

Interaction Points : Exploiting Operating System Mechanisms for Inter-component Communications

Daniel G. Waddington
Bell Laboratories
101 Crawford's Corner Rd
Holmdel NJ, 07733
dwaddington@lucent.com

Ramesh Viswanathan
Bell Laboratories
101 Crawford's Corner Rd
Holmdel NJ, 07733
rv@research.bell-labs.com

ABSTRACT

This paper introduces the concept of 'interaction points' which are currently used in an experimental programming environment, the In-process Modular Programming (IMP) platform. IMP is a Microsoft COM-style platform designed specifically for building component-based applications in real-time and embedded environments, and used for research prototyping at Lucent Technologies, Bell-labs. Interaction points serve as a unified abstraction for a wide range of inter-component communication mechanisms. They allow the programmer to easily take advantage of available operating system level mechanisms such as semaphores, shared memory, spinlocks and I/O buffers, in order to realise advanced inter-component communication paradigms. They excel in supporting broadcast and multicast communication, which are often difficult to realise with point-to-point communications such as those provided by conventional program function calls. In addition to supporting operating system mechanisms, interaction points also provide a mechanism for building re-useable custom communication facilities that are not themselves, considered worthy of componentisation. Finally, the use of interaction points helps to maintain an explicit definition of communications on component-interfaces and thus avoid problems of ill specified 'hidden' component interactions that often lead to problems during third party re-use.

General Terms

Components, performance, design, experimentation, languages.

Keywords

Component engineering, interaction points, operating-system level communications, real-time, embedded, modular, network services.

1. INTRODUCTION

As software systems become progressively complex, the advantages to be gained through object oriented (OO) techniques and component programming become increasingly valuable. It is reasonably accepted that for software systems to continue to grow in both size and complexity, these techniques will need to be adopted. This is particularly evident where the software procurement process is required to be a multi-party effort, driven either by the need for specialised contributions or a need to off-load portions of the development for larger systems. In such cases, features offered by component-based programming techniques (which are distinctly different from those of pure OO), including independent extensibility and open interfacing, are vital to the procurement process.

Despite their relative broad acceptance as sound software engineering principles, the application of OO techniques and component based programming in real-time and embedded environments has not yet been significant. What application of these techniques there has been, is limited to relatively heavy-weight execution environments where resources are plentiful and temporal requirements reasonably flexible. The use of such techniques in more light-weight embedded environments, where resources are scarce and requirements more stringent, has up until now, been somewhat taboo. Some attempts have been made to apply component-based engineering approaches in the development of real-time applications, but only a partial number of concepts have been adopted [Stewart, 97].

In order to avoid problems of inefficient resource utilisation, careful consideration should be made in the design of the component model from the ground up. Functionality must be both incremental and optional, enabling the deployment of features on a per-requirements basis. Existing component technologies, such as Sun JavaBeans and Microsoft's .NET, offer a wide range of component programming features, but are nevertheless inherently demanding on their supporting environment. Furthermore, these technologies are aimed at desktop and server applications, and are therefore particularly generic in their functionality. The requirements for component development in a general-purpose domain are not transitive to real-time and embedded environments. Nevertheless, we believe that the advantages of component engineering can be leveraged by embedded and real-time systems, providing that the component model is suited to a specific set of requirements. For real-time and embedded systems these include, but are not limited to, the following:

- **Efficient Communications** – shortest path communications, without unnecessary intermediate mechanisms that may be imposed because of generic requirements (e.g. CORBA support for distributed transparency often mandates some form of marshalling across all calls, even if they are local).
- **Deterministic Communications** – in a real-time environment, intra-component and component-environment interactions must be at least temporally bound.
- **Operating Systems Level Communications** – support for interaction paradigms which take advantage of the underlying OS level mechanisms.
- **Tailorability** – embedded and real-time environments are likely to offer varying memory models, thread and process models, file systems, devices, etc.
- **Scheduler Awareness** – components within real-time systems must be aware of timing constraints and fit into the system's scheduling model.

Of these requirements, this paper focuses upon the provisioning of operating system level communications support within the component model. The work presented is part of the ongoing Morpheus project at Bell Laboratories, Holmdel. The Morpheus project aims to combine 'good' engineering practice with high level abstractions and software analysis tools to aid the development of component-based applications, coupled with formal modelling to further compositional reasoning and verification. The resulting platform is hoped to provide a basis for the development of both 'reliable' and 'predictable' network applications and software services (such as found in edge-routers, caches and firewalls) through re-useable software components. Of course, we do not envisage the use of software-based processing where performance is critical, such as in the network core, only where software is able to meet the required basic level of performance.

The rest of the paper is broken down as follows. Section 2 reviews the advantages that component-based engineering can offer and makes some discussion about the requirements of component platforms. Section 3 provides a first look at the concept of interaction points which form the main focus of the paper. Continuing, Section 4 presents additional details of the IMP platform and the engineering of interaction points. In Section 5 we make some evaluation of interaction points and offer some comparisons with conventional component solutions. Section 6 briefly reviews some relevant related work in the area of component software for real-time and embedded applications. Finally, Section 7 concludes the paper with a summary of the work.

2. COMPONENT-BASED SOFTWARE ENGINEERING IN REAL-TIME AND EMBEDDED ENVIRONMENTS

A component, as defined by [Szyperki, 99], is a software unit of independent deployment, subject to third-party composition, with no persistent state. Component engineering involves the use of both a component-based design strategy, through a component *model*, and modularised software technology, through a component *framework*. The component model defines a set of component types, their interfaces, the construction conventions, and some level of specification defining allowable component compositions (or patterns) and inter-component interactions.

A component framework, from the definition given in [Bachman, 00], provides a variety of runtime services to support and enforce the component model. A more complete overview of component-based software engineering can be gained from texts [Bachman, 00] and [Szyperski, 99].

In brief, secondary advantages which can be gained from component-based engineering that are particularly relevant to real-time and embedded applications, include the following:

- **Software Verification and Validation** – As software systems continue to expand, reliability becomes increasingly difficult to predict. Component-based approaches help address this by allowing the verification and validation of finer grained subsystems and atomic software units, together with imposed design constraints. In turn this allows some level of reasoning about the complete system, thus providing enhanced predictability. For instance, one could reason about the timing constraints demanded by a given composition in order to give an indication of feasibility and scalability of the system as a whole.
- **Tailored Provisioning** – Many of today's software systems rely on the premise of 'provide for all occasions'. That is, functionality is provisioned for a set of generic requirements. Consequently, in order to address such a broad set of requirements, a large number of software systems become bloated through over-provisioning. This problem is particularly evident in COTS operating systems and applications. Component-based engineering promises to alleviate this problem by allowing 'tailored' provisioning, i.e. you get what you need. This is particularly useful in embedded environments where resources are particularly scarce.
- **Run-time Flexibility** – This again relates to the ability of systems to address changing requirements. On a finer level of granularity, requirements may change during the run-time of the system. Component-based engineering lends itself to supporting run-time changes in the software system through adherence to a uniform model coupled with the inclusion of appropriate mechanisms at unit interaction boundaries (such as thread locking and synchronisation).

The requirements of a component model and component framework are generally driven by the application domain. For example, let us consider Java. The framework for Java components is the Java Virtual Machine (JVM). The JVM provides services for instantiating and managing Java classes, as well as extensive run-time type checking and introspection, together with a multitude of other run-time services. As a consequence the JVM can be considered relatively heavy-weight in terms of its execution demands, that is compared to what is available in most embedded environments. Furthermore, the feature set provided by a COTS JVM is not suited to real-time applications. For example, Java generally does not make any consideration of temporal constraints or alternative interaction paradigms (alternative to method calls).

The observation we are trying to make is that the features offered by a given component technology must be tailored to the requirements of the application domain. Thus, the use of component-based engineering for the development of network services demands specific attention to performance, efficiency and a close coupling with the underlying real-time operating system.

3. INTERACTION POINTS

Components (implemented as dynamic units of code, or dynamic libraries) are composed into applications (implemented as processes). Our proprietary component platform, used for the development of prototype applications for network processing, is the In-process Programming Model (IMP) [Waddington, 01]. IMP specifies, among other things, a model for component communications. The model has been designed with a conscious concern for its intended use as a platform for real-time and embedded applications and introduces a novel construct for inter-component communications that allows the developer to exploit operating system level facilities for component communications; this is 'Interaction Points'. This section motivates and illustrates the advantages to be gained from this new construct.

Many existing component technologies model inter-component communications as request/reply or one-way request interactions, which are ultimately engineered as program function calls. The probable reason for this is that function calls are relatively portable and also well understood. The existence of other component interaction paradigms is nevertheless, fairly well recognised. For example, the ISO RM-ODP model breaks down interactions into finer levels of granularity and introduces the concept of signals and flows [ISO, 98]. In addition, the majority of existing COTS operating systems include a rich set of software based interaction mechanisms. These include synchronisation services (semaphore, mutexes, write locks, spin locks, condition variables), control services (sleep, wake, kill, etc.), device communications (network sockets), soft interrupts, messages, events and more. Nevertheless, many existing component models strictly limit communications across interfaces to methods (program calls) and properties (queried through program calls). As a result, a developer has two choices. First, he/she may decide to avoid the use of any ‘advanced’ interactions outside of the component. This means that mechanisms, such as those given above, need to be re-engineered with pure call based interactions and some form of indirection, which often leads to additional complexity in design and implementation, and often poor performance. The alternative choice is to ‘violate’ the component model and leave some component dependencies inexplicit, which inevitably leads to failure during third party re-use.

The nature of interactions in real-time or embedded systems, differ from that found in conventional desktop/server systems. This is principally because of close coupling with operating system communication mechanisms and underlying hardware devices. Interaction points provide a unified approach to the incorporation of ‘advanced’ communications between components, and also formalise the process of binding such communications. In addition, their use maintains that dependencies remain explicitly defined as part of the component interface model. This point is crucial since in most cases the developer of component-based applications in real-time systems must be able to use the underlying real-time constructs for the application to be effective (particularly scheduling and synchronisation APIs). Interaction points also allow the component developer to effectively re-use proprietary communication mechanisms which are not themselves worthy of componentisation (i.e. adding reference counting and interfacing functionality).

Figure 1 illustrates both a conventional and an interaction point based implementation of simple event driven communications, in which a clock (server) generates an event, which is multicast to one or more counters (clients). In the conventional implementation, shown left, a third party event manager is needed to dispatch events. This event manager can use the component framework to access operating system event services, and because events processed by the operating system are only dispatched to the event manager itself, the component model is not violated. Nevertheless, complexity arises through the need to establish a number of call-backs between the clock, event manager and the counters, due to the asynchronous nature of an event driven paradigm.

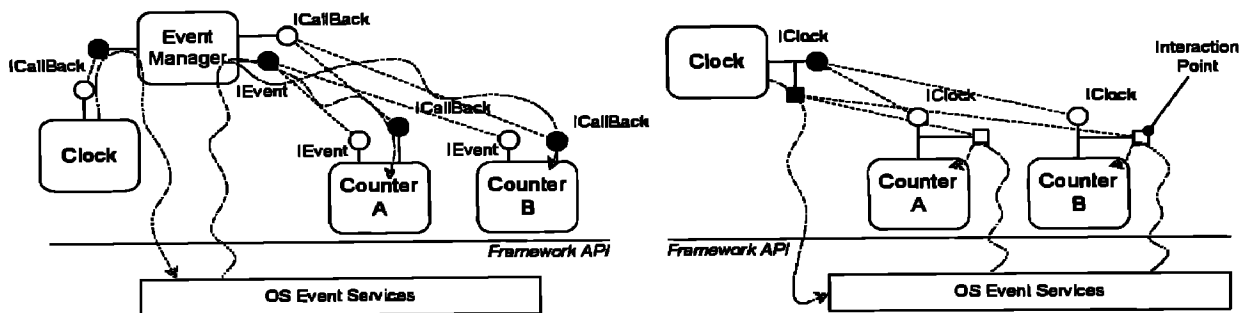


Figure 1. Inter-component events implemented conventionally (left) and through Interaction Points (right)

The interaction point based implementation simply requires each counter to bind to a single clock interface, that exports an ‘event’ interaction. In this instance, clocks and counters interact directly through the framework services, and hence operating system mechanisms. Model integrity is maintained since all points of communication are explicitly captured by the component’s interface. This alternative implementation lends itself

Table 1. Example Interaction Point Types

Interaction Class	Description
Synchronisation Mechanisms	Semaphore, mutexes, spinlocks, events and other thread synchronisation techniques.
Shared Resources	Shared cache, main memory, I/O memory and persistent storage resources.
Control and Scheduling	Thread control and management, process control and management. Resource dispatching.
Hardware Interfaces	Interrupts (hard and soft), I/O ports and mapped memory locations.
Application Data	Network packets, protocol messages (e.g. IIOP), multimedia frames, etc.

4. IMPLEMENTATION

4.1 Overview of IMP Platform

This section overviews the IMP platform in order to give the reader a fuller context of how interaction points are used within our own prototyping platform. At a fundamental level, IMP makes use of Microsoft COM's approach to interface representation, universal identification and object implementation [Microsoft, 95]. Nevertheless, use of COM technology is limited to the design; IMP does not make use of any part of the COM development or runtime environments. IMP is also designed for POSIX based environments.

IMP interfaces are based upon the use of pure virtual function tables in the same manner as Microsoft COM. This means that although IMP is primarily designed with the use of C++ in mind, providing that a programming language is able to support virtual function tables, it could in theory be used for IMP. Interfaces are extremely lightweight and do not incorporate extensive marshalling or intermediary communications (except of course where this form of intermediary is required for the specific application). Programming calls, which are considered the basic form of interaction, only involve pointer de-referencing and processor function CALLs [Ellis, 91].

Similar to Microsoft COM, the IMP component model is based on the use of potentially more than one interface on any given component. Each component must provide a basic 'bootstrap' interface which is known as `IUnknown`. We have extended the COM defined `IUnknown` interface to include support for both required and provide interface directionality. In brief, the methods `AddRef` and `Release` are used for reference counting, whilst `QueryInterface` and `QueryRequiredInterface` are used to get hold of interface pointers and interface pointer receptacles, respectively. Because a component may not necessarily depend on other interfaces, the `QueryRequiredInterface` function is not a pure virtual function, and hence is optionally implemented.

```

interface IUnknown
{
public:
    virtual HRESULT QueryInterface(REFIID riid, void ** ppvObject)=0;
    virtual ULONG AddRef()=0;
    virtual ULONG Release()=0;
    virtual HRESULT QueryRequiredInterface(REFIID riid, void **& pvObject) { return
E_NOTIMPL; }
};

```

The semantics of IMP `IUnknown` interface are not strictly defined or uniform. For instance the general semantics of `IUnknown::QueryInterface` is to assign to `ppvObject`, a pointer to the requested interface which is attached to the current component instance. However, it would be quite acceptable to use this method to assign a pointer corresponding to the requested interface attached to a new component instance. Hence, even the semantics of

`IUnknown` cannot be assumed. Whether this lack of uniformity is seen as good or bad is a subject of debate, but in the IMP component model a default semantics is assumed bar the existence of explicit specification inferring otherwise. Finally, one issue that may be raised at this point is, how does a client get hold of the initial `IUnknown` interface for a given component? The answer, is through the component entry point `CreateInstance` which all components must provide. The process of instantiating a component and accessing an interface is in brief as follows. As interfaces are uniquely identified, so too are components; each component has an associated CLSID (derived from the term class identifier). The client uses the CLSID, which may be inherently known or referenced from some name lookup facility, to instantiate a component through the framework API's `ImpCreateInstance` call. This call in turn looks up the associated component source (usually, but not necessarily a dynamic library) and executes the entry point function `CreateInstance` whilst passing an interface receptacle. Although the client generally requests `IUnknown` of the component, some other more specific interface may be requested. In our current prototype, the lookup facilities component-to-code mappings are implemented as a simple persistent file. This is considered more portable and embeddable than Microsoft's proprietary registry solution. IMP also allows the use of a remote registry for target platforms which do not have a local file system.

4.2 Programming with Interactions Points

Interaction points are realised on IMP component interfaces and act as placeholders for some communications mechanism that is either required or provided by the interface. The point itself (generally implemented as an object) is exposed through a conventional function call (as part of an interface) which is used to 'bootstrap' the binding across interaction points. Each interaction type (associated with an interaction point) is uniquely identified in a similar manner to the use of UUIDs to identify interfaces. This unique identification of interaction types helps to prevent misconstrued semantics between independently built components. The code excerpt given below defines an IMP interface `IDummy`, that provides two conventional methods, `start` and `trigger`, and one interaction point named `port_data_in`, of type `CLSID_DummyInteractionPoint`. It also requires an interaction point named `port_data_out` of the same type. The macros `REQUIRED_INTERACTION` and `PROVIDED_INTERACTION` declare conventional method which are called at bind time to access the interaction points. When a 'requirer' queries a 'provider' for an interaction point, both the correct name and type must be used. This helps to ensure that the requirer is getting what it is expected.

```
// IDummy interface declaration in IMP
//
interface IDummy : public IUnknown
{
public:
    virtual HRESULT start()=0;
    virtual HRESULT trigger(const char * msg)=0;

    PROVIDED_INTERACTION(port_data_in, CLSID_DummyInteractionPoint);
    REQUIRED_INTERACTION(port_data_out, CLSID_DummyInteractionPoint);
};
```

The following macros are used to simplify the definition of interaction points in component interface declarations. Note that each defines a different role, required, provided and both (dual).

```
// code from IMP.h system file
//
#define REQUIRED_INTERACTION(X,Y) virtual HRESULT X(InteractionPoint *)=0
#define PROVIDED_INTERACTION(X,Y) virtual InteractionPoint * X()=0
#define DUAL_INTERACTION(X,Y) virtual HRESULT X(InteractionPoint *&)=0

#define REQUIRED_INTERACTION_IMPL(X,Y) HRESULT X(InteractionPoint * Y)
#define PROVIDED_INTERACTION_IMPL(X) InteractionPoint * X()
#define DUAL_INTERACTION_IMPL(X,Y) HRESULT X(InteractionPoint *& Y)
```

Interaction points are generally implemented as C++ objects which must inherit from the base class `InteractionPoint`. This base class provides the unique identifier constant and a method `QueryInteraction` which serves a similar purpose to `QueryInterface`, but for interactions as opposed to interfaces. The component developer must implement the boot strap mechanisms associated with each interaction point as follows:

```
// port_data_in - provided interaction point of type CdummyInteractionPoint
PROVIDED_INTERACTION_IMPL(CDummy::port_data_in) {
    return dynamic_cast<InteractionPoint *>(&m_providedip);
}

// port_data_out - required interaction point of type CdummyInteractionPoint
REQUIRED_INTERACTION_IMPL(CDummy::port_data_out, ip) {
    if(ip==NULL) return E_UNEXPECTED_ROLE;
    // otherwise get hold of the type of interaction we expect
    if(ip->QueryInteraction(CDummyInteractionPoint::ID, (void **)&m_requiredip)!=S_OK)
        return E_UNEXPECTED_INTERACTION;
    return S_OK;
}
```

The source code above shows the use of `m_providedip` and `m_requiredip`. These represent an interaction point class instance and class pointer respectively. The interaction point class is the crux of the implementation of the interaction point and contains the required resources for the given communications.

```
// Implementation of an Interaction Point Class in IMP
//
class CQnXInterrupt : public InteractionPoint {
public:
    INTERACTION_GUID; // uniquely identifies the type

private:
    int m_id, m_interrupt;

public:
    //////////////////////////////////////
    // executed by IP provider
    CQnXInterrupt(CQnXInterrupt * parent):m_id(0) {
        m_interrupt = parent->m_interrupt;
    }

    CQnXInterrupt(int intr):m_interrupt(intr), m_id(0){}

    //////////////////////////////////////
    // executed by IP client
    int AttachHandler(const struct sigevent * (*handler)(void *area, int id)) {
        m_id = InterruptAttach(m_Interrupt, handler, NULL, 0, _NTO_INTR_FLAGS_END);
        return m_id;
    }

    int DetachHandler() { InterruptDetach(m_id); m_id=0; }
}
```

Interaction points provide sufficient resources to communicate across a given paradigm. This not only includes shared resources, such as the interrupt number in the previous excerpt, but may also includes functional support to both aid and constrain use of the interaction point. For example, an interaction point based on the use of proprietary semaphores, may include a POSIX style semaphore API which can be directly used by the component developer. Furthermore, there is no reason why interaction points cannot be standardised and re-used across independent component developments.

4.2.1 Using Interaction Points from a Client Perspective

Interaction points are at a similar level of granularity to that of method calls. Thus an IMP component interface maintains one or more conventional methods or interaction points. As previously mentioned, each interaction point is associated with a unique identifier (GUID). In order that a client get hold of an interaction point on a given component interface, the interaction point identifier (which generally relates to type) must be known a priori. This is the same approach used by Microsoft COM interfaces. The interaction point's GUID is either gained through a definition in some header file associated with the component, or alternatively, one can use the IMP metadata support which allows static and dynamic interface interrogation. The following code excerpt illustrates the use of interaction points as defined in the previous excerpt.

```
// client-side use of interaction points
IDummy * pItf;
CQnXInterrupt * pIntPoint;
HRESULT hr;

// first get hold of the component's interface
//
hr = ImpCreateInstance(CLSID_MyComponent, IID_IDummy, (void**) &pItf);

// assuming we know the class of the interaction point
//
pItf->port_data_in->QueryInteraction(CQnXInterrupt::ID, (void **) &pIntPoint);

// now we can use the interaction point
//
pIntPoint->AttachHandler(myHandler);
```

4.3 Example : Interaction Points for Network Packet Processing

This section provides a detailed example demonstrating the use of interaction points for a specific packet processing application, implemented in our test bed. The application, which makes up part of an IP tunnelling device, is made up as follows. The system incorporates a number of ingress and egress network interfaces. Each network interface essentially consists of hardware transmit and receive FIFO queues, mapped control registers and interrupt generation¹. Packets arriving on an ingress network interface are transferred into a FIFO queue and an interrupt is generated to signal that packet data is awaiting processing. Each network interface is directly serviced by a single packet receiver which is implemented in software. The packet receivers are responsible for transferring packet data from the associated network interface and copying the packet from the hardware FIFOs into a main memory queue, in order to await processing by a packet processors. Unlike the relationship between packet receivers and network interfaces, packet processors can choose which packet queues to service according to their configuration. However, each packet is only processed by one packet processor. Packet processors are responsible for carrying out 'processing tasks' on packets, which include transformation, encapsulation and route determination. Once a packet has finished being processed by a packet processor, it is taken by a packet sender for dispatch to an egress network interface. Furthermore, any packet sender can service any awaiting packet. Packet senders process packets of any type.

Figure 3 illustrates the topology of the packet processing application. The critical point of this application is that the processing activities of the packet receivers, queues, processors and senders are strictly decoupled. By this we mean that the packet processors and packet senders are asynchronous, i.e. only a packet processor can determine when it is ready to process a packet from a queue, and likewise only a packet sender is able to determine when it has completed its current activity and is ready to send another packet from packet processor. Also note that there is no egress queuing since this is not strictly required.

¹ Based on the 3Com 90xC Fast Ethernet chipset.

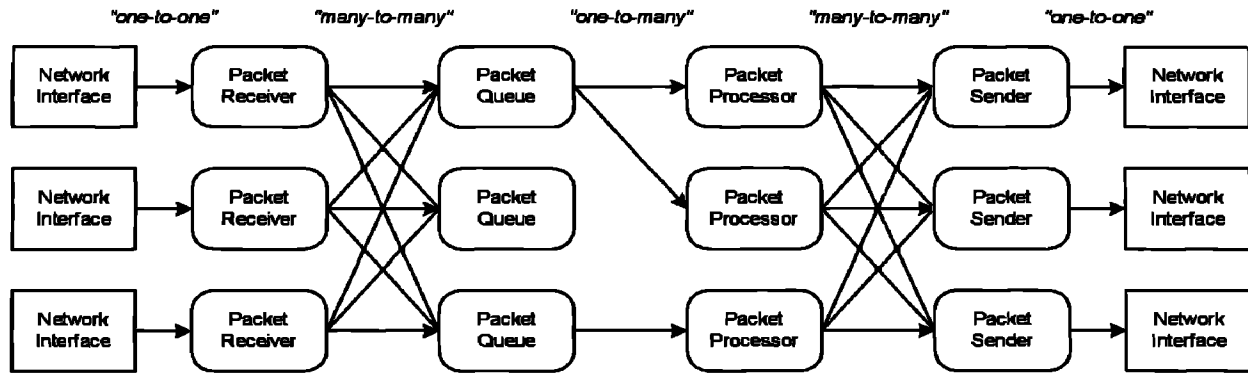


Figure 3. Packet Processing Scenario

One might suggest that asynchronous processing of this nature is not desirable in a real-time system. However, in this case it is necessary since both packet processing and packet sending are non-deterministic, and in fact processing time is determined on a per packet basis. It might also be suggested that a single packet processor be used to service all receivers and senders. Whilst in fact this would be possible, in our system we desire the ability to dynamically introduce new packet processors into the system and remove packet processors from the system without halting its execution. Finally, the use of multiple packet processors helps to realize parallel processing and allow packets to be processed irrespective of any blocked processors.

4.3.1 Application Component Architecture

Packet receivers, queues, processors and senders are all implemented as individual software components. Describing the component composition from left to right, assuming data flows in this direction, the first components are the packet receivers. Each packet receiver maintains a receiving and sending interface. The receiving interface, `INicRx` (Network Interface Card) provides two interaction points, one that services the appropriate hardware interrupt for the NIC, and another which transfers data from the FIFOs into main memory. The interrupt handler is thus provided by the packet receiver component. Obviously it must still behave strictly as a handler, only carrying out minimal processing through a restricted instruction set. In this instance, the interrupt handler transfers the contents of the waiting FIFO into main memory where it can be handled later. After the transfer, an event is set that is used to wake up a worker thread which then completes the processing, outside of the interrupt context. The FIFO to main memory transfers are initiated through the `IP_NicDMA`, which uses mapped control registers on the network interface hardware [3Com, 99].

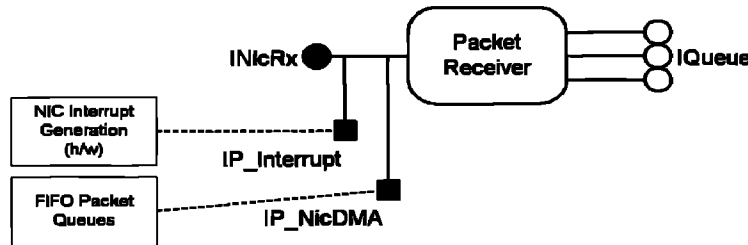


Figure 4. Packet Receiver

The worker thread completes the processing by adding the new packet to the appropriate queue via an invocation on `IQueue`, provided by a packet queue component. Through this invocation, the packet receiver passes to the packet queue the data structure information. There exists one queue per packet type and therefore the packet receiver must parse the packet header before inserting the packet to the appropriate queue. There is no need to perform an additional copy, since adding the buffer to a queue simply entails altering the queues linked list

structure. New packet types, and hence packet queue components, are not required to be run-time configurable, and therefore conventional interface re-binding is acceptable.

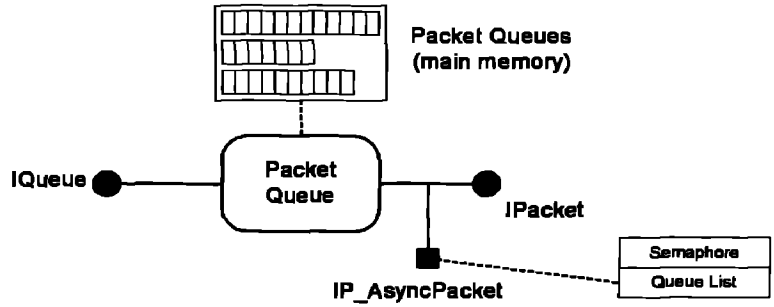


Figure 5. Packet Queue

Access to packets that are ready for processing is arbitrated through semaphores. Once a packet has been added to a packet queue, through a call on `IQueue`, the packet queue component increments its semaphore. Packets are serviced from packet queues via the `IP_AsyncPacket` interaction point. This essentially maintains the semaphore for the queue and the linked list of packets currently in the queue. As part of the code which can be compiled into the component, this interaction point also provides support for indexing fields and adding/removing elements from the buffer chain.

Packets waiting in packet queues are processed by one of the available packet processor components. Each packet processor maintains a single thread of execution for each queue that it services. Generally, each packet processor only handles a single type of packet and thus only services one given queue via a single thread. Each thread is either actively processing a packet, or waiting on the arrival of a new packet and is blocked on the respective semaphore. Obviously the semaphore mechanism is sequenced and therefore access to packets is dependent upon other waiting packet processors.

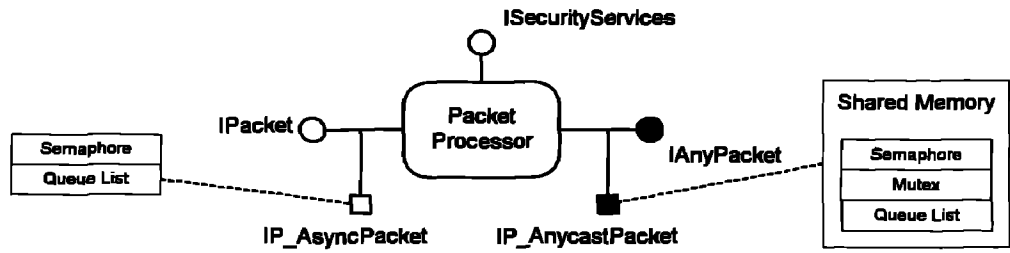


Figure 6. Packet Processor

The packet processors, as the name suggests, are responsible for performing processing on the packets. In this application packet processing may include data transformation (e.g. header translation, NAT, encryption, authentication) and encapsulation/decapsulation (e.g. tunnelling). Packet processors only handle one packet at a time and may employ sub-systems for some part of the processing. This is particularly useful for security encryption and decryption, which often needs to be done on a specialised hardware sub-system. The existence of multiple packet processors helps to increase efficiency by disallowing packets which are being processed by a sub-system, to cause blocking on the rest of the processing activity.

The packet processor, advertises the availability of a packet to the packet senders through the `IP_AnycastPacket` interaction point which enables any available packet sender to take and dispatch the packet. This interaction point uses a shared semaphore to arbitrate access to the queue list (protected via a mutex). The packet processor adds the ready packet to interaction point's queue list and then increments the semaphore.

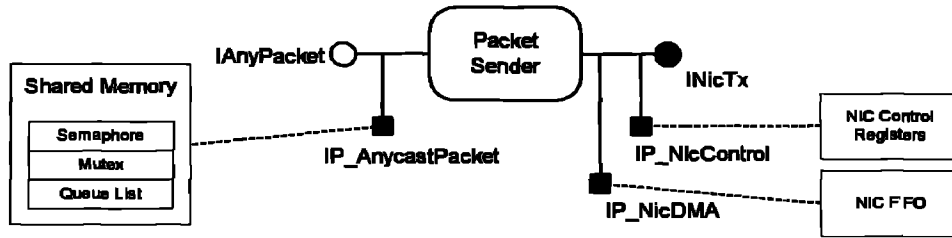


Figure 7. Packet Sender

The packet sender component uses NIC hardware registers to initialise DMA transfers of packets from the main memory to the FIFO buffers. Once a packet has been sent, the main memory packet buffers are released and the packet sender then returns to await a new packet. However, the packet sender will not attempt to take another packet until it has received transmit acknowledgement from the egress network interface. Thus, the packet sender remains blocked whilst waiting for this confirmation.

5. EVALUATION

We now make some evaluation with respect to the example given in Section 4.3. Figure 8 shows the packet flow from an operating system perspective and illustrates the sequence of interaction mechanisms involved in the end-to-end communications.

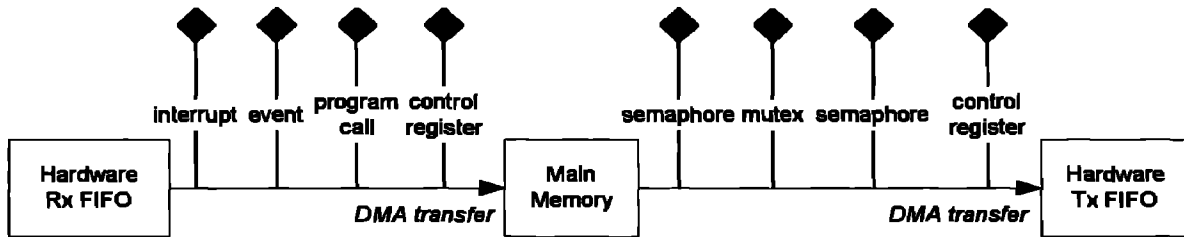


Figure 8. Operating System Mechanisms Involved in Packet Flow

The use of interaction points in the example packet processing application allows the following gains to be made:

- i. Dynamic configuration is greatly simplified. Communications between the packet queue and packet processor, and the packet processor and packet senders are realized through shared resources. This means that new components can be dynamically added to the system without the need to perform rebinding.
- ii. Overall efficiency is increased because we do not have to construct multicast and broadcast paradigms from conventional unicast program call, and thus additional indirection is not required.
- iii. The architecture is greatly simplified by avoiding conventional call-backs and polling.

It is important to note that the methods provided by the interaction point classes are only called during the initial bootstrap process, they are not involved in the packet processing and therefore no additional overhead is incurred through the use of interaction points.

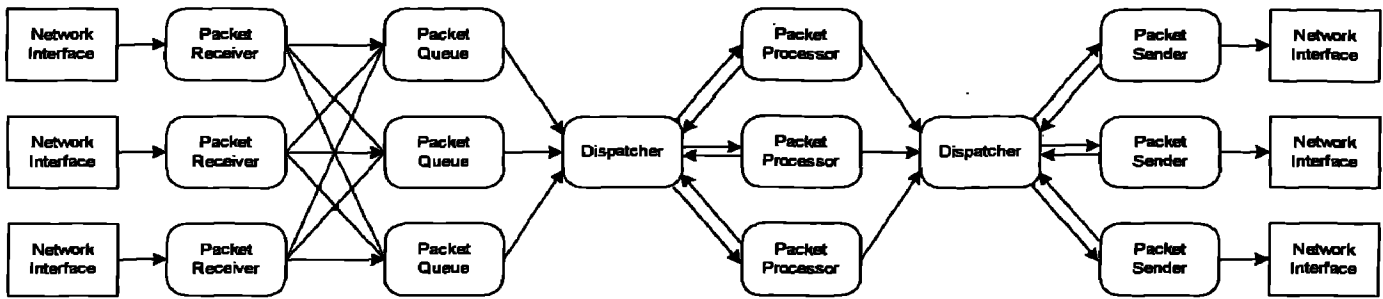


Figure 9. Alternative Method-call Based Implementation

The figure above shows a likely alternative program-call based implementation of the packet processing application without the availability of interaction points. The lack of interaction points means that additional 'dispatcher' components are required to facilitate the desired communication paradigms. Packet queues register the availability of packets to the dispatcher with a method call on the dispatcher interface. The dispatcher then forwards the packets to a selected packet processor via a call back (i.e. packet processors register their availability by calling the dispatcher). This re-design in effect introduces six method calls per packet (three per dispatcher) and certain synchronisation requirements on the dispatcher data structures². On an Intel Pentium II processor, each additional component method call, implemented as a C++ virtual function call, typically consists of the following machine instructions:

Dump of assembler code for component method call:

```

sub    $0x4, %esp                -- set up stack frame
mov    0xfffffe4(%ebp), %eax
mov    (%eax), %eax
add    $0x8, %eax
pushl  0xfffffe0(%ebp)          -- initialise parameters
push  $0x0
pushl  0xfffffe4(%ebp)
mov    (%eax), %eax             -- dereference interface pointer
call  *%eax                     -- call method
add    $0x10, %esp              -- clean up stack

```

Dump of assembler code for method implementation int method(int, void *):

```

push  %ebp                      -- save stack frame
mov   %esp, %ebp                -- unpack parameters
mov   0x8(%ebp), %eax
mov   0xc(%ebp), %eax
mov   0x10(%ebp), %eax
...                                     -- do work
mov   $0x0, %eax
pop   %ebp                      -- restore stack frame
ret                                     -- return from program call

```

The above machine instructions are executed over approximately 105 processor cycles. The exact number of processor cycles is difficult to calculate since the number of clock cycles is strictly dependent upon the number of

² Access to the dispatcher data structures must be synchronised. The overhead of this is comparable to the overhead of using the semaphore mechanisms in the interaction point based example, and therefore these are not included in the overall calculation.

parameters, use of stack frames, the processor instruction set and compiler optimisations. Furthermore, pipelining techniques, as found in the Intel Pentium architecture may also results in a lower instruction-to-clock ratio, whilst other factors such as cached code misses may increase this ratio [Intel, 99a].

One might observe that the interaction point implementation using semaphores to synchronise across multiple threads, incurs an additional thread context switch overhead. In most real-time systems, such as VxWorks, thread switches are highly optimised for performance and therefore their cost is negligible. The overhead arises because of the use of 'decoupled' interactions across independently executing threads, which are suited to rebinding. If this ability to perform dynamic rebinding is not a strict requirement, then the alternative approach becomes more feasible. However, if this were the case, the interaction points implemented in the example would be replaced by customised 'JMP' interactions, which would result in better performance. Nevertheless, we believe that the use of interaction points will always be equal or more efficient than conventional program call implementations.

5.1 Observations on Performance

As previously discussed, the performance disadvantage resulting from the additional program calls is dependent upon the nature of the program call, such as the number of parameters, how the stack is managed and whether or not registers are used instead of a stack. However, to give the reader an idea of the potential performance gains, we review a theoretical calculation, which examines the performance cost of a 105 clock overhead, for each additional method call in our example application.

```
Based on a 500 Mhz processor system handling data at 300 Mbps.
Packet size is based on Ethernet frames of 1500 bytes.

Packet throughput is 300 Mbps = 39321600 bytes per second

=> 39321600/1500 = 26214 packets per second

=> 500,000,000 / 26214 = 19073 cycles per packet

Program call overhead ~ 105 processor cycles (taken as an approximation)

=> additional overhead in given application is 105 * 6 = 630 cycles per packet.

-> total overhead per packet = 630/19073
    = 3.3%
```

Thus from the above calculations one can see that by introducing only 630 additional processor cycles per packet (a mere 0.00126 microseconds) some degradation in performance is evident. Although the performance advantage to be gained through the use of interaction points may not be considered critical to the success of the application, their use can help to tighten performance in the data path, particularly when the component processing chain is lengthened.

6. RELATED WORK

Significant contributions to component research in real-time systems has been made by Stewart et. al. Their work has led to the design of the Port-based Object (PBO) framework aimed at developing real-time control systems [Stewart, 00]. The PBO is based on the principle of composing a number of independent processes (i.e. components which are not directly dependent on functionality provided by other components) through data-driven coupling. Consequently this model minimizes inter-component dependencies and thus makes integration of component into a system more straight forward. Data interactions are driven through the notion of input and

output 'ports'. Ports offer asynchronous and variable input/output, and are implemented through C language function tables (in a similar vein to the C++ virtual function tables used in IMP). Each component maintains a number of ports together with access to configuration constants, which parameterise a component's behaviour, and 'resource ports' which abstract hardware I/O exchange. The resulting framework offers a good level of performance with reasonable pluggability, and lends itself particularly well to dynamic reconfiguration. However, the PBO framework does not permit arbitrary interfaces and mandates a tightly constrained compositional model. It also does not provide any form of component introspection and therefore its independent extensibility is questionable. Finally, the PBO approach does not allow the developer to use operating system level communications to facilitate inter-component interaction.

The two pseudo standard component platforms OMG CORBA and Microsoft .NET/COM+ make some attempt to allow the developer to make use of more advanced communication paradigms [Microsoft, 97][OMG, 98]. Both architectures make use of additional services which are comparable to the dispatcher components discussed in the previous section. They provide services for event handling, messaging queue and connection points (which equate to call-backs). Neither platform consider low level operating system mechanisms primarily due to the need for portability. Aside from the standards tracks, some research has been done on improving these models for multimedia communications by integrating mechanisms into the platform to support continuous interaction paradigms [Coulson, 96][Dang Tran, 95][Donaldson, 98]. Nevertheless, these solutions focus on supporting continuous media, irrespective of the available underlying operating system mechanisms.

Finally, work on the Click Modular Router, at the Laboratory for Computer Science, MIT, is looking at the use of component-based software for network router applications [Morris, 99]. The project uses a proprietary component model, based on C++, to realise a modular router. Components in the router are responsible for a number of network processing functions, including packet forwards, classification, header transformation, field handling and routing queries. Each router component can have any number of 'ports' which are directional network packet exchange points. Ports are implemented as conventional C++ functions such as `void push(int, Packet *)` and `Packet * pull(int)`, and operating system specific features are generally avoided.

7. SUMMARY

We have introduced the concept of interaction points as a technique for supporting 'advanced' inter-component communications in real-time and embedded systems. The focus has been made on real-time and embedded systems where performance is critical, and the use of underlying operating system features often vital to the success of the application. There is of course no reason why interaction points cannot be used in other domains also. We have discussed the implementation of interaction points in the IMP programming environment and demonstrated their use in an example packet processing application. Finally, we have reviewed the potential gains to be made through the use of interaction points from both architectural design and performance perspectives.

Interaction points help to bring the advantages of component-based engineering into the real-time and embedded systems domain. Until now the use of component technologies in these domains have been somewhat taboo, often due to impracticability of supporting run-time and other performance constraints. We believe that interaction points go some way towards increasing the feasibility of component use in such systems, something fundamental to the progression of software engineering in this domain.

8. ACKNOWLEDGEMENTS

We would like to thank Wind River Systems for their support of the Bell-labs Morpheus research project.

REFERENCES

[3Com, 99] 3Com Corporation, "3c90xC NICs Technical Reference", Technical Document 89.0931.00, September 1999.

- [Bachman, 00]** F.Bachman, L.Bass, C.Buhman, S.Comella-Dorda, F.Long, J.Robert, R.Seacord and K.Wallnau, "Volume II: Technical Concepts of Component-Based Software Engineering", Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, May 2000.
- [Coulson, 96]** G.Coulson and D.G.Waddington, "A CORBA Compliant Real-Time Multimedia Platform for Broadband Networks", in Proceedings of the International Workshop on Trends in Distributed Systems (TreDS'96), Springer Series LNCS, Vol. 1161, pp.14-29, June 1996.
- [Dang Tran, 95]** F.Dang Tran, V.Perebaskine, "TORBoyau: Architecture and Implementation", Technical Report, NT/PAA/TSA/TLRJ4587, CNET, Centre Paris, 38-40, Rue du General Leclerc, Issy-les-Moulineaux, France.
- [Donaldson, 98]** D.Donaldson, M.Faupel, R.Hayton, A.Herbert, N.Howarth, A.Kramer, I.MacMillian, D.Otway and S.Waterhouse, "DIMMA - A Multi-Media ORB", Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Process, Middleware '98, pp. 141-156, The Lake District, UK, September 1998.
- [Ellis, 91]** M.Ellis and B.Stroustrup, "The Annotated C++ Reference Manual", ISBN 0-201-51459-1, Addison Wesley Publishing Company, May 1991.
- [Intel, 99a]** Intel Corporation, "The Intel Architecture Software Developer's Manual", Volume 1, Basic Architecture, Document 243190, 1999.
- [Intel, 99b]** Intel Corporation, "The Intel Architecture Software Developer's Manual", Volume 1, System Programming, Document 243192, 1999.
- [ISO, 96]** ISO/IEC 9945-1, IEEE Std 1003.1, "Portable Operating Systems Interface (POSIX) – Part 1 : System Application Program Interface (API)", Information Technology Standard, July 1996.
- [ISO, 98]** ISO/IEC 10746-4, "Open Distributed Processing - Reference Model - Reference Model : Architectural Semantics", Information Technology Specification, 1998.
- [Levens, 00]** G.Levens and M.Sitaraman, "Foundations of Component Based Systems", Cambridge University Press, ISBN 0-521-77164-1, 2000.
- [Pietrek, 00]** M.Pietrek, "Avoiding DLL Hell: Introducing Application Metadata in the Microsoft .NET Framework", Microsoft Developer Network Magazine, October 2000.
- [Microsoft, 95]** Microsoft Corporation, "The Component Object Model Specification", Technical Specification, Version 0.9, <http://www.microsoft.com/com/resources/comdocs.asp>, October 1995.
- [Microsoft, 97]** Microsoft Corporation, "Object-oriented Software Development Made Simple with COM+ Runtime Services", Microsoft Systems Journal, November 1997.
- [Morris, 99]** R.Morris, E.Kohler, J.Jannotti and M.E.Kaashoek, "The Click Modular Router", in Proceedings of the 17th ACM Symposium on Operating Systems Principles, December 1999.
- [OMG, 98]** Object Management Group, "CORBAtelecoms: Telecommunications Domain Specifications", Document Number 96-06-02, Version 1.0, <http://www.omg.org/corba/ctfull.html>, June 1998.
- [Stewart, 96]** D.B.Stewart and G.Arora, "Dynamically Reconfigurable Embedded Software – Does It Make Sense?", in Proceedings of IEEE International Conference on Engineering of Complex Computer Systems (IECCS) and Real-time Applications Workshop (RTAW), Montreal, Canada, pp. 217-220, October 1996.
- [Stewart, 97]** D.B.Stewart and P.K.Khosla, "Design of Dynamically Reconfigurable Real-time Software Using Port-Based Objects", in IEEE Transactions on Software Engineering, Vol. 23, No. 12, December 1997.
- [Stewart, 00]** D.B.Stewart, "Designing Software Component for Real-Time Applications", in Proceedings of IEEE Embedded Systems Conference, San Jose, CA, September 2000.

[Szyperski, 99] C.Szyperski, “Component Software – Beyond Object Oriented Programming, Addison-Wesley Publishing, ISBN 0-201-17888-5, 1999.

[Waddington, 96] D.G.Waddington, G.Coulson and D.Hutchison, “Specifying QoS for Multimedia Communications within Distributed Programming Environments”, Third International COST237 Workshop, Barcelona, Spain Multimedia Telecommunications and Applications, Springer Series LNCS 1185, p.75-101, November 1996.

[Waddington, 01] D.G.Waddington and R.Viswanathan, “In-process Modular Programming : A Platform for Real-time and Embedded Component Development”, Internal Technical Report, Network Services Management Research Dept, Bell-labs, Lucent Technologies, April 2001.