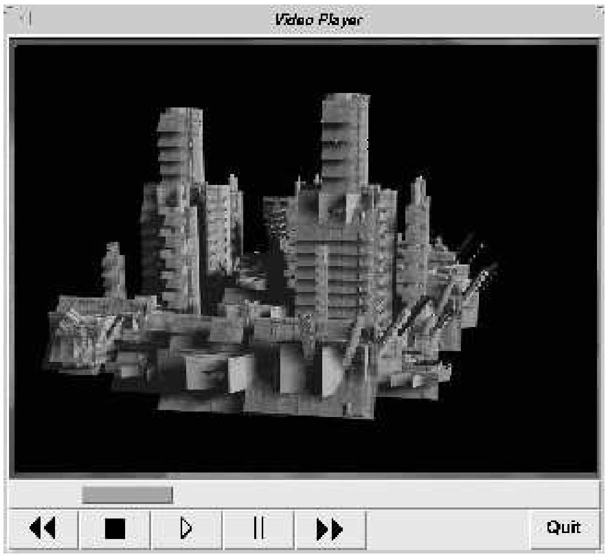

Figure 2: Snapshot of video player application.

Figure 2 shows a screen snapshot of a slightly fleshed out version of the above program.

Wires have their own control interface. A data stream can be started, stopped, resumed, and monitored. Additional functionality such as multi-casting and switching are implemented as well.

MEDIA CHUNKS AND DATA OBJECTS

Data streams are viewed as a series of packets or chunks of arbitrary size. Chunks are composed of data frames and description objects. Data frame objects simply hold pure data. Description objects hold semantic information describing the pure data held by container objects.

Data frame object sizes are influenced by the type of data they hold. A frame in a video chunk may be equivalent to a single video frame. A frame in an audio chunk may contain a few milliseconds of sound. In a mouse chunk it may contain an x and y offset. The size of a data frame is also affected by other factors such as data encoding algorithms and network transport packet sizes.

Description object sizes are generally small. The amount of information needed to describe a data set is assumed to be only a small fraction of the size of the data set.

We use the following structure to describe a chunk.

```
typedef struct {
  /* media type of chunk (audio, video) */
  mediaType type;
  /* actual data chunk */
  DataObj*  data;
  /* semantic information on chunk */
  DataObj*  info;
  bool      infoChanged;
} Chunk;
```

Chunks are often allocated and destroyed from within different modules. Therefore, we encapsulate the actual storage buffers within a structure of type DataObj. This also provides the flexibility for transparently utilizing different storage methods such as shared memory and memory mapped devices. Modules utilizing the data objects contained within chunk structures use the methods of the data object.

The data structure describing a data object is as follows:

```
typedef struct _dataObj {
  /* size of active data range */
  u_32  size;
  /* ptr to the actual data */
  VOID* dPtr;

  /* actual length of data chunk */
  u_32  length;

  /* method for changing the amount of
   * available memory.
   * Returns TRUE on success. */
  bool  (*setBufSize)(struct _dataObj* self,
                      u_32 newLength);

  /* clean up, including this structure */
  bool  (*free)(struct _dataObj* self);
}
```

This looks very much like a "poor man's" C++. Remaining with C in the context of the large body of C code of Tcl/Tk and our own modules seemed to be the right choice at the time. In the meantime we have added large modules written in C++. In hindsight, it might have been advantageous to take the plunge at that time and in fact we are currently consider a re-write in C++. However, the design will remain largely the same and we might just trade the uncertainties of pointer casting with the verbosity of C++. Currently, each object publishes its interface through a structure containing function pointers for every public method.

PORT INTERFACE

As mentioned above we want to minimize the impact on a single module and shift all the functionality of moving data between modules to the wire. The following describes the port interface provided by each participating module. Modules can provide two different types of ports, *in-ports* for receiving chunks and *out-ports* for sending chunks. When modules are created, they register their name and a list describing their ports' interfaces with a central database. When a wire connects to a port it retrieves the description of the ports it intends to connect.

The relationship between wire and modules is that of a master-slave; all activity originates from the wire. After a wire is created and the source and sink modules are identified, it is immediately in *active* mode. There is no initial synchronization between a wire and its connected modules. The wire will begin by sending a `doChunk` message to the out-port of the source module. This request can yield the following replies:

`Wi_OK`	The module had a chunk ready and returned it.
`Wi_Wait`	The module will be ready in a specified time.
`Wi_Call`	The module is not ready and requests the wire to register a callback to allow to signal the wire when the module is finally ready.

In the event of `Wi_OK` the source module also returned a valid chunk and the wire will immediately switch to delivery mode and in turn send a `doChunk` message to the in-port of the sink module.

For all other replies a port indicates that it is not ready yet and should be queried again in the future. Some modules are producing chunks at regular intervals and use `Wi_Wait` to indicate the time by which they are ready. In contrast, if the computation of a module performs in a different context, or receives data across the network, the availability of the chunk is unspecified. The `Wi_Call` reply requests the wire to register with the module through a different method, so the module can later signal the wire to resume.

The above outlined mechanism requires the following interface to be provided by every port of a module:

```
typedef struct {
  /* data transfer */
  WipDataI    data;
  /* register trigger */
  WipTrigI    trigger;
} WirePort;

typedef struct {
  /* Port's entry point to get/put a
   *  chunk. */
  WiDoProc*  doChunk;
  /* Handle provided to above procedure */
  VOID*      hdl;
} WipDataI;

typedef struct {
  /* Register callback to notify
   *   registree that the associated
   *   port is ready */
  WiTrigRegProc* set;
  /* Handle provided to above procedure. */
  WiTrigRegHdl   hdl;
} WipTrigI;
```

It should be noted that the wire is not really aware of the chunks flowing through it. It simply provides a "container" for the modules to place a chunk into. The modules are free to replace the chunk the received with another one, or even return the container empty (Figure 3).

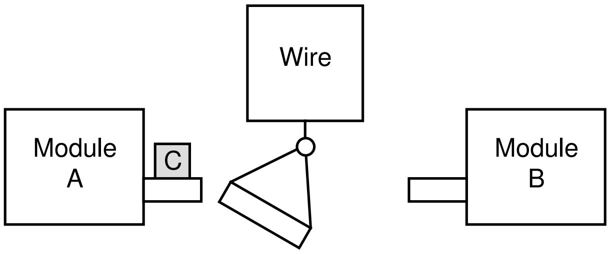

Figure 3: Wire provides a container to deliver any chunk object.

For instance, our video display module (VD) implements a ping-pong buffer in the following way: When the first chunk arrives, the VD returns an empty container. On every following delivery it will return the previously delivered chunk. It is entirely up to the modules to decide how many chunks will be allocated.

To prevent memory leaks we established the convention that ownership of a chunk is transferred to a module during the call to its `doChunk` interface. Any chunk being returned to the wire briefly becomes the property of the wire which will immediately transfer it to the opposite module.

Only when a wire is stopped and subsequently asked to expire, will it also destroy the chunk remaining in the container.

WIRE INTERNALS

The big challenge for applications with a GUI is to find the right balance between efficiency of processing and responsiveness to user input. Taking advantage of advanced features of modern operating systems, such as threads, will shift the responsibility to the operating system which does not always lead to satisfying results while increasing the complexity of the application.

In the scenario described above we can maintain responsiveness by controlling the granularity of the processing tasks. We want to point out, that while many of our applications run within exact defined deadlines we cannot enforce them but gain enough flexibility through modularity to group the members of all processing pipelines so they reach most deadlines in time, where "most" is normally a sufficient success criteria.

The basic functionality of a wire is to ask the source module for a chunk which it then tries to deliver to the source module. This task will be repeated over and over again if the wire is in *run* state.

If both modules can provide or consume a chunk whenever the wire calls, a single wire may exhaust all compute resources. For a wire to be a "fair" citizen of an application it needs to give up control regularly to allow other tasks to progress as well.

One reason for suspending a wire is a reply of `Wi_Wait` and `Wi_Call` from a `doChunk` message. In the case of `Wi_Wait` the wire will request the run-time environment to be awaken after a specified time interval and will then suspend. This service is provided by TK through the `Tk_CreateTimerHandler` function.

In the case of a `Wi_Call` reply the wire will first register with the replying port through the port's WipTrigI interface and then suspend. When the port is finally ready it will signal the wire which in turn will retry the `doChunk` message.

If both ports immediately reply with `Wi_OK` the wire will inform the local scheduler that it wants to resume, but is now ready to give up control.

Within the confines of a single process thread and the Tk event loop, the wire registers a timer callback with zero time and returns to the event loop. Any already matured timer event will be serviced before control returns to this wire.

It should be pointed out that as of version 3.6, timer events have priority over file events. That means that any free running wire will block processing of file events, especially events from the X server. A work-around is to schedule a regular `update` through a `after` command which also creates a timer event.

Wire Control Interface

Wires can be created through the `wire` command which returns the name of a new command to be used in future interactions with the new instance.

```
> wire ?
wire commands: ?-pause? \
    -from srcModule -to dstModule

> wire -from $clip -to $video
wire0

> wire0 ?
wire0 commands: start ?chunkCnt? | pause
   | running? | configure | whenDone proc
```

When a wire is created with both modules defined, it will immediately begin exchanging chunks between the connected ports. The wire can also be told to stop and will do so after the chunk, currently in transit is delivered. The number of chunks being processed can also be limited through an optional parameter to the `start` command. In both cases, a procedure registered with the `whenDone` command is called when the wire stops. For instance, this callback can be used to deactivate a "stop" button in a VCR application.

Example: A simple Video-On-Demand

To demonstrate the flexibility of the described module/wire combination we will extend the introductory "Hello World" example to a simple Video-On-Demand (VOD) application.

By replacing the producer with a network module we convert the original "VCR" into a "TV" listening on a specific network address:

```
set output [video .v]
pack . .v
set input [netModule -listen $port]
wire -from $input -to $output
```

The wire in the server program now connects the database module to a similar network module which transmit the received chunk to a specified network address:

```
set input [[movie "HelloWorld.moov"] \
           track -video]
set output [netModule \
            -connect $clientHost $port]
wire -from $input -to $output
```

As we can see the end modules are created in the same way as before and are in fact completely unaware of the additional step of transmitting the chunks across the network.

Additional Features

Recently, we added a *monitor* port to the wire which is allowed a peek at the chunk in transit. It was originally envisioned for collecting statistics on through-put performance. However, in a recent experiment we use the chunk size to control a video encoders. Whenever a chunk passes through the wire a control algorithm (implemented as Tcl script) is executed, which in turn controls a video encoder as well as the service parameters of an outgoing network connection to maintain constant video quality.

We also experimented with a multicast feature where multiple consumer modules are connected to a single wire. The wire selects the receiver from a channelId field in the chunk object (omitted in the above description). We use this feature in applications with multiple photo widgets where the images are downloaded from a remote server through a single network pipe.

DISCUSSION

The main challenge in treating multiple threads fairly is similar to the task assigned to the scheduler in an operating system. However, within a single processing context each thread needs to give up control voluntarily and has to avoid (often at great cost) to block on any system calls. The threads mechanism offered in some OSs would help greatly for integrating data-driven and event-driven computing within a single program.

The wire framework was initially designed to ease development of stream processing modules. Although, in a distributed environment a *wire object* can represent the resources associated with delivering the data from the sink to the source [3]. In our previous example, the wire could itself create network connections if it realizes that the two objects it connects to are located on different hosts. We are currently investigating that approach in a distributed networking language derived from Tcl [2].

CONCLUSION

We described a framework which allows event-driven and data-driven components to co-exist cooperatively within a single TK process. Data processing modules can be connected to each other through a uniform interface which we specified. We then described a wire object which provides control and management of data transfer between modules.

REFERENCES

[1] M. Ott, et al., "A prototype ATM network based system for multimedia-on-demand," IEEE COMSOC Workshop, Kyoto, May 1994.

[2] M. Ott, "Jodler - A scripting language for distributed applications," Tcl Workshop, New Orleans, June 1994.

[3] G. Michelitsch, M. Ott, S. Weinstein "Multimedia beyond video-on-demand," Workshop on Community Networking, Princeton, June 1994.