IEEE TSE 15, 6, June 1989.

CONSTRUCTING DISTRIBUTED SYSTEMS IN CONIC

Jeff Magee, Jeff Kramer, Morris Sloman

Department of Computing, Imperial College of Science and Technology,
180 Queen's Gate, London SW7 2BZ.

Abstract **–** Existing distributed systems vary from those which merely provide interconnection of autonomous systems to those which provide a complete language environment for writing distributed programs. The former tend to support flexibility and provide ready access to system facilities, but suffer by being complex to use. Language environments are simpler to use and can provide safer environments by performing checks, but tend to be aimed at constructing distributed programs rather than systems, and tend to hide and prevent access to many system level facilities. Both tend to be weak in their support for the configuration and modification of distributed applications.

The Conic environment provides a language-based approach to the building of distributed systems which combines the simplicity and safety of a language approach with the flexibility and accessibility of an operating systems approach. It provides a comprehensive set of tools for program compilation, configuration, debugging and execution in a distributed environment. The environment is particularly strong in its **configuration** facilities. A separate configuration language is employed to specify the configuration of software components into logical nodes. This provides a concise configuration description and facilitates the re-use of program components in different configurations. Applications are constructed as sets of one or more interconnected logical nodes. Arbitrary, incremental change is supported by **dynamic configuration**, the capability to dynamically create, interconnect and control logical nodes. In addition, the system provides user transparent datatype transformation between heterogeneous processors. Applications may be run on a mixed set of interconnected computers running the Unix™ operating system and on bare target machines with no resident operating system.

This paper sets out the basic principles adopted in the construction of the Conic environment and, in particular, describes the configuration and run-time facilities provided. Examples are used to illustrate the approach.

Index Terms **–** Configuration language, configuration management, dynamic configuration, distributed systems, networked systems, programming language, operating system, run-time system.

™ Unix is a trademark of AT&T Bell Laboratories

1. INTRODUCTION

While the advantages of the use of distributed systems are well known and widely acclaimed, there is still little agreement as to how to provide the necessary support for modularity, concurrency, synchronisation, communication and configuration. The approaches taken vary from those attempts to merely adapt and interconnect existing autonomous systems to those which provide a complete language environment in which to write distributed programs.

For instance, many Operating Systems (OS) provide direct access to communication facilities. Examples of this approach include the SNA LU 6.2 interface in IBM operating systems [10], the DECNET NSP interface in DEC operating systems [34], and the socket interface to TCP/IP protocols in most Unix™ systems. A distributed application is implemented as a collection of sequential programs which communicate using the relevant networking system calls. However, the communication interfaces are complex and difficult to use. The naming conventions and interprocess communication primitives are usually non-uniform, using different conventions and providing different semantics for internal and remote interactions. Little support is provided by the OS environment for initial configuration of a set of program components into an executable distributed application, nor for subsequent monitoring and control of the configuration. Similarly, little or no interface checking is supported to ensure compatibility of interconnected programs. Applications programmed in this way are thus difficult to construct, debug and maintain. The main advantage of the OS approach is that it is **flexible**, in that a distributed application is composed of a (potentially) changing set of interconnected programs.

On the other hand, distributed programming languages [30,9,1,3] reduce the complexity of constructing distributed applications by providing modularity, concurrency, synchronisation and communication facilities integrated into a single language framework. They provide support for compile, link and run-time checks to ensure operation or message compatibility between components. In addition they provide consistent naming, communication and synchronisation for both local and remote interactions. Thus language environments are generally **simpler** to use and can provide **safer** environments. However, configuration facilities are often part of the programming language which results in a single large distributable program rather than the OS view of a system as a changing set of interconnected programs. We believe that this makes unpredicted modification and the provision of redundancy more difficult. In many applications, particularly real-time ones, it is useful to have a set of components which form a unit for configuration or failure recovery, and which can be separately reconfigured.

Conic provides a language-based approach to the building of distributed applications which combines the simplicity and safety of a language approach with the flexibility of an

operating systems approach. 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 separate configuration language provides a concise configuration description and hierarchical composition, and is employed to specify the configuration of software modules (processes) into logical nodes. A ***logical node*** is the system configuration unit. It is a set of tasks which execute concurrently within a shared address space. Systems are constructed as sets of one or more interconnected logical nodes.

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** [14] of logical nodes. This permits on-line modifications to a running Conic system using the configuration language.

Conic was designed for the support of distributed embedded systems, but in practice has been used to construct a wide range of applications from general distributed algorithms to system support utilities and services. The flexibility objectives of Conic are similar to those of LYNX [26] in providing language support for loosely coupled distributed programs; however we have not concentrated on the client-server paradigm of system construction, but have provided support for general peer (mesh) interactions. The example in section 3 illustrates this.

A. Design Principles

As described above, we believe that distributed systems require a combination of language supported convenience and safety with operating system flexibility. The flexible configuration facilities described above are essential, particularly in an experimental environment. Other principles which we believe to be important are support for mixed host and target environments, uniformity, simplicity and efficiency, and portability. These are briefly discussed below.

A host environment provides many useful services and utilities. Targets are useful in providing the distributed processing power and real-time response required for interaction with devices. In order to exploit both, distributed applications should be capable of running in **mixed host and target** environments. For instance, applications in the areas of factory automation, process control and telecommunications require major parts of the system to exhibit real-time response, but also require access to file servers, graphic displays, logging and printing services. A Conic application, consisting of one or more interconnected logical nodes,

can be configured to run in such a mixed host-target environment. Fig. 1 depicts a typical execution environment. A Conic logical node may be executed on a host as a Unix™ process or directly on a target. 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.

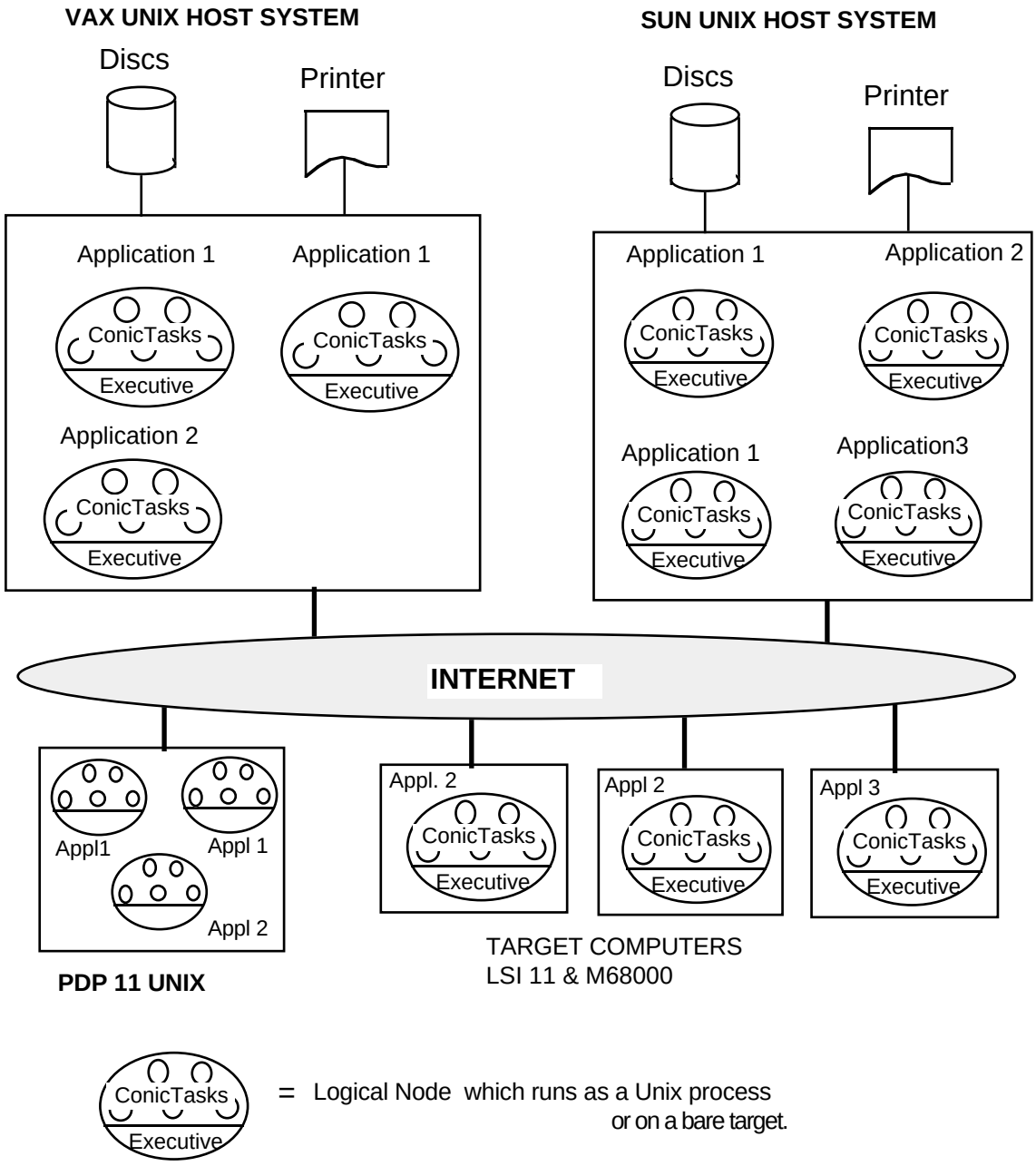

Fig. 1. Distributed applications in a Conic environment

The Conic system and its environment is "**open**" in that it provides easy access to all its facilities [24] by use of a common message passing interface structure for all component

interaction (see section 2). 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.

The underlying Conic support system has been designed to be **simple and efficient.** Wherever a design decision has occurred, it has been our experience that it is better to provide constructs which incur no hidden costs. In an open system such as ours, it is more important to provide extensible facilities which permit more complex facilities to be implemented 'on top' where required, rather than forcing users to pay the cost of powerful primitives even when they are not used. Furthermore, configuration of the system permits selection of those run-time facilities actually required.

The Conic development facilities and the run-time support was designed to be **flexible** and **portable** in the sense of software allocation and in handling computer heterogeneity. This portability across a variety of both host and target processors reflects both academic and industrial requirements. Allocation flexibility allows software to be developed and tested on a single machine and then distributed across mixed host and target computers. This requires that local and remote communication have the same semantics, so that reallocation of software does not change the logical behaviour. The programming language approach of Conic has allowed the automation of the data transformations required for communication between non-homogeneous computers. This is similar to the approach of typed remote procedure calls as in Courier [36] and contrasts with the need for explicit calls on subroutine libraries (e.g. Sun XDR [31]) in a typical OS.

The rest of this paper concentrates on the Conic support environment. In section 2 we briefly outline the important features of the Conic programming and configuration languages. Section 3 describes how distributed applications are constructed using the dynamic configuration tools. The run-time support environment (node executive, configuration manager and server) is described in section 4. Finally, we discuss experience in using Conic and present some conclusions.

II. THE CONIC PROGRAMMING AND CONFIGURATION LANGUAGES.

Conic provides a language based approach to building distributed systems which clearly distinguishes between the programming of individual software components and the building of systems from these components. In this section we give an overview of these languages.

A. Conic Module Programming Language

The language for programming individual software components (modules) is based on Pascal which has been extended to support modularity and message passing primitives [13]. The language allows the definition of a *task module type* which is a self-contained, sequential task (process). At configuration time, *module instances* are created from these types. Module instances exchange messages and perform a particular function such as controlling a device or managing a resource.

The Module interface is defined in terms of strongly typed ports which specify all the information required to use the module. An *exitport* denotes the interface at which message transactions can be initiated and provide a local name and type holder in place of the source name and type. An *entryport* denotes the interface at which message transactions can be received and provides a local name and typeholder in place of the source name and type. The binding of an exitport to an entryport is part of the configuration specification and can only be performed within the programming language by sending messages to the configuration management facilities (described later).

The Conic task module thus 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 ie. Conic modules are *reuseable* in many different situations.

The programming language supports communication primitives to *send* a message to an exitport or *receive* one from an entryport. The message types must correspond to the port types. There are two classes of message transaction:

i) 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.

ii) A **Request Reply** provides bidirectional synchronous message passing. The sender is blocked until the reply is received from the responder. A fail clause allows the sender to withdraw from the transaction on expiry of a timeout or if the transaction fails. The receiver may also block waiting for a request. As an alternative to replying, the receiver of a message can **forward** it via an exitport to another task.

Definition Units are used to define constants, types, functions and procedures which are common between different modules within a system. These can be compiled separately and imported into both task modules and other definition units. Definition units may also define

data and initialisation code and so provides a facility similar to Modula-2 [35] modules and Ada® [33] packages.

The following example of a task module (Fig. 2.1) which acts as scaling filter for its inputs gives the "flavour" of Conic programs.

```
task module scale(scalefactor:integer);
    entryport
        control: boolean;
        input:  real reply signaltype;
    exitport
        output: real reply signaltype;
    var
        value: real;
        active: boolean;
begin
    active := false;
    loop
        select
            receive active from control
        or
            when active
            receive  value from input reply  signal =>
                send value/scalefactor to output wait signal;
        end
    end
end.
```

Fig. 2.1 - Task module

The *scale* task of Fig. 2.1 receives real values on its entryport *input* and sends scaled values to the exitport *output* when the boolean variable *active* has the value *true*. The value of *active* is set by boolean values received from the entryport *control*. *Input* and *output* are request-reply ports, where the reply type *signaltype* is a base type of zero length. The variable *signal* of type *signaltype* is automatically declared by the compiler. The example shows the abbreviated form of the receive-reply statement since no statements are executed between receiving the request and replying. Receive and reply may be separated by processing, in which case the reply in this example would become ***reply*** *signal* ***to*** *output*. The entryport *control* is a notify port with a default buffer queue length of 1. The declaration,

```
control : boolean queue  8;
```

would declare a buffer queue for 8 boolean values. The default buffer exhaustion strategy is to overwrite the oldest buffer, reflecting the most common uses of the notify transaction which are event signalling and status updating. [28] discusses the interprocess communication primitives in more detail. Note that parameters, such as *scalefactor* in Fig. 2.1, can be passed to a task instance at creation time to tailor it for a particular environment.

® Ada is a registered trademark of the U.S. Department of Defence.

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 [4] and Amoeba [20].

B. Conic Configuration Language

The Conic configuration language [5] is used to specify the configuration of tasks which constitute a logical node. A 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. 2.2 describes a group module composed of the two task types *scale* (from Fig. 2.1) and *sensor*. 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 two instances of the task type *sensor* (*temperature* and *pressure*) and two instances of the task type *scale* (*Tscale* and *Pscale*) are specified. The **link** construct declares the interconnections between instance exitports and entryports.

```
group module monitor(Tfactor,Pfactor:integer);
    exitport
        press, temp: real reply signaltype;
    entryport
        control:boolean;
    use
        scale; sensor;
    create
        temperature: sensor;
        pressure: sensor;
        Tscale: scale(Tfactor);
        Pscale: scale(Pfactor);
    link
        temperature.output to Tscale.input;
        pressure.output to Pscale.input;
        Tscale.output to temp;
        Pscale.output to press;
        control to Tscale.control, Pscale.control;
end.
```

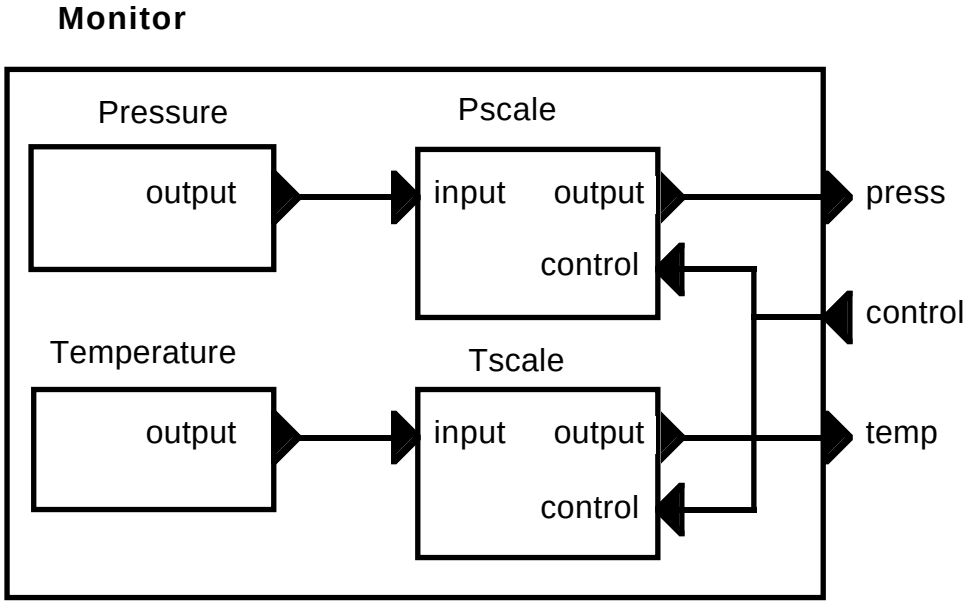

Fig. 2.2 - Monitor group module

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.

Each group module specification is separately compiled into a symbol table and a procedure 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 4.) 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.

III DYNAMIC CONFIGURATION

Distributed programs in Conic are constructed with the aid of the dynamic configuration tools from sets of pre-compiled logical node types. A logical node may run either as a Unix process or on a standalone target depending on the run-time support modules which are configured into it. 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 control the numbers of tasks created within nodes and the values passed to those tasks.

To illustrate the program construction process in Conic, the following outlines the

construction of a simple distributed application. The application supports the multi-screen display of a moving text "snake". The *snake* when it reaches the edge of one screen moves to the beginning of the next. Each screen supports one or more *segments* which are horizontal paths along which the snake may move. *Segments* have a direction indicating whether the snake moves from left to right or right to left across the screen. The diagram of Fig. 3.1 illustrates a three screen display into which two *snakes* have been injected. Each screen has two segments (top segment- left to right, bottom segment - right to left). These segments are connected together to form a ring so that when a *snake* has been injected it continuously travels around the three screens. The snake in Fig. 3.1 has the string value "O____=".

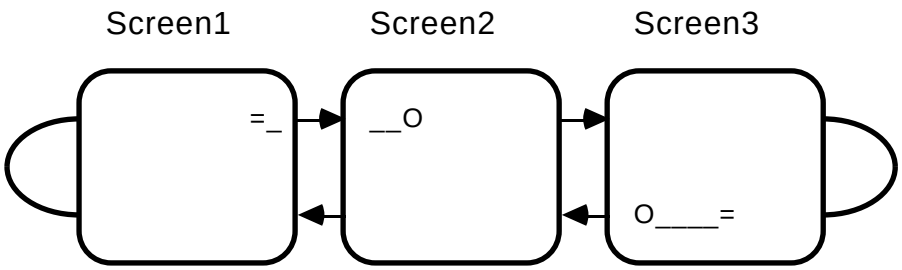

Fig. 3.1 - Multi-screen "snake" display

Each segment may be in one of four states: a snake may be entering the segment, a snake may be leaving the segment, a snake may be travelling across the segment or the segment may be empty. Snakes are transferred between segments one character at a time. Analogous to trains and sections of railway track, a segment may only have one snake entering, leaving or resident at any one time. While artificial, this example raises configuration issues which we have encountered in "real" applications in the areas of flexible manufacturing and control systems. It is felt that the exposition overhead of these real domains would obscure the issues of interest.

A. Task Programming

It is natural to implement the functionality of a segment as a task type in Conic so that a display configuration can be constructed by interconnecting instantiations of this task type. The segment task (Fig. 3.2) takes two parameters, *Ypos* which determines the horizontal position of the segment on a VDU screen and *direction* which determines the direction the segment will move the snake across the screen (*direction = 0* gives right to left and *direction = 1* gives left to right). Snakes are prefixed by the ASCII character *SOH* and terminated by the character *ETX*. Characters for a snake entering the segment are received from the entryport *input* and snakes leave the segment via the exitport *output*.. The definition module *segdisplay* supplies procedures for initialising the segment display and displaying snake movement. The function *movesnake* moves the snake one position each time it is invoked, accepting the next input character as a parameter. It returns NULL characters until the snake reaches the segment

boundary and then returns the characters which constitute the snake.

```
1  task module segment( Ypos,direction:integer);
2  use
3       ascii:soh,etx,nul;
4       segdisplay: initseg, movesnake;
5  entryport
6       input: char reply signaltype;
7  exitport
8       output:char reply signaltype;
9  var
10      ch:char;
11      state:(idle,entering,moving,leaving);
12 begin
13       initseg(Ypos,direction);   state:=idle;
14      loop
15          select
16              when (state=idle)  receive ch from input reply signal
17              =>  if ch=soh then begin
18                          movesnake(soh);
19                          state:=entering;
20                          end;
21          or
22              when (state=entering)  receive ch from input reply signal
23              =>  movesnake(ch);
24                  if ch=etx then state:=moving;
25          or
26              when (state=entering)  timeout 100
27              => initseg(Ypos,direction); state:=idle;
28          or
29              when (state=moving)
30              =>  if ch<>soh then
31                          ch:=movesnake(nul)
32                  else
33                      send ch to output
34                          wait signal => state:=leaving;
35                          fail => { retry };
36                      end;
37          or
38              when (state=leaving)
39              =>  ch:=movesnake(nul);
40                  send ch to output
41                      wait signal=> if ch=etx then state:=idle;
42                      fail => initseg(Ypos,direction); state:=idle;
43                  end;
44          end;
45      end;
46 end.
```

Fig. 3.2 - Segment task

The task program takes the form of a guarded command which is repeatedly executed. The arms of the guarded command are the actions performed for each of the states (idle, entering, moving, leaving) which the segment can take. To avoid fragmented snakes occurring because of either communication failures or re-configuration the segment is re-initialised if a failure occurs in the entering or leaving states (lines 26,27,42}

The only other active component required for this example is a task to generate snakes.

The task of Fig. 3.3 generates a snake when its enclosing node is started and subsequently moves the node back into the stopped state.

```
1   task module snakegen(snake:string);
2   use
3       ascii : SOH, ETX;
4       strings :strlen;
5   exitport
6       out:char reply signaltype;
7   var
8       i : integer;
9   begin
11      while not linked(out) do delay(100);
12      send SOH to out wait signal;
13      for i := 1 to strlen (snake)
14          send snake^[i] to out wait signal;
15      send ETX to out wait signal;
16  end.
```

Fig. 3.3 - Snake generator task

B. Group Modules and Logical Nodes

As stated above, Conic distributed applications are constructed from logical node types. Logical node types are constructed from task types using the Conic Configuration Language. The snake display example can be constructed from two logical node types: *screen* - which contains one or more instances of the *segment* task and *generator* - which contains one instance of the *snakegen t*ask. The configuration language descriptions and diagrammatic representations for these logical nodes are depicted in Figs. 3.4 and 3.5.

```
group module screen(N:integer=2;  spacing:integer=8);
use
    unixexec;
create
    unixexec;
entryport
    input [1..N] : char reply signaltype;
exitport
    output [1..N] : char reply signaltype;
use
    segment;
create family k:[1..N]
    seg [k] : segment (ypos = k*spacing, direction = k mod 2);
link family k:[1..N]
    seg[k].output to output[k];
    input[k] to seg[k].input;
end.
```

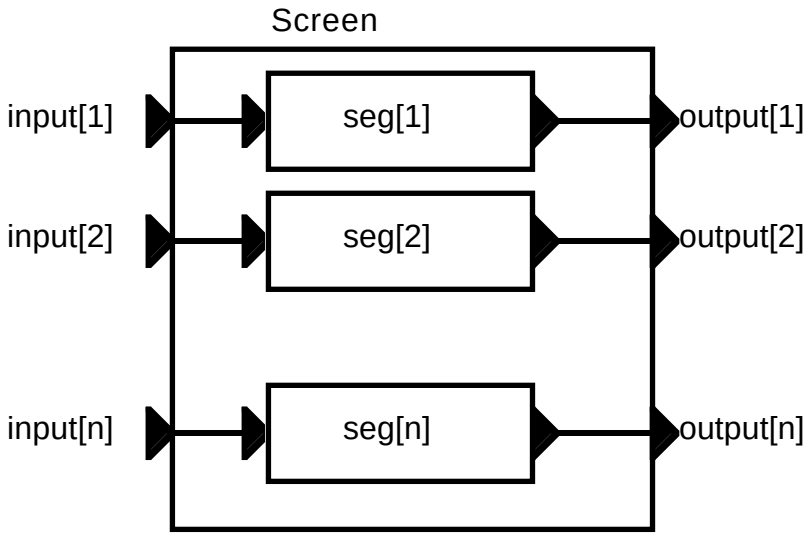

Fig. 3.4 - Screen logical node

In addition to application tasks, logical nodes contain the run-time support necessary for the environment in which they are to execute. Both *screen* and *generator* are intended to run under a Unix host operating system and consequently they include an instance of the group module *unixexec* which supports multi-tasking, message passing and dynamic configuration operations in conjunction with Unix. The structure of run-time support is described in the next section.

The Conic Configuration language supports default parameter values. For *screen*, the default number of segments is 2 and the default spacing between segments on the VDU display is 8 lines. The create statement specifies a family of *N* segment instances with odd numbered segments having the direction left to right and even segments having the direction right to left. The default value of *N* can be overidden by passing a value to the node at creation time.

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 (in this case *unixexec*).

```
group module generator(s:string="O_____=");
use
    unixexec;
create
    unixexec;
exitport
    output:char reply signaltype;
use
    snakegen;
create
    snakegen(s);
link
    snakegen.out to output;
end.
```

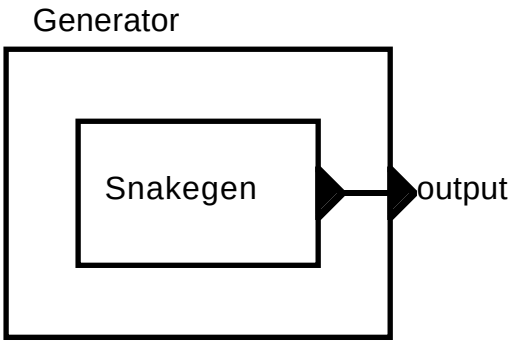

Fig. 3.5 - Generator logical node

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.

C. 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. To construct the snake display example we have two logical node types, *screen* and *generator*. The display of Fig. 3.1 could be configured to run on three VDU devices connected to one host or on three windows on a single Sun workstation. In the following, we will describe how the display can be mapped onto the hardware configuration of three Sun workstations depicted in Fig. 3.6

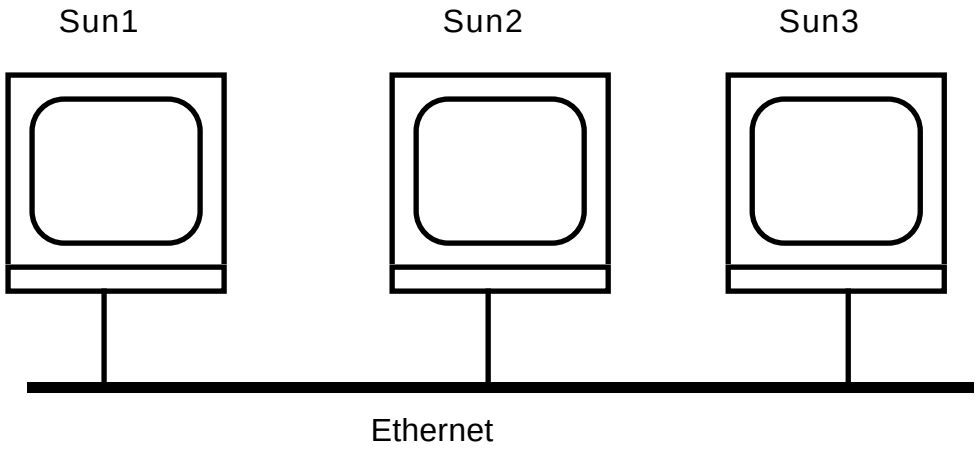

Fig. 3.6 - Hardware configuration

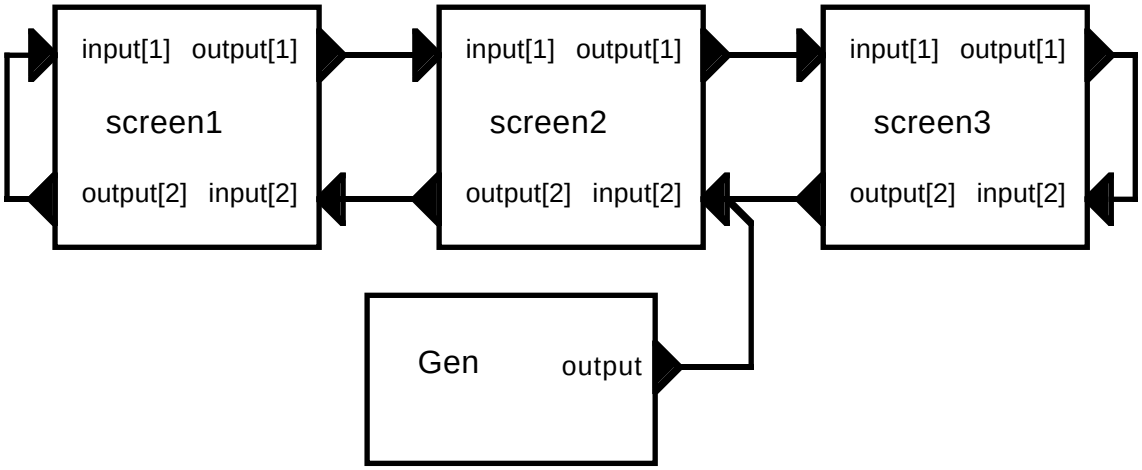

Fig. 3.7 - Logical configuration

The logical configuration shown diagrammatically in Fig. 3.7 is constructed by submitting the following set of configuration statements 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.

Configuration statements:-

```
manage snakedemo

create screen1:screen at sun1
create screen2:screen at sun2
create screen3:screen at sun3
create gen:generator at sun2

link screen1.output[1] to screen2.input[1]
link screen2.output[1] to screen3.input[1]
link screen3.output[1] to screen3.input[2]
link screen3.output[2] to screen2.input[2]
```

```
link screen2.output[2] to screen1.input[2]
link screen1.output[2] to screen1.input[1]

link gen.output to screen2.input[2]
```

The **manage** statement 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 users Unix login name.

The **create** statement creates the specified logical node type **at** a location. In this example *screen1* is created at *sun1*, *screen2* and *gen* at *sun2* and *screen3* at *sun3*. Each instance of the *screen* logical node type has been created with its default parameters. A *screen* with four segments and different spacing between segments could be created with the statement:

```
create bigscreen:screen(4,6) at sun1
```

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 statements can be executed directly by Unix **csh** as commands. The commands invoke the manager with their names as parameters in the standard Unix fashion. Consequently, cunning **sh** macros can be defined to shorten the text of configuration descriptions (such as the list of link statements above).

Additional snakes can be injected into the system by the commands:

```
create gen2("O***=):generator at sun2
link gen2.output to screen2.input[2]
```

Generators terminate and dissappear as soon as they have completed injecting a snake.

An additional screen can be added to the right of the loop by the following set of configuration statements:

```
unlink screen3.output[1] from screen3.input[2]
create screen4:screen at sun3
link screen3.output[1] to screen4.input[1]
link screen4.output[1] to screen4.input[2]
link screen4.output[2] to screen3.input[2]
```

As described in the next section, the above **create** will both instantiate *screen4* and also create an additional Sun window for *screen4* to run in.

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.
ports <node> - lists a node's interface ports and types
links <node> - lists the entryports connected to a node's exitports.

D. 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 Berkley 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 [18] 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.

This section has concentrated on the structural aspects of constructing a distributed application. We have largely ignored aspects of application consistency. For example, segments make no effort to preserve snakes during re-configuration or to avoid deadlock when accepting new snakes. The preservation of consistent system state during reconfiguration requires synchronisation between the management system and the distributed application. Recent work [HK,DS] has outlined a protocol for performing this synchronisation which preserves the configuration independence of modules.

IV. 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 [16,23,6] offered by BSD4.3 and 2.9. 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. Reports from a group of users who are porting Conic to a VAX/VMS host environment indicate that this has proved reasonably successful.

The execution environment on which our development system runs at Imperial College consists of VAXs, Sun Workstations and some aging PDP11s running various versions of Berkeley UNIX and interconnected by Ethernet (see Fig. 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 Ethernet) are available for applications which require real-time response. Typically these targets are used for controlling real-time control experiments. The compilation system supports cross-compilation from the Suns and VAXs to PDP11 targets. Although possible, to date there has been no requirement for cross-compilation between VAXs and 68000s. 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, both the structure of the run-time environment and the rationale behind its design are outlined.

A. Configuration Management

Our initial conception of dynamic configuration management [14] 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.

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 (Fig. 4.1). 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.

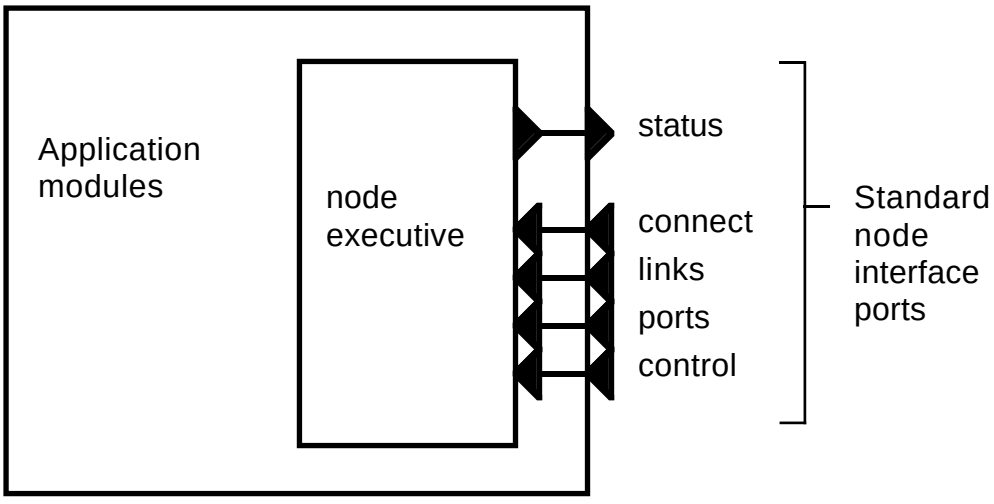

Fig. 4.1 - Node standard interface ports

The services provided by the node's management interface entryports are as shown in Fig. 4.1, 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. 4.2.

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, UDP/IP address and its configuration state.

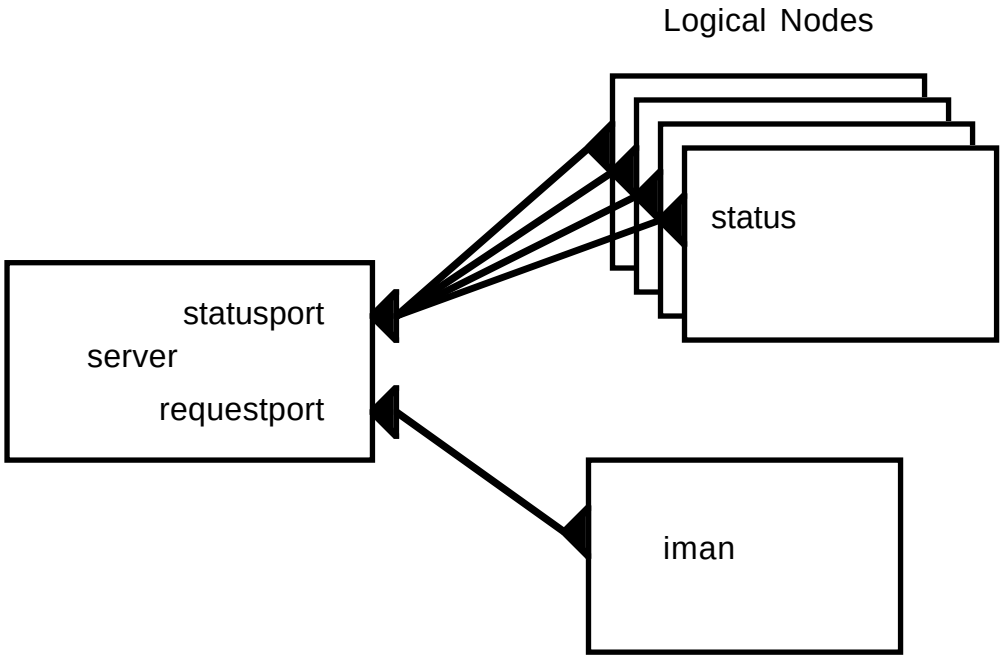

Fig. 4.2 - Configuration name server

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.

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 Fig. 4.2 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 to be managed 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 the Sun Workstation of Fig. 4.3 from a manager running on the VAX would type the commands:

manage snakedemo

create newscreen:screen **at** sun1

The manager locates the virtual target node *sun1* by communicating with the name server, links to it, and sends a message containing the string:

"screen - newscreen snakedemo".

The virtual target *sun1* then executes this command in the usual UNIX way (fork & exec).

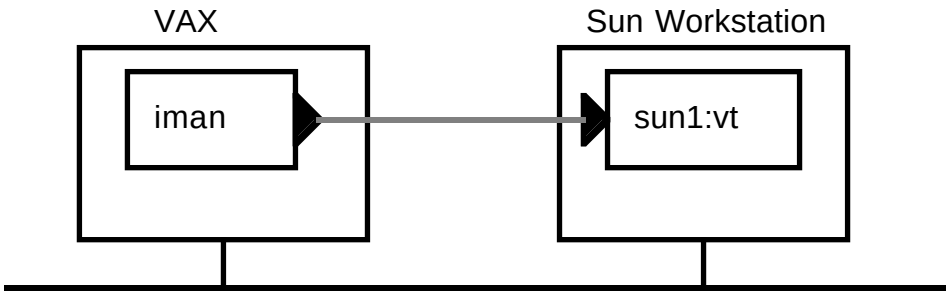

Fig. 4.3 - Remote creation

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 the virtual target to create a Sun Window for each node it instantiates. In this case, the virtual target is created on the Sun with the command:

vt shelltool - sun1 snakedemo

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 **sh** macro as an argument. This macro

would copy the code from a remote location using **rcp** and then execute it.

B. 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 Fig. 4.4. 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 in the following.

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) [22] to translate Internet addresses to Ethernet addresses. The particular Ethernet driver included in the target manager depends on the details of target hardware. A more detailed description of Conic communications may be found in [27].

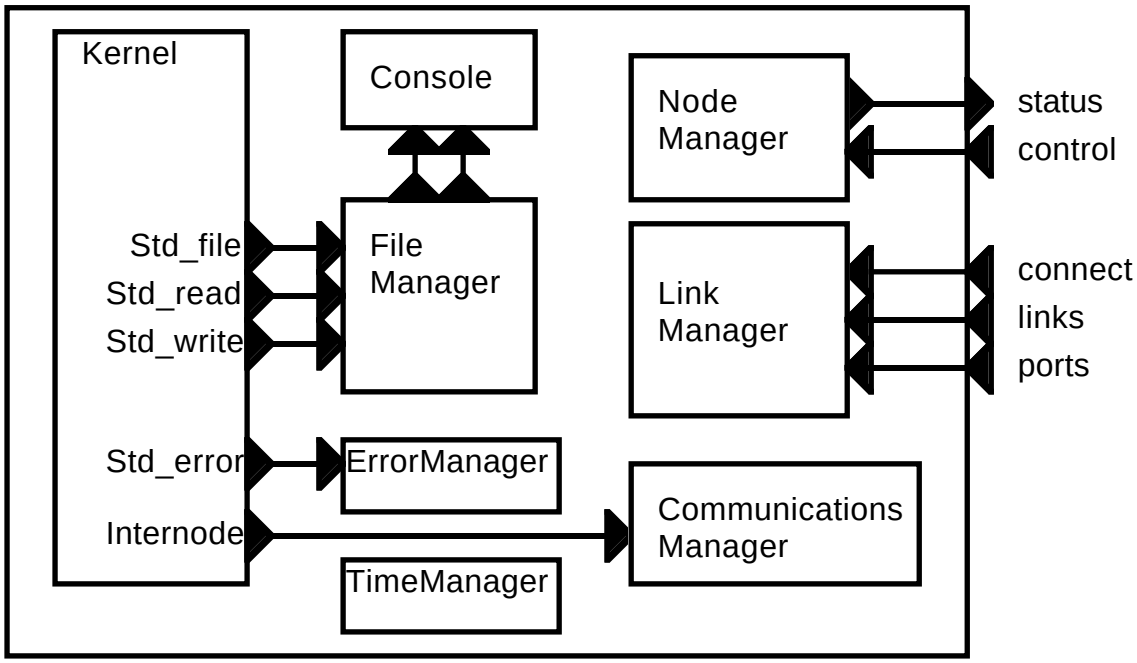

Fig. 4.4 - Generic node executive

The *FiIe 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 4.1. 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* 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 Fig. 4.5 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.

	Inter-node Intra-node	Inter-node (intra-host)	(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. 4.5 - Inter-task communication performance

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

C. 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) [32]. 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[31] 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 2 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.

D. *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 4.1.) 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 [20] 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 [28]. 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 potentially could 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 [15] 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 [21].

V. CONCLUSIONS

An earlier version of the Conic environment, with static configuration, has been used for a number of 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 at the University of Sussex for research on self-tuning adaptive controllers [7]. The Conic system has been supplied to universities in Canada, France, Japan, Korea and Sweden.

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 (as opposed to programs).

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.

< need something here about new work , note ref [12] has dissappeared >

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 transactions by extending the standard facilities provided by the executive [2] rather than as base primitives as in Argus [17]. We have also experimented with the use of passive module redundancy and the reconfiguration facilities to provide fault-tolerance in a transparent manner [19].

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 [25].

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 [11] and MLP [8] 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.

< this next needs to change>

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.

ACKNOWLEDGEMENTS

Acknowledgement is made to British Coal for a grant in aid of these studies, but the views expressed are those of the authors and not necessarily those of British Coal. This work

has also been partially funded by the SERC under Grant GR/C/31440. We particularly acknowledge the contribution of our colleagues Naranker Dulay and Kevin Twidle (who provided the first implementation of the snakes example) to the concepts described in this paper and to the implementation of the Conic environment itself. Finally we wish to thank the referees for their helpful comments and suggestions.

REFERENCES

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

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

[3] 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.

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

[5] 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.

[6] "DOD standard internet protocol", ACM Computer Comms. Review, 10(4), Oct. 1980, pp.12-51.

[7] P. Gawthrop, "Implementation of distributed self-tuning controllers", EUROCOM 1984, Brighton, Peter Peregrinus, pp384-352.

[8] 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.

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

[10] Special Issue on SNA, IBM Systems Journal 22(4), 1983.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

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

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

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

[19] O. Loques, J. Kramer, Flexible fault tolerance for distirbuted computer systems" IEE Proc. pt. E, 133(6), Nov. 1986, pp. 319-337.

[20] 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.

[21] 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.

[22] D. Plummer, "An Address Resolution Protocol (RFC 826)", Nov. 1982.

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

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

[25] D.Robinson, M. Sloman, "Domain based access control for distributed systems", Marconi, Research Centre ITM 86/87, GEC Research Ltd, Great Baddow, Chelmsford, CM2 8HN, U.K., Dec. 1986.

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

[27] 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.

[28] M. Sloman, J. Kramer, "Distributed systems and computer networks", Prentice Hall 1987.

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

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

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

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

[33] USA Department of Defense, "Reference manual for the Ada™ programming language",

Proposed Standard Document, July 1980.

[34] S. Wecker, "DNA: the Digital network architecture", IEEE Trans. on Comms., COM 28(4), April 1980, pp. 510-526.

[35] N.Wirth, "Programming in Modula-2", Springer Verlag, 1982.

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