DISTRIBUTED SYSTEM CONSTRUCTION: EXPERIENCE WITH THE CONIC TOOLKIT

Naranker Dulay, Jeff Kramer, Jeff Magee, Morris Sloman, Kevin Twidle

Department of Computing,
Imperial College,
180 Queensgate,
London SW7 2BZ.

Keywords:

Distributed systems, distributed programming, configuration, host/target environment, dynamic configuration.

Abstract:

For the last eight years the Distributed Systems Research Group at Imperial College has conducted research into the development of an environment to support the construction and operation of distributed software. The result has been the Conic Toolkit: a comprehensive set of language and run-time tools for program compilation, building, debugging and execution in a distributed environment. Programs may be run on a set of interconnected host computers running the Unix™ operating system and/or on target machines with no resident operating system.

Two languages are provided, one for **programming** individual task modules (processes) and one for the **configuration** of programs from simpler groups of task modules. In addition the enviroment supports the re-use of program components and allows the configuration of new components into running systems. This **dynamic configuration** capability is provided by a distributed configuration management tool which is the primary method of creating, controlling and modifying distributed application programs. The system also supports user transparent datatype transformation between heterogeneous processors.

This paper describes and reflects on the major design principles of the Conic toolkit and discusses the experiences both of the Conic research group and the various other universities and industries who are using the toolkit.

1. INTRODUCTION

The Conic Toolkit provides a language-based approach to the building of distributed applications. Flexible configuration, modularity and reuse of software components is facilitated by separation of the language for **programming** individual task modules ("programming in the small") from the language for **configuring** programs from predefined modules ("programming in the large"). The configuration language provides a concise configuration description and hierarchical composition, and is used to specify the configuration of software modules (processes) in terms of instances of components and their logical interconnection.

Large distributed applications are subject to both **evolutionary** and **operational** changes. Evolutionary changes occur through the need to incorporate new functionality and technology in a manner which is difficult to predict. Operational changes result from the need to redimension to cater for growth and to reorganise to recover from failures. It is impractical and uneconomic to take out of service an entire distributed system simply to modify part of it. Conic caters for these requirements by language and run-time support for **dynamic configuration** [Kramer 85] of logical nodes. This permits on-line modifications to a running Conic system using the configuration language.

™ Unix is a trademark of AT&T Bell Laboratories.

Various versions of the Conic toolkit have been in use for about 8 years at Imperial College, by research groups at other universities and in industry. We have used the environment as the basis for further research, for substantial student research projects and for student exercises on concurrency and communication protocols. The industrial users include British Coal for the implementation of underground monitoring and communication in coal mines; British Petroleum for research into reconfigurable control systems and GEC for the development of an object-oriented support system and front-end security processor. Conic has also been used for a number of years for research on self-tuning adaptive controllers [Gawthrop 84]. It is also being used for research and teaching at universities in Canada, France, Japan, Korea and Sweden.

This paper will reflect on the the major design principles and on the evolution of Conic as a result of user experiences.

2. DESIGN PRINCIPLES

i) Language Approach

Providing support for distribution at the language level permits modularity, concurrency, synchronisation and communication facilities to be integrated into a single framework [Strom 85, Hoare 78, Andrews 86, Black 87, Scott 87]. Compile, link and run-time checks can ensure message compatibility between components. Consistent naming, communication and synchronisation can be provided for both local and remote interactions. Thus language environment are generally **simpler** and **safer** to use.

ii) Separation of configuration and programming

A configuration specification together with a change history give a specification of the current system configuration. If configuration operations are embedded in the programming language, the current configuration of the system can only be determined from the state of the individual components which is difficult to determine in a distributed environment. In addition, unpredicted changes, and installation of new component types into a system are more difficult with the single language approach.

iii) Modular Approach

Modularity is key property for the provision of flexibility and reuseability of components [Wegner 84]. Modules are treated types as there is often the need for multiple instances of a module type within a system. In addition, providing the same support for modularity, in terms of external interfaces, at both the programming and configuration levels permits a configuration to consist of either individual program modules (tasks) or nested configuration (group) modules. This provides a powerful abstraction mechanism.

iv) Host Target Environment

Conic was originally designed for support of embedded systems where the simple target computers used for real-time applications lack the facilities for program development. Host computers are used to develop software for subsequent downline-loading into the targets. In practice Conic has been used to construct a wide range of applications, from general distributed algorithms to system support utilities and services, on both targets and hosts (see fig. 2.1). The capability of running in a **mixed** host target environment permits targets to be used for device interaction and real-time response, while the hosts provide access to the file servers, graphics displays and printing services.

v) Simplicity

It is better to provide simple, extensible constructs which permit more complex facilities to be implemented 'on-top', rather than forcing users to to pay the cost of powerful primitives even when they are not required.

vi) Flexibility

The Conic system and its environment is "**open**" [Redel 80] in that it provides easy access to all its facilities by use of a common message passing interface structure for all component interaction. Both

distributed applications and the Conic support system itself are constructed using the same tools and techniques. With the exception of less than 100 lines of assembly code in the kernel, *all* the software for the Conic environment is implemented in Conic. This **uniformity** permits users to tailor or extend the system facilities to suit their particular requirements, although this is not normally performed by naive applications programmers. The ability to easily modify the system is an essential attribute for an experimental environment. It also facilitates configuration of the support system itself to suit particular hardware or application environments.

Fig. 2.1 depicts a typical Conic environment. A **logical node** is the system configuration unit. It is a set of tasks which execute concurrently within a shared address space on a host as a Unix™ process or directly on a target. Systems are constructed as sets of one or more interconnected logical nodes. Communication between tasks within a logical node and between logical nodes is supported **uniformly** by message passing. This provides a simple communication facility between local and remote tasks which hides the complexity of the network interface. On a target computer, Conic executes with no resident operating system other than the Conic executive, but can still access the services and facilities of the general purpose host operating system.

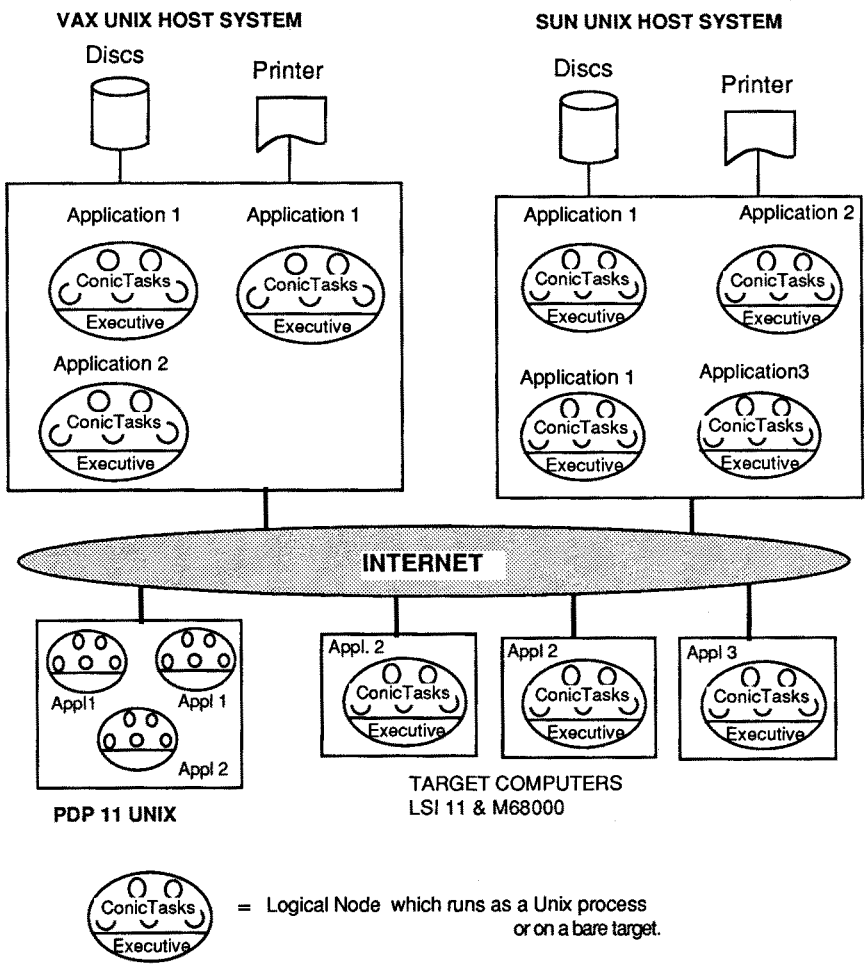


Fig. 2.1. Distributed Applications in a Conic Environment

3. CONIC MODULE PROGRAMMING LANGUAGE

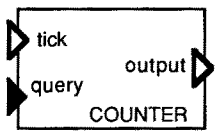
3.1 Task Modules

Modularity is a key property for providing flexibility. The Conic programming language is based on Pascal, with extensions for modularity and message passing [Kramer 84].

The language allows the definition of a **task module type** which is a self-contained, sequential task (process). A task module type is written and compiled independently from the particular configuration in which it will run and so provides **configuration independence** in that all references are to local objects and there is no direct naming of other modules or communication entities. This means there is no configuration information embedded in the programming language and so no recompilation is needed for configuration changes as is the case with other languages such as CSP [Hoare 78] and Ada [DOD 80].

At configuration time, **module instances** are created from module types. Module instances exchange messages and perform a particular function in the system such as performing a computation, managing a resource or controlling a device. Multiple instances of a module type can be created on the same or different stations in a distributed system and a station can contain many different modules instances.

Fig. 3.1 is an example of a task module which counts ticks from a clock and sends a message when the count has reached a limit. The value of the count can be queried.




```
task module counter (limit:integer);
 exitport
     output : integer;
 entryport
     tick : signaltype;
     query : signaltype reply integer;
 var count  : integer;
 begin
     count := 0;
     loop
         select
             receive signal from tick
                 => count := count + 1;
                     if count = limit then
                       begin
                         send count to output;
                         count := 0;
                       end;
         or
             receive signal from query  reply count
         end;
     end;
end.
```

Fig. 3.1 Example Task Module

CONIC modules have a well defined, strongly typed interface which specifies all the information required to use the module in a system. The interconnections and information exchanged by modules is specified in terms of **ports**. An **exitport** denotes the interface at which message transactions can be initiated and specifies a local name and message type in place of the destination name. In fig. 3.1, the count is sent to the task's *output* exitport when its value reaches a limit given as an instantiation

parameter. At configuration time, the exitport can be linked to any compatible entryport (ie. of type integer) of a task which wishes to receive the message. The **entryports** *tick* and *query* in fig. 3.1, denote the interface at which message transactions can be received. At configuration time, any task with a compatible exitport can be linked to these entryports. The programming language uses local names within the task instead of directly naming the source and destination of messages. The binding of an exitport to an entryport is part of the configuration language and cannot be performed within the programming language. Therefore there is no need to recompile a task module when it is reused in different situations. This provides complete configuration independence for a task module.

At instantiation time, parameters can be passed to a module to tailor a module type for a particular environment, for example the limit value passed to the *counter* task in fig. 3.1, or the device address passed to a device driver (see section 3.4).

There are two classes of ports which correspond to the message transaction classes described below. **Request-reply Ports**, such as *query* in fig. 3.1 are bidirectional. They specify both a request and reply message type. **Notify Ports** such as *tick* and *output* are unidirectional ie. they have no reply part. For convenience, it is possible to define families (arrays) of identical ports as described in section 4.

Ports define all the information required to use a module and so it is very simple to replace a module with a new or different version with the same operational interface.

3.2 Communication Primitives

Communication primitives are provided to **send** a message to an exitport or **receive** one from an entryport. The message types must correspond to the port types. The primitives provide the same syntax and semantics for local (intra station) and remote (inter-station) communication. Differences in performance between local and remote communication are inevitable due to network delays. This **Communication Transparency** allows modules to be allocated either to the same or different stations, which can be particularly useful during the development of embedded systems in that modules can be fully tested together in a large computer with support facilities and then later distributed into target stations.

There are two classes of message transactions:

a) A **Notify transaction** provides unidirectional, potentially multi-destination message passing. The send operation is asynchronous and does not block the sender, although the receiver may block waiting for a message. There is a (dimensionable) fixed size queue of messages associated with each entryport. Messages are held in order of arrival at the entryport. When no more buffers are available the oldest message in the queue is overwritten. The Notify transaction can be used for time critical tasks such as within the communication system, with the queue size corresponding to a flow-control window or for periodic status information, when the latest information is of interest and the entryport specifies a single buffer.

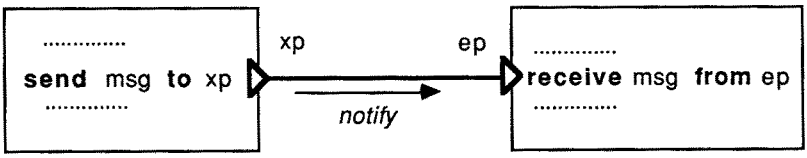


Fig. 3.2 The Notify Transaction

b) A **Request Reply** transaction provides bidirectional synchronous message passing. The sender is blocked until the reply is received from the receiver. A fail clause allows the sender to withdraw from the transaction on expiry of a timeout ('tval' in fig. 3.3) or if the transaction fails. The receiver may block waiting for a request. On receipt of a request, the receiver may perform some processing and return a reply message. In place of a normal reply, the receiver may either **forward** the request to another receiver (thereby allowing third party replies) or it may **abort** the transaction.

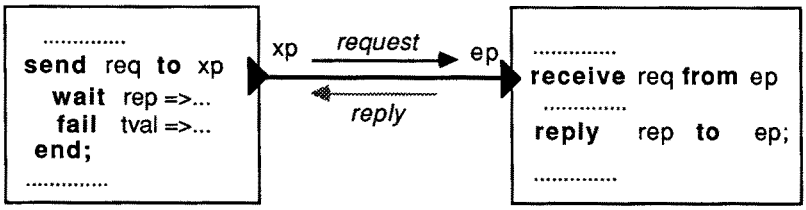


Fig. 3.3 Request-Reply Transaction

Standard functions are provided to determine whether an exitport is linked to an entryport, the number of messages queued at an entryport or the reason for a send-wait failing.

Any of the receive, receive-reply, receive-forward, or receive-abort primitives can be combined in a **select** statement (fig. 3.4). This enables a task to wait on messages from any number of potential entryports. An optional guard can precede each receive to further define conditions upon which messages should be received. A timeout can be used to limit the time spent waiting in the select statement. The order of selection is defined by the textual ordering of the alternatives in the select statement i.e. if there is a message waiting on both *ep1* and *ep2* in fig 3.4, then the the message on *ep1* will be received first.

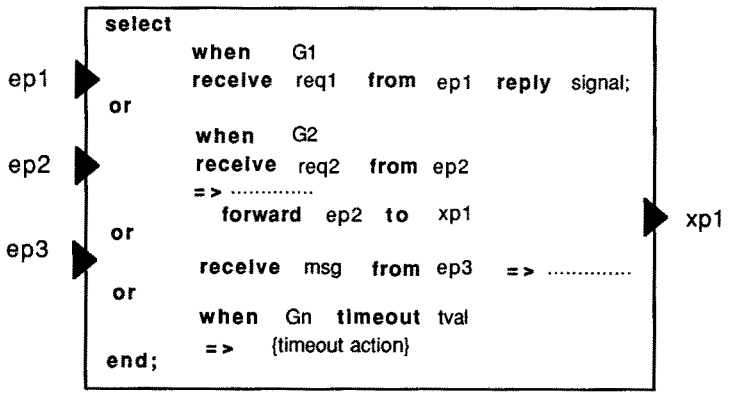


Fig. 3.4 Selective Receive

3.3 Definitions Unit

The module is the basic reuseable software component within a system. However there are many definitions which are common between different modules within a system. Definitions of constants, types, functions and procedures may be defined in separate definitions units. These can be compiled independently and can be imported into a module to define a **context**. This avoids errors introduced by having to redefine message types in communicating modules. For example the definition of message type *valtype* would be imported from a definitions unit called *msgtypes* by means of a declaration such as:

```
use msgtypes : valtype;
```

The definitions unit allows the introduction of language "extensions" without modifying the compiler. For example a set of standard string definitions and manipulation procedures can be made available as a definitions unit as shown in fig. 3.5. This exports 2 functions *strlength* and *strcpy*, and a type *string*.

```
define stringdefs: strlength, strcopy, string;

   const strmax = 128;
   type string = record
                    len:integer;
                    ch :array[1..strmax] of char;
   end;
   function strlength (s:string):integer;
      ......
   procedure strcopy (s1,s2:string);
      ......
end.
```

Fig. 3.5 An Outline Definitions Unit

A definitions unit can encapsulate data, initialisation and access procedures for the data. This is similar to an Abstract Data Type but only a single instance can be declared when it is imported into a task module. However multiple instances of the encapuslating task module can be declared. The encapsulating task can access the data via exported procedures or directly (if the data variables are exported) but other modules must access the data via the encapsulating task's message passing interface.

3.4 Input-output

The programming language supports the standard Pascal and C I/O procedures, whichcan be freely mixed. These are automatically transformed by the compiler into message passing operations on standard, pre-declared task exitports.

In addition, CONIC provides simple primitives to support the programming of device handlers as application tasks. We have experimented with 3 versions of interrupt handling. Initially we used the Modula 1 type of kernel call *waitio (interrupt vector)* [Wirth 77]. This was called by the device handler task whenever it wished to wait for an interrupt. It was not possible to wait for both an interrupt and a message and the *waitio* resulted in a task context switch to the device handler for every interrupt, which slowed down the response time. We then tried the Ada mechanism of the Kernel converting an interrupt to a message [DoD 80]. However this was very slow as it resulted in a context switch and a message transfer for each interrupt.

With the current mechanism, a device driver task defines a procedure for each interrupt it handles. Fig. 3.6 shows a transmitter driver for a serial port, based on LSI 11 hardware. It makes use of a set of special kernel calls imported from a definitions unit called *kercalls*. The task raises its priority to *system* to ensure that it is not preempted by any other task while transmitting a message. Different device drivers may have different hardware priority levels, allowing nested interrupts. The *intmap* procedure maps a handler procedure to the interrupt generated on the given vector. It also specifies an entryport from which the driver task will receive a signal from the handler procedure. The interrupt procedure runs in the context of the interrupted process, so it cannot use the normal message primitives, but it can make a special kernel call to send a signal when it has completed its function.

The above mechanism is very efficient, yet it means interrupt handlers are not part of the kernel, but are syntactically part of the device driver task. Consequently device drivers can be written and incorporated into a system without modyfing the kernel. This simplifies the writing and configuration of device drivers.

```
task module transmit (status,vector : natural);

 use
       commstypes : msgtype;
       kercalls : priority, {system, normal etc.}
           setpriority, {to set task priority}
           SendSignal, {special message from interrupt handler}
           intmap; {maps handler procedure to interrupt vector}
 entryport
    tx : msgtype reply signaltype;
    done : signaltype;
 const
    enable = 0100#8;
    disable = 0;
 var
    txstat : ^natural;
    txbuff : ^char;
    msg : msgtype;
    count : integer;

 procedure inthandler;
    begin
        if count <= msg.len
        then begin
            xbuff^ := msg.chars[count];
            count := count + 1;
        end
        else begin
            txstat^ := disable; {completed}
            SendSignal; {to done entryport}
        end;
 end;

begin
   ref (txstat,status);  {converts address to pointer type}
   ref (txbuff,status+2);
   setpriority (systempr); {raises task priority}
   intmap (done, vector, inthandler);
   {Kernel sets inthandler to run when interrupt received on vector}
   {done entryport is set up to receive signal from handler}
   loop
    receive msg from tx;  {wait for message to transmit}
        count := 1;
        txstat^ := enable; {generates immediate interrupt}
        receive signal from done;
    reply signal to tx;
   end ;
end.
```

Fig. 3.6 Device Driver Task Module

3.5 Discussion

The Request-reply and notify communication primitives have proved to be an excellent choice in that they do cater for most interaction requirements. In an early version of Conic we tried to do without an the asynchronous notify, but this led to deadlocks and a proliferation of tasks which gave immediate replies to make the synchronous send-wait appear asynchronous to the sender. Relying only on a synchronous request-reply primitive definitely complicated the programming of many applications. This is borne out by other systems which previously provided only remote procedure calls and are now introducing asynchronous remote procedure calls [Liskov 87]. There is a need for some form of overwrite strategy in the notify, as otherwise there must be some form of backward flow of information from the receiver to the

sender. This leads to less efficient implementation and a send which is not really asynchronous i.e. the sender is delayed or blocked if there are no buffers at the receiver. This can be easily programmed as a flow control protocol, if the overwrite semantics of the notify are inappropriate and blocking the sender on buffer exhaustion is required.

The select statement gives priority on textual ordering. This could result in starvation, but none of the users reported this as a problem. Usually different entryports are used for different types of requests rather than different clients, and requests on a particular port are queued in arrival order. Guarded recieves could be used to overcome starvation problems if required.

The port based indirect addressing for communication primitives has proved very useful. This is one of the most important contributions to Conic's configuration flexibility in that it enables components to be reused in many alternaive configurations.

Although remote procedure calls (RPC) are currently very fashionable, we have no regrets about the provision of message primitives. Our request-reply transactions is similar to an RPC without prameter marshalling. However it is more flexible as it has a clause for handling errors. RPC implementations often provide another sysnchronisation mechanism (e.g. monitors) as well, whereas the message primitives can be used for both communication and synchronisation.

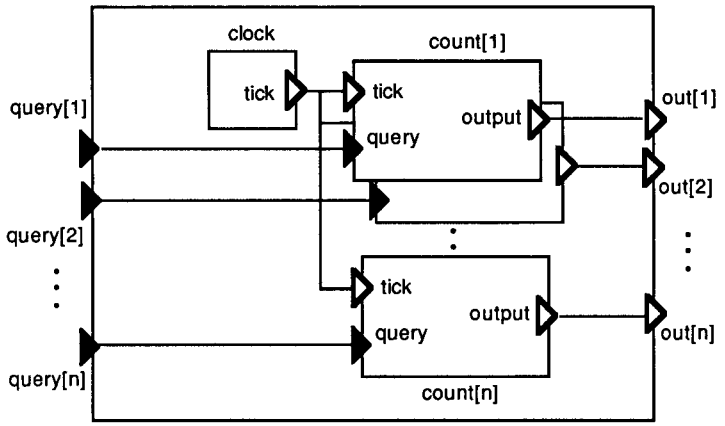
On the whole, our user experience has been that the minimal extensions we have provided to Pascal have made it sufficiently versatile to be used for programming a wide range of applications. However the Configuration language and dynamic configuration form the most interesting and novel aspects of the Conic toolkit.

4. CONIC CONFIGURATION LANGUAGE

One of the key elements in the provision of flexibility is the need to separate the programming of individual software components (task module types) from the building of a system from instances of modules. This has led to the development of the CONIC Configuration Language [Dulay 84] which is used to specify the instances of module types and their interconnection to form a logical node. An interactive variant of the language is also used to specify to the dynamic management system the configuration of logical nodes which constitute a distributed application.

The structure of tasks within a logical node is described as a hierarchy of group modules. For example, Fig. 4.1 describes a group module *counters* composed of the two task types *counter* (from Fig. 3.2) and *clock*. The **use** construct specifies the set of message types necessary to declare a module interface (in this case null since the messages are of base types) and the set of task and/or group module types. Instances of task (or group) types are specified by the *create* construct. In the example one instance of *clock* is created and a **family** (array) of instances of *counter*. The interface to *counters* is a family of both exitports and entryports.

The **link** construct specifies the interconnection of module instances by binding a module exitport to a module entryport e.g. linking the *tick* exitport on *clock* to the *tick* entryport on each instance of *counter* in fig. 4.1. Both type and operation compatibility are checked so an exitport can only be linked to an entryport of the same data and transaction type. Multiple exitports can be linked to a single entryport which is particularly useful for connecting clients to servers (eg. a file server). A single exitport can be linked to multiple entryports which provides multidestination message transactions. Multidestination is generally used for notify transactions. In addition **link** binds group module interface ports to ports on internal module instances. This linking is merely a name mapping and does not entail any run-time overheads, i.e. there is no copying or queuing of messages at interface ports.




```
group module counters (limit : integer = 1000, n : integer = 2);
    exitport
        out [1..n] : boolean;
    entryport
        query [1..n] : signaltype reply integer;
    use
        clock; counter;
    create
        clock;
    create family k : [1..n]
        count [k] : counter (n*limit);
    link family k : [1..n]
        clock.tick          to count [k].tick;          {multi-destination}
        query [k]           to count [k]. query;
        count [k].output    to out [k];
end.
```

Fig 4.1 Counters Group Module

The Conic Configuration language supports default parameter values. For *counters*, the default number of instances *n* is 2 and the default *limit* is 1000. The default value of *n* and *limit* can be overidden by passing a value when creating an instance of a group module.

It should be noted that the interface to a group module is identical to that of a task module. When a group module type has been defined, it may be instantiated and connected in exactly the same way as a task. Hence complex configurations can be built up by nesting groups and tasks within groups to any required level. We have found the group module abstraction to be a powerful way of structuring the tasks which constitute a logical node: the unit of distribution.

Each group module specification is separately compiled into a symbol table and a set of functions which will instantiate its structure at node instantiation time. A group module type which includes an instance of the run-time executive (itself a group module - see section 6) can be compiled into an executable load file from which logical nodes are created. The hierarchical structure of configuration specifications has no run-time overhead as it is flattened into a uniform address space of task instances at the time a node is instantiated.

Conic provides no explicit support for sharing data between task modules. However, within a logical node messages can contain pointer values. Consequently, a task can give direct access to the data it encapsulates. Mutually exclusive access can be enforced using the message passing primitives for synchronisation. In the respect that tasks exist in the same address space within a logical node, Conic tasks are similar to the "lightweight" processes of the V-kernel [Cheriton 84] and Amoeba [Mullender 86].

5. DYNAMIC CONFIGURATION

5.1 Logical Nodes

Distributed programs in Conic are constructed from sets of pre-compiled logical node types with the aid of the dynamic configuration tools . A logical node may run either as a Unix process if it includes an instance of *unixexec* or on a standalone target if it includes an instance of *targexec*. These run-time executives support multi-tasking, message passing and dynamic configuration operations. They are described in section 6.

Like group modules, logical nodes are types in the sense that more than one node instance may be created from the code file which represents the node type. Actual parameters substituted at instantiation time can control the numbers of tasks created within nodes and the values passed to those tasks.

It should be noted that the interface to a logical node is specified in exactly the same way as the interfaces of group and task modules. The distinction between a group module implementing a logical node and any other group module is that the logical node includes a run-time support executive.

5.2 Example

The following example, based on the Sieve of Eratosthenes to generate prime numbers, is used to illustrate the dynamic construction of a distributed pipeline in Conic. The task module types required are shown in Fig. 5.1. The generator generates a stream of odd numbers and the sieve task prints the first prime it receives, then removes multiples of this prime from the stream of odd numbers. Note that a sieve attempts to send the filtered stream to its right neighbour. This fails, and periodically retries if there is no neighbour i.e. the exitport is not linked.

```
task module generator;
exitport
    right:integer reply signaltype;
var value:integer;
    output:text;
begin
    writeln(2);             {First prime}
    flush(output);
    value:=3;
    loop
        send value to right wait signal;
        if value=maxint then exit;
        value:=value+2;
    end;
end.
```

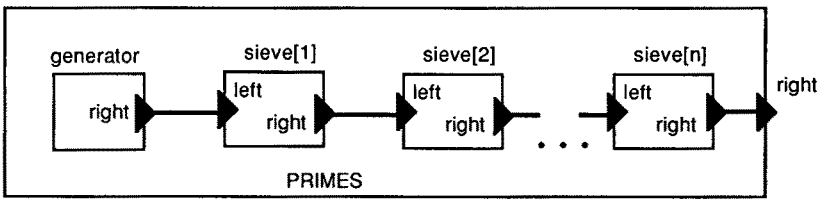
```
task module sieve;
entryport
    left: integer reply signaltype;
exitport
    right:integer reply signaltype;
var
    prime, multiple, value : integer;
    output:text;
begin
    receive prime from left reply signal;
    writeln(prime);
    flush(output);  {force output}
    multiple:=prime;
        loop
        receive value from left reply signal;
        while value>multiple do
            multiple:=multiple+prime;
        if value<>multiple then
            loop
                send value to right wait signal
                => exit;
                fail => delay(1000);
                end;
            end;
    end;
end.
```

Fig. 5.1 Task Modules for Sieve of Eratosthenes

Figure 5.2 shows a logical node which generates the first *n* primes. The include in the second line includes a header file to create an instance of the relevant type of executive. For example to run on Unix it would generate the statements:

```
use     unixexec;
create  unixexec;
```

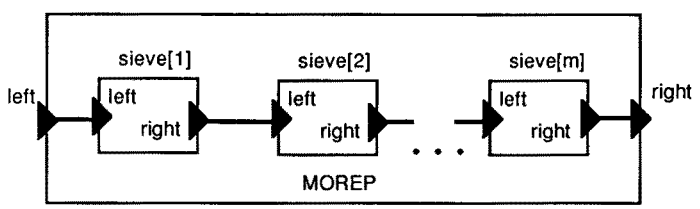
The executive is a group module which provides run-time support for multi-tasking, message passing and dynamic configuration facilities. It is described in more detail in section 6.




```
group module primes(n:integer);
    #include <node.h>                    {create executive}
    exitport
        right:integer reply signaltype;
    use
        sieve;
        generator;
    create
        generator;
    create family k:[1..n]
        sieve[k];
    link family k:[1..n-1]
        sieve[k].right to sieve[k+1].left;
    link
        generator.right to sieve[1].left;
        sieve[n].right to right;
end.
```

Fig. 5.2 Logical Node to Generate First n Primes

Fig. 5.3 describes a slightly different node type which will generate an additional *m* primes when connected to the first node. Assume it is meant to run on a target.




```
group module morep(m:integer);
    #include <node.h>
    entryport
        left:integer reply signaltype;
    exitport
        right:integer reply signaltype;
    use
        sieve;
    create family k:[1..m]
        sieve[k];
    link family k:[1..m-1]
        sieve[k].right to sieve[k+1].left;
    link
        left to sieve[1].left;
        sieve[m].right to right;
end.
```

Fig. 5.3 Logical Node to Generate *m* Additional Primes

The host compilation system produces an executable code file for each logical node type. To simplify the compilation and subsequent maintenance of complex logical node types, the Conic host system includes a *makefile generator* tool. This analyses group module specifications to determine dependencies and generates the required input file for the Unix *make* facility to build a logical node type from its constituent group module, task module and definition unit sources.

In the next section we show how the above logical nodes can be configured as a distributed application and subsequently dynamically extended.

5.3 Managing an Application Configuration

Conic distributed application programs are constructed from a set of pre-compiled logical node types. Each logical node type is contained in an executable code file. In the following, we will describe how the application can be mapped onto the hardware configuration of a Sun workstation and 2 targets depicted in fig. 5.4.

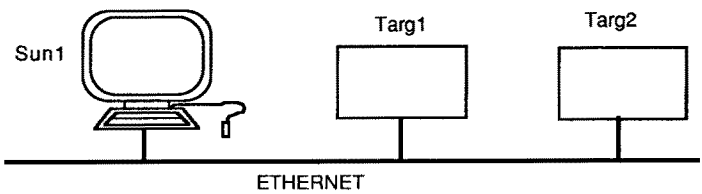


Fig. 5.4 Hardware Configuration

The logical configuration shown diagrammatically in fig. 5.5 is constructed by submitting the following set of configuration commands to a configuration manager. The commands may be typed interactively to an invocation of the manager (**iman**) or may be read from a file. The manager may be run in a window on one of the Suns or on a separate machine.

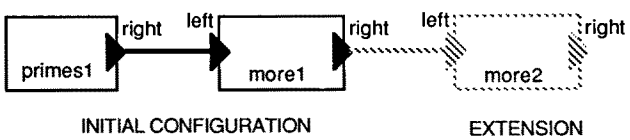


Fig. 5.5 Logical Configuration of Primegen

Configuration commands:-

```
manage primegen

create primes1: primes(50) at sun1
create more1: morep(50) at targ1

link primes1.right to more1.left

start primes1 more1
```

The **manage** command provides a name for the distributed application. A user may thus control one or more distributed applications concurrently. Each time the configuration manager is invoked, the user must specify the application he wishes to control. If omitted this name defaults to the user's Unix login name.

The **create** command creates the specified logical node type **at** a location. In this example *primes1* is created at *sun1* and *more1* at *targ1*. The **link** command is used to connect interface ports between logical nodes.

The language used to communicate with a configuration manager corresponds with the configuration language used to construct group modules. As yet the configuration manager does not implement the family construct supported by the group module compiler. This is mitigated to some extent by the fact that configuration commands can be executed directly by Unix shell as commands. The commands invoke the manager with their names as parameters in the standard Unix fashion.

Having constructed the initial application structure, it is now possible to modify it. An additional instance of *morep* is created at targ2.

```
create more2:morep(150) at targ2
link more1.right to more2.left
start more2
```

Additional control commands are also available to **stop** logical nodes, **remove** instances and **unlink** ports. As well as providing commands to control a configuration, the manager provides a set of queries to let the user examine the state of his system:

systems	lists the set of applications currently running.
nodes	lists the set of nodes within a system together with their current state (*started*, *stopped*).
ports <node>	lists a node's interface ports and types
links <node>	lists the entryports to which a node's exitports are connected.

5.4 Summary and Discussion

This section has attempted to give a user's view of the Conic system. The functionality of an application is implemented by task modules and definition units using the Conic Programming Language. These tasks may be combined into groups to provide extra levels of structuring using the Conic Configuration Language. The set of task and group types is then partitioned into logical node types. These logical node types form the unit of distribution. When defining a logical node type the user must consider the environment in which the node is to execute (host or target) and include the appropriate run-time support executive. Compiling a logical node type results in an executable code file. This compiled node type, although it is constrained as to whether it may run on a host or target, is unrestricted as to its hardware location and the particular logical configuration in which it will run. Furthermore, the number of task instances contained within a logical node can be specified by parameters at node creation time.

The initial construction and subsequent modification of an application is carried out using a configuration manager which allows the user to create instances of logical nodes at specified locations within his network. These instances are interconnected to form the logical application configuration.

Essentially, the Conic system has two constraints in the dynamic configuration flexibility that it offers. Firstly, the set of task and group types from which a node type is constructed is fixed at node compile time. The principal reason for this is the simplification to the dynamic configuration system which results from management at the node level. The internal structure of a node is essentially invisible to the configuration management system. A secondary reason is that it is nearly impossible under Unix to implement loading and linking of new code into a running process in such a way that is portable across the different versions of Berkeley Unix and the different machine architectures supported by these versions.

The second constraint is that the number of task and group instances within a node is fixed at the time a node is created. Although the set of task types is fixed, additional instances of these types *could* be created inside a node in response to application or configuration system actions. This second constraint is largely as a result of the historical development of the Conic system and is less easy to justify. One of the original objectives of the Conic system was to provide a strict separation between programming-in-the-small (provided by tasks and definition units defined using the Conic Programming Language) and programming-in-the-large (provided by group modules defined using the Conic Configuration Language). It was felt that providing primitives for task creation and inter-connection within the programming language would lose this strict separation. Currently, the Conic group is investigating ways of providing dynamic tasking within a node, without completely losing the separation. The distinction between programming and configuration is felt worth preserving since it results in system structures which are easy to understand and in modules which can be used in many different applications.

The objections to static tasking outlined in [Liskov 85] are largely overcome in CONIC through the use of the **forward** statement. This allows a server task to forward messages, the servicing of which may incur local or remote delays, to one of a pool of "worker" tasks. The **forward** transfers the request message to a worker allowing the server to continue immediately and enabling the worker to reply directly to the original sender of the request. However, the size of the pool of worker tasks is fixed at node instantiation time.

Our initial conception of dynamic configuration management [Kramer 85] involved what was essentially an on-line database which recorded the current configuration state. It was intended that a dynamic configuration manager would use this database to retrieve information on the current application configuration in order to perform changes. The dynamic manager would both change the system and update the configuration database. The database was intended to "mirror" the system providing translations from symbolic names to actual addresses. The database would ensure that only consistent and validated changes could be performed. One motivation for this design was that translation information need not be stored in target nodes which have no backing store and may have limited main store. This translation information would have been significant since we intended to manage systems at all levels down to the level of a task module.

The design outlined above had a number of significant problems, primarily concerned with the implementation of the database. To achieve a distributed and robust management system, it would have required a distributed database implementation with the attendant problems of maintaining replicated data and performing consistent atomic updates. While solutions exist to these problems and a distributed database could have been constructed we felt that this design was overly complex. The database would constrain the speed with which changes could be performed. This speed is particularly important when re-configuration is required as a result of failure. Consequently, we abandoned this design and the current implementation results from two fundamental decisions.

Firstly, it was decided that the user's requirement for dynamic configuration could be satisfied by management at the level of logical nodes. Essentially, the logical node became both the unit of configuration management and the smallest unit of failure. This decision dramatically reduces the quantity of information which must be handled by the management system. In the systems we have constructed to date, the configuration of tasks within a node is more complex than the configuration of nodes which combine to form an application. Nodes typically have 10 to 100 constituent task instances, including the executive.

Secondly, rather than have a separate configuration database, it was decided that a running application would be its own database. Each logical node would contain enough information to describe its own interface and its links to other nodes. The quantity of this information is small enough, as a result of the previous decision, to hold in main memory. A configuration manager obtains information on an application by querying a name server to find the set of logical nodes which constitute the application. Information concerning the node itself is obtained by communicating directly with the node.

6. RUN-TIME SUPPORT

Conic applications are intended to run in a mixed host-target environment. Logical nodes running on target machines must be able to communicate with nodes running under a host as a process. This constrains the Conic run-time system to use a communications protocol offered by the host operating system. Consequently, internode communication is implemented using the Internet UDP/IP datagram protocol [Leffler 83, Postel 83] offered by Berkeley Unix. However, to facilitate porting to different host operating systems, operating system dependencies are restricted to a small number of modules in the run-time system. Access to operating system functions by other parts of the run-time system is always carried out by sending messages to these modules.

The execution environment on which our development system runs at Imperial College consists of VAXs, Sun Workstations and some aging PDP11's running various versions of Berkeley UNIX and interconnected by Ethernet (see fig 2.1). Users may develop software on any of the machines and run it on some (or all) of these host computers. In addition, target 68000 and LSI11/73 computers (also connected to the Ethernet) are available for applications which require real-time response. Typically these targets are used for real-time control experiments. The Conic system supports cross-compilation from the Suns and VAXs to PDP11 targets. This environment means that the software for a particular application may be developed on a number of host machines, executed on both these and additional host and target machines,

and managed from a different machine. The Conic support environment must thus allow the distributed development of applications as well as their distributed execution and management in this heterogeneous hardware environment.

In the following section, both the structure of the run-time environment and the rationale behind its design are outlined.

6.1 Configuration Management

Node Interface

In addition to its application defined interface, each compiled logical node type has a set of ports which provide the management interface to instances of the node (Figure 6.1). This standard interface is implemented by the node's executive: *unixexec* for nodes which run as UNIX processes, and *targexec* for nodes destined for targets.

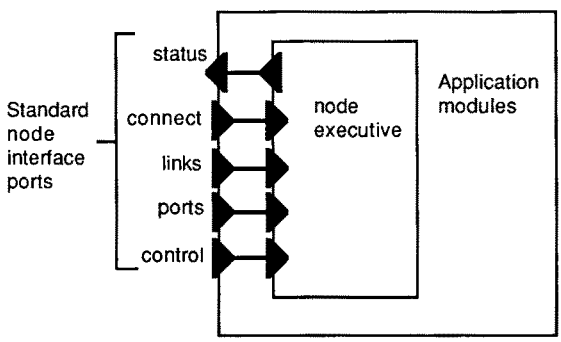


Figure 6.1 - Node Standard Interface Ports

The services provided by the node's management interface entryports are as shown in Fig. 6.1, and are as follows: *ports* returns a description of the node's interface in terms of the names and types of its ports; *links* returns the set of connections or links from the node's exitports to external entryports; *control* changes the configuration state of the node (started, stopped) in response to requests; *connect* links or unlinks node exitports to external entryports in response to requests. The exitport *status* is linked at node startup time to the name server as shown in Fig. 6.2.

Name Server

The name server has the only "well-known" or fixed UDP/IP address in the system. When a node is instantiated it obtains the address of the server from a Unix environment variable and links its exitport *status* to the *server* entryport *statusport*. The node registers itself with the server by sending a message containing its system name, node instance name, node type name, Internet address and its configuration state.

The server is a central point of failure in the configuration management system since it is the only place that configuration managers can find the addresses of logical nodes. To overcome this reliability problem, nodes send registration messages to the server at regular ten second intervals in addition to informing the server of a change of configuration state. If the server crashes and is subsequently restarted, it can recover its full database on the set of logical nodes within 10 to 20 seconds. Further, provision is made for replicating the server by allowing nodes to link to one or more instances of the server node on startup. Registration messages are then sent periodically to each server to which the node is linked. The robustness of the configuration management system is thus a function of the communication overhead that a user is willing to pay.

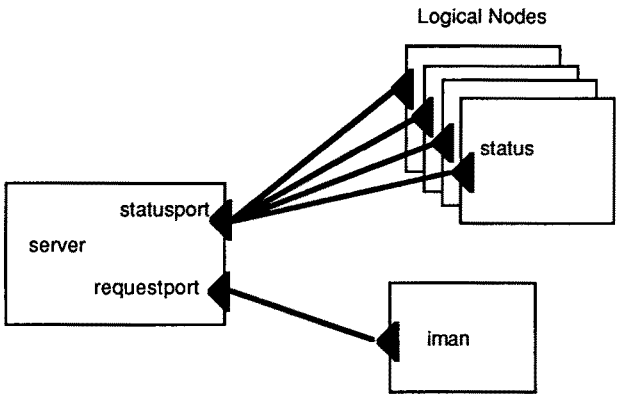


Figure 6.2 - Configuration Name Server

As with the rest of the management system, the name server is implemented entirely in Conic as a logical node type and may consequently run on a host or target computer depending on the node executive included.

Configuration Manager (*iman*)

The logical node type *iman* provides the user interface to configuration management. It may be invoked directly as a UNIX command to provide an interactive command interface or it may be invoked by command files as described in the previous section. When invoked, the manager *iman* links to the server as shown in Figure 6.2 and obtains the names and addresses of all the nodes running in a particular application system which, by default, is the user's UNIX login name. The system can be changed using the **manage** command as described in the previous section. The manager performs configuration actions on a node by linking its exitports to the management entryports of the node and invoking the management services provided by the node's executive. Since the Conic message passing primitives do not guarantee reliable delivery, the protocols used to invoke management actions on a node are designed to be idempotent.

There is no restriction on the number of instances of *iman* which may be active managing a particular system. Consequently, it is currently possible for a manager to perform incorrect operations based on an inconsistent view of the system it is managing. We are investigating the implementation of a robust locking mechanism which would survive server crashes and prevent managers from destructive interference when modifying the system. The problem is similar to file access locks required for multiple readers - one writer, but is simpler in that we do not actually require changes (writes) to be transparent.

Virtual Target (*vt*)

Logical nodes may be instantiated either by executing them directly as UNIX commands or by using the **create** statement supported by the *iman* interface. The command format for the first method is:

<node type name> [<parameters>] - [<node instance name> [<system name>]]

For example, the name server is created with the command:

server - conicserver conic

which creates an instance of the node **server** named **conicserver** in the system **conic.** As mentioned before, the system name defaults to the user's login name and in addition, the instance name defaults to the UNIX process number. This method is appropriate for creation on the user's local host; however, it does not support creation at either remote hosts or targets.

Remote creation on hosts is performed by a manager with the agency of a **virtual target** node running at the remote site. The virtual target is in effect a UNIX "shell" with a message passing interface. For example, a user wishing to create a logical node at a VAX from a manager running on the Sun Workstation (fig. 6.3) would type the commands:

manage primegen
create more2 : morep(75) **at** vax1

The manager locates the virtual target node *vax1* by communicating with the name server, links to it, and sends a message containing the string:

"morep 75 - more2 primegen".

The virtual target *vax1* then executes this command in the usual UNIX way (fork & exec). The nodetype must be compiled for the VAX and reside on it.

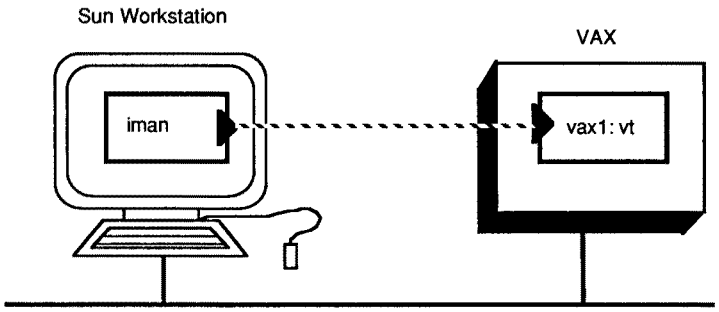


Figure 6.3 - Remote Creation

The advantage of implementing remote creation using this technique is the ease with which Conic applications can use host operating system resources. For example, suppose we wish a virtual target to create a Sun Window for each node of type *demo* it instantiates. In this case, the virtual target is created on Sun2 with the command:

vt shelltool - sun2 demo

The virtual target is designed to prefix commands from managers with its own arguments. Consequently, from the previous example, *sun1* will execute the command:

shelltool screen - newscreen snakedemo

Shelltool is the Sun workstation command which creates new windows.

In the same way, virtual targets running on a host support creation at real targets by invoking a download command (e.g. vt download target1- target1, provides access to the real target named target1). Currently, the code for a logical node type is assumed to be locally accessible to the virtual target. However, virtual targets can be given a UNIX shell macro as an argument. This macro would copy the code from a remote location using **rcp** and then execute it.

6.2 Node Executive

The structure of the runtime executive included in each logical node is the same for target executives as for host executives. This generic structure of a node executive is depicted in Figure 6.4. However, the implementation of some modules differs depending on whether they are used in the host executive *unixexec* or the target executive *targexec*. The functionality of each module and the differences between their host and target implementations are outlined below.

The *kernel* supports multi-tasking and inter-task communication within a node. It is implemented in Conic as a task module and is treated as such for configuration purposes. However, unlike normal task modules, it is not scheduled but executes in response to kernel calls from other task modules. A small amount of assembly code is required to provide task context switching. The host kernel provides facilities to handle UNIX signals whereas the target kernel supports real interrupt handling. Apart from this difference and a difference in the details of kernel entry, the host and target kernels are the same.

Messages destined for remote nodes are passed by the kernel to the *Communication Manager*. Under UNIX this module merely frames the message with a Conic inter-task communication header and passes it to the UNIX networking software via socket system calls. The target communications manager implements the full UDP/IP Internet protocol to frame messages and the Address Resolution Protocol

(ARP) [Plummer 82] to translate Internet addresses to Ethernet addresses. The particular Ethernet driver included in the target communication manager depends on the details of target hardware. A more detailed description of Conic communications may be found in [Sloman 86].

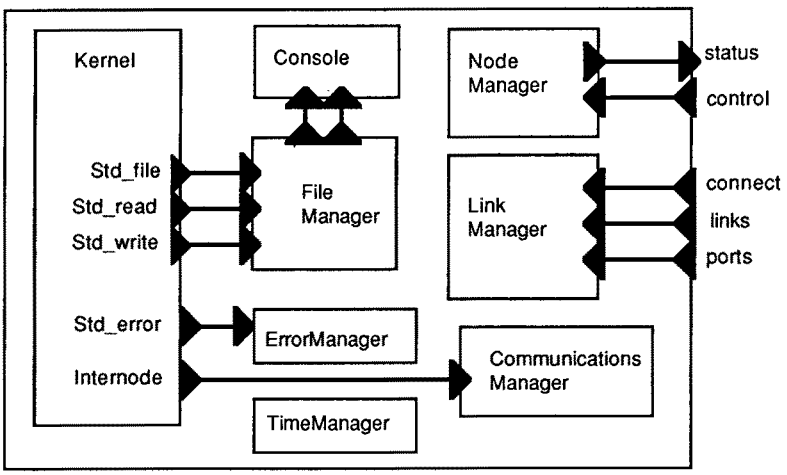


Figure 6.4 - Generic Node executive

The *File Manager* handles user task requests for both file and console I/O. Under UNIX, this manager either performs the appropriate system call or passes the request to the *console* module. The *console* module is necessary under UNIX to make the synchronous I/O calls appear asynchronous for other tasks running within the UNIX process (otherwise a *read* call from one task would suspend all tasks waiting for the read to complete). On a target, the file manager either forwards file requests to a node running on the host or passes them to the console module, which in this case is a real device driver.

The *Error Manager* is the same module on both host and target. It is usually configured to display error messages on the local console, but it may optionally produce a file containing the state of a task's variables at the time the error occurred. A tool is available to display the contents of this file symbolically.

Again, the *Link* and *Node Manager* modules are the same for both host and target. They implement the management interface described in section 6.1. Finally, the *Time Manager* module handles the targets real time clock interrupt or the Unix ALARM signal to provide real-time within the node.

Both *unixexec* and*targexec* are Conic group modules which represent a commonly used executive configuration. However, users are at liberty to configure their own version of the executive. They may do this using the standard modules or their own implementations of these functions. The executive is tailored to different target hardware configurations by including different versions of the device driver modules.

The table of Figure 6.5 gives an idea of the performance of inter-task communication on the range of host computers currently supported by Conic. The times in milliseconds are for a request-reply cycle transferring a 20 byte request message from sender to receiver and a 1 byte reply message.

	Intra-node	Inter-node (intra-host)	Inter-node (inter-host)
Sun 3/160	0.6ms	8.8ms	10.9ms
VAX 11/750	1.5ms	45ms	66ms
PDP 11/44	0.73ms	49ms	53ms
Sun - PDP	•••	•••	37ms
Sun - VAX	•••	•••	49ms
PDP - VAX	•••	•••	55ms
MVME133/1 (16.67 MHz 68020 target)	0.57ms	•••	5.2ms
Sun3 - 133	•••	•••	7.5ms

Fig. 6.5 - Inter-task Communication Performance

The figures were obtained when both the machines and the inerconnecting Ethernet were lightly loaded.

6.3 Support for heterogeneous machines

As previously mentioned, logical node types can be compiled and run on computers based on the 68000, VAX or PDP11 architectures. This is possible since both the group and task module compilers are based on the Amsterdam Compiler Kit (ACK) [Tanenbaum 83]. ACK makes use of an intermediate code (EM) to allow compilers to generate code for more than one target architecture.

To allow logical nodes running on different processor types to communicate, messages between nodes must be transformed to conform to the way data is represented on the destination machine. There are fundamentally two techniques for doing this. Firstly, messages can be transformed to a common data representation before being sent to the network. The destination machine then transforms the message to its local data representation. This technique is followed by the Sun RPC facility which uses XDR [Sun 85] as the common data representation. The disadvantage of this technique is that it requires two message transformations even when the machines communicating are of the same type. The advantage is that in an open network environment, each machine need only know how to transform between the common representation and its local representation. The addition of new machine types is thereby facilitated.

The second technique involves transformation only at the destination machine if required. A machine sends the message as a byte string in its local data representation together with a descriptor which identifies the source machine type and describes how the message is constructed from base types. The destination machine uses this descriptor to transform (if necessary) the message to its local data representation. The advantage of this technique is that it enhances communication performance by avoiding unnecessary data transformations. The disadvantage is that a machine must know how to transform all source representations into its local representation.

We have chosen the second technique in Conic for the following reasons. Most importantly, we wish to avoid any performance overhead in communication between homogeneous machines. Additionally, the technique fits well into the Conic environment since communication is always between typed exit and entry ports. Consequently, the message descriptor can be associated with the ports avoiding the overhead (although small) of transmitting it. Existing node types can easily be re-compiled to accommodate the (usually simple) additional transformation algorithm. Finally, the number of machine types supported by the Conic system is small.

Consequently, when the group module compiler produces a logical node type it associates type descriptors with each node interface port. These descriptors describe how the message type is constructed from the base types of the Conic language. An example of a descriptor is given below:

```
TYPE message = RECORD
                    str:PACKED ARRAY[1..100] OF char;
                    i,j,k:integer;
                    long:longint;
                    reading:real;
                END;

descriptor ::   100Ciiilr     {C=packed character, i= integer,l= long integer
                                and r= real}
```

The only additional information sent in a message is a tag identifying the source machine type.

Entry and exitports as described in section 3 may have both a request and a reply message type. For data transformation purposes it is only necessary to record the type descriptor for the entryports request type and the exitports reply type since transformation is always done at the destination. However, we record the request and reply descriptors at both entry and exit port ends of a link. The reason is to allow the configuration manager to perform type checking before setting up a link. The type descriptor is part of the interface description returned by the node's executive. Consequently, before a link is set up the manager checks that the exitport's type names and descriptors match exactly the entryport's type names and descriptors.

This is a weaker form of type checking than that performed by the group module compiler which checks that linked ports are using exactly the same version of a compiled type. This weakened form of type checking at the node level permits the independent (rather than separate) compilation of nodes which can later be configured safely into the same distributed application system. It avoids the problems of having to distribute symbol tables representing compiled types between machines of different types. The requirement for users on all machines to have access to the same versions of compiled types would make distributed development of systems difficult in our distributed environment.

6.4 Discussion

This section has described how the dynamic configuration facilities used in the previous section are provided. A management system may be easily tailored to a user's environment by the appropriate creation of instances of the three node types - *server, iman* and *vt* which together implement dynamic configuration management. When available, existing operating system resources and facilities can be simply accessed by virtual targets. New target hardware configurations can be accomodated by creating new versions of the target executive from existing modules and new device driver modules. In summary, the construction of the dynamic configuration support environment using Conic has the advantage of providing itself with the flexibility it provides for applications. Configuration actions are all supported by requesting actions on entryports. Consequently, applications may themselves request configuration changes when desired, for instance to recover from failures.

While giving much more flexibility than the original database approach to providing configuration management, this implementation can result in erroneous configuration actions as a result of more than one manager performing reconfiguration operations on the system at the same time (as described in section 61.) Our current research is investigating the provision of configuration transactions which would ensure consistent changes to the configuration.

Observant readers will have noted that a virtual target gives anyone access, through a configuration manager, to the files and programs it can access. This lack of security is inherent in the Berkeley networking software since anyone who knows the address of a socket may send a message to it. Unlike Amoeba [Mullender 86] which encrypts port addresses, socket addresses are not protected in any way and may be easily forged. Conic currently makes it easy to exploit this insecurity!

Related to security, is the concept of a management domain [Sloman 87]. The configuration system currently manages *systems* which are disjoint sets of logical nodes. We do not support the interconnection of nodes in different systems. A more complex view, applicable to very large systems, would be the division of a system into management domains each containing a set of nodes which could potentially inter-communicate. Responsibility for managing different parts of the system would reside with different users. Authorisation to change a part of the system could be checked before allowing a user to **manage** that part of the system. This would go some way to alleviating the security problem outlined above. The HPC proposal [Le Blanc 85] outlines a similar approach to Conic in the area of management

and specifies a number of possible operations for manipulating domains and process hierarchies. However, as yet no implementation has been reported in the literature.

To date, we have constructed applications consisting of tens of logical nodes. The constraint on system size is largely a function of the servers capacity. It is likely that to accommodate systems with hundreds of nodes, we will have to partition the server function into a number of logical nodes and exploit locality to reduce the communication overhead as is done in the Clearinghouse nameserver [Oppen 83].

7. CONCLUSIONS

Conic has been used at Imperial College, other universities and in industry for implementing communication protocols, operating systems, image processing, adaptive control, distributed discrete event simulation, distributed databases etc. It is gratifying that all our users have found the concepts embodied in Conic, and the facilities provided by its support environment, to be easy to assimilate and use. They are particularly enthusiastic about the use of the configuration language to describe and construct their systems and about **dynamic configuration** using logical nodes. The functionality provided seems to be more than adequate to support the flexibility required in distributed systems.

The separation of programming from configuration has enabled us to maintain the knowledge of the configuration structure and status necessary to make unpredicted configuration changes. It is difficult to envisage how such arbitrary changes can be incorporated in a system where configuration information and control is embedded in the programming language and hence in the program. Planned changes in Conic, such as in response to failures, can be initiated from the programming level by communication with the configuration manager. However, further work is required to investigate the interaction between running programs and the process of dynamic configuration. Identification of the possible points for reconfiguration is related to the notion of module quiescence, where a module is inactive and awaits stimulus before performing further actions. Previous work [Kramer 78] using invariants to characterise module quiescence appears promising.

The selection of **simple** and **efficient** primitives for Conic have provided a sound basis for the implementation of experimental distributed systems. Where functionality was sacrificed for simplicity and/or efficiency, more complex operations can generally be provided at a higher level. For example we have provided atomic transactions by extending the standard facilities provided by the executive [Anido 86] rather than as base primitives as in Argus [Liskov 83]. We have also experimented with the use of passive module redundancy and the reconfiguration facilities to provide fault-tolerance in a transparent manner [Loques 86].

Support for **mixed hosts / targets** has provided an extremely versatile environment. The fact that operational distributed targets can communicate with Conic logical nodes running under Unix has obviated the development of standard facilities such as a file system or printer spooler. It has allowed us to keep targets simple as the complex components of the Conic support environment can run on the host computers. In addition, the ability to test distributed systems on a Unix host prior to down-line loading to a distributed architecture, has speeded up the development process in many cases.

The **uniformity** provided by the use of Conic itself for implementation of the support environment, has proved useful in tailoring the facilities provided. For example the communication system can be configured to include a connection service, routing over interconnected subnets or drivers for different LANs. In addition, the **accessibility** of the system facilities ("open architecture") has even permitted users to adapt and modify the executive to support their requirements. For example, in their development of a run-time environment for an object-oriented system, GEC Research have modified some of the Conic intertask communication primitives and introduced support for manipulating capabilities.

As explained, the environment supports allocation **flexibility** and provides the necessary transformations (**portability**) for a restricted set of non-homogeneous computers. Structuring the executive as Conic modules has meant that the standard Conic configuration tools can be used to build the run-time system for the variety of hosts and targets. It would have been difficult to maintain and support this variety of machines any other way. However, the environment currently supports only a single programming language. This has the advantage that the compiler can check message type compatibility between messages and ports and that port interconnections can be validated for type compatibility at configuration time. Therefore no run time checks are needed. Furthermore, the transformations required

for transferring messages between heterogeneous computers are comparatively simple as the compiler generates similar data structure representations in different target computers. Some current work, based on that of Matchmaker[Jones 85] and MLP [Hayes 86] is aimed at supporting additional module programming languages. The Conic configuration facilities will provide the basis of integrating diverse language components with those implemented in Conic.

Our future work is mainly centred on investigating the expressive power of configuration languages and support for dynamic configuration. We propose to investigate the use of guarded configurations to cater for conditional situations and recursion, and to examine the use of configuration constraints, properties which should be preserved across configuration changes. We also intend to continue to use Conic and the basis for more general distributed system research such as software heterogeneity, distributed algorithms, fault tolerance and security in management domains.

As can be seen from the above description, Conic provides a flexible and sound environment for the implementation of experimental distributed systems, both to ourselves and our various users. Conic has benefitted from user experience and we intend to continue this fruitful partnership.

7. REFERENCES

Andrews 86 G. Andrews, R. Olsson, "The evolution of the SR programming language", Distributed Computing, 1, July 1986, pp. 133-149.

Anido 86 R. Anido, J. Kramer, "Synchronised forward & backward recovery", 7th. IFAC DCCS, Germany, Sep. 1986, to be published by Pergamon Press.

Black 87 A. Black, N. Hutchison, E. Jul, H. Levy, L. Carter, "Distribution and abstract types in Emerald", IEEE Trans. on Software Eng. SE-13(1), Jan. 1987, pp. 65-76.

Cheriton 84 D.Cheriton, "The V-Kernel a software base for distributed systems", IEEE Software, 1 (2), April 1984, pp. 19-43.

Dulay 84 N.Dulay, J.Kramer, J.Magee, M.Sloman, K.Twidle, "The Conic configuration language, version 1.3", Imperial College Research Report DoC 84/20, November 1984.

Gawthrop 84 "Implementation of distributed self-tuning controllers", EUROCOM 1984, Brighton, Peter Peregrinus, pp384-352.

Hayes 86 R. Hayes, R.D. Schlichting, "Facilitating mixed language programming in distributed systems", TR 85-11a Dept. of Computer Science, University of Arizona, Tucson 85721, March 1986.

Hoare 78 C.A. R. Hoare, "Communicating sequential processes," CACM, 21(8), Aug. 1978, pp. 666-677.

Jones 85 M. Jones, R. Rashid, M. Thomson, "An interface specification language for distributed processing", Proc. 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages., ACM Jan. 1985.

Kramer 78 J.Kramer,R.J. Cunningham,"Towards a notation for the functional design of distributed processing systems", in IEEE Proc. 1978 Int. Conf. Parallel Processing, Aug. 1978, pp 69-76.

Kramer 84 J.Kramer, J.Magee, M.Sloman, K.Twidle, N.Dulay, "The Conic programming language, version 2.4", Imperial College Research Report DoC 84/19, October 1984.

Kramer 85 J.Kramer, J.Magee, "Dynamic configuration for distributed systems", IEEE Transactions on Software Engineering, SE-11 (4), April 1985, pp. 424-436.

Le Blanc 85 T.J. Leblanc, S. A. Friedberg, "HPC a model of structure and change in distributed systems", IEEE Trans. Comp., C-34 (12), Dec. 1985, pp. 1114-1129.

Leffler 83 S. Leffler, S. Fabry, W. Joy, "A 4.2 bsd communications primer", Computer Systems Research Group, Univ. of California, Berkley, July 1983.

Liskov 83 B.Liskov, R.Sheifler, "Guardians and actions: linguistic support for robust distributed programs", ACM TOPLAS, 5 (3), July 1983, pp. 381-404.

Liskov 85 B. Liskov, M. Herlihy, L. Gilbert, "Limitations of remote procedure call and static process structure for distributed computing", MIT Lab. Computing Science,Cambridge MA 02139,, Programming Methodology Group Memo 41, Sept. 1984, revised Oct. 1985.

Liskov 87 B. Liskov, L. Shrira, "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems", MIT Lab. Computing Science,Cambridge MA 02139, Aug. 1987

Loques 86 O. Loques, J. Kramer, Flexible fault tolerance for distirbuted computer systems" IEE

Proc. pt. E, 133(6), Nov. 1986, pp. 319-337.

Mullender 86 S.J. Mullender, A.S. Tanenbaum, "The Design of a Capability Based Distributed Operating System", Computer Journal, Vol. 29 No.4, Aug 1986, pp. 289-299.

Oppen 83 D.L. Oppen, Y.K. Dalal, "The Clearinghouse: a decentralised agent for locating named objects in a distributed environment," ACM Trans. on Office Systems, 1(3), July 1983. pp. 230-253.

Plummer 82 D. Plummer, "An Address Resolution Protocol (RFC 826)", Nov. 1982.

Postel 83 J. Postel, "User Datagram Protocol (RFC 768)", Information Sciences Institute, University of Southern California, 4376 Admiralty Way, Marina del Ray Calif. 90291.

Redel 80 D. Redel et. al. "Pilot: an operating system for a personal computer", CACM 32(2), Feb. 1980, pp. 81-92.

Scott 87 M.L. Scott, "Language support for loosely coupled distributed programs", IEEE Trans. on Software Eng. SE-13(1), Jan. 1987, pp. 77-86.

Sloman 86 M.Sloman, J.Kramer, J.Magee, K.Twidle, "Flexible communications for distributed embedded systems", IEE Proc. Pt. E, 133(4), July 1986, pp. 201-211.

Sloman 87 M. Sloman, "Distributed systems management", IFIP TC 6.4 LAN Management Workshop, Berlin, July 1987, North Holland.

Strom 85 R. Strom, S. Yemini, "The Nil distributed systems programming language: A status report", ACM SIGPLAN Notices, 20(5), May 1985, pp. 36-44.

Sun 85 "External Data Representation Reference Manual (Part 800-1177-01, Rev. A-ß)", Sun Microsystems Inc., Mountain View, Ca, Jan 1985.

Tanenbaum 83 A.Tanenbaum, H.van Staveren, E.Keizer, J.Stevenson, "A practical toolkit for making portable compilers", CACM 26 (9), Sep. 1983, pp. 654-662.

USA DOD 80 USA Department of Defense, "Reference manual for the Ada™ programming language", Proposed Standard Document, July 1980.

Wegner 84 P. Wegner "Capital intensive software technology", IEEE Software, 1(3), July 1984, pp. 7-46.

Wirth 77 N. Wirth, "Modula: a language for modular multiprogramming", Software Practice and Experiences, 12, 1982, pp. 719-753.

Xerox 81 "Courier: the remote procedure call", XSIS 038112, Xerox OPD, 333 Coyote Hill Rd., Palo Alto, Ca 94304, 1981.