DECK: A new model for a distributed executive kernel integrating communication and multithreading for support of distributed object oriented application with fault tolerance support.

Marcos E. Barreto[*] Philippe O. A. Navaux[†] Michel P. Rivière [‡]

September 21, 1998

Abstract

DECK (Distributed Executive Communication Kernel) is a communication layer that provides support for multithreading and fault tolerance. The approach retained in DECK is close to other distributed communication kernels like PM2, Athapascan, Nexus, TPVM or Chant in its way to integrate communication and multithreading to efficiently overlap communication by computation and provide low latency remote thread creation mechanisms. However, DECK differs from these communication kernels from the services offered and its modular architecture.

The main goal of DECK is to implement a new model for the design of distributed executive kernel to efficiently use the new underlying hardware architectures (SMP architectures and fast communication adapters like Myrinet or memory oriented adapter like SCI) and provide a portable layer that abstract the problems linked with the integration of communication and multithreading while offering support for heterogeneity.

A great lack in the current implementation of communication libraries or distributed executive kernel is the support for basic services at the thread level and support for fault tolerance. Indeed, communication library like PVM or MPI are often used as communication layer to ensure portability and take benefits of specific implementation to ensure a good efficiency on specific architectures however the support for fault tolerance, multithreading, scalability and interoperability are usually not offered.

In the case of DECK, we propose a model where a distributed application can dynamically instantiate clusters of processes among an heterogeneous network of computers or parallel machines and this using multiple communication protocols or communication interfaces to ensure good performances regarding the underlying hardware architecture. The programming model proposed offer both classic synchronous and asynchronous remote service calls for thread creation and message passing for synchronization and data exchange.

These basic functionalities, that form the low level communication and execution layer of DECK, are enforced by a service layer that propose the basic fault tolerant services like naming and group services or data management services for the marshaling and un-marshalling of complex data structures. The layered and modular approach followed by DECK enable many other extensions while keeping a high degree of portability and efficiency.

Keywords : distributed systems, communication and executive kernels, multithreading, fault tolerance.

[*] Professor do Centro Educacional La Salle de Ensino Superior e Mestrando do CPGCC/UFRGS: Processamento Paralelo e Distribuído, Tolerância a Falhas em Sistemas Distribuídos. e-mail: barreto@inf.ufrgs.br

[†] Professor CPGCC/UFRGS: Dr. Eng. em Informática (Instituto Nacional Politécnico de Grenoble - França 1979): Arquitetura de Computadores, Processamento Paralelo, Avaliação de Desempenho. e-mail: navaux@inf.ufrgs.br

[‡] Pesquisador CNPq-CPGCC/UFRGS: Dr. em Informática (Universidade Joseph Fourier de Grenoble - França 1997): Arquitetura de Computadores, Processamento Paralelo, Sistemas Distribuídos. e-mail: riviere@inf.ufrgs.br

1 Introduction

Cluster of workstations are very often used as virtual parallel machines to execute distributed applications as they offer a good ratio performance price. This tendency has increased with the availability of powerful multiprocessors architectures (or SMP) and high-performance communication interfaces such as Myrinet [15], ATM [19], SCI (IEEE 1596).

To support such distributed architectures, many environments have been developed to help programmers to express the parallelism of their applications. Message passing libraries such as PVM [9] or MPI [5] are know available for a wide range of operational systems and architectures. In parallel, specific MSA (Messaging System Architecture) has been developed to support the characteristics of the new generation of high-performance communication interface [15].

An important effort has also been made to use efficiently SMP architectures [18]. Multithreading libraries are now proposed by most systems such as Unix or Window/NT and they are often the key to achieve good performance on these multiprocessor architectures as they begin to be included directly at the system level [6, 1].

However, the integration of communication libraries and multithreading kernel are not an easy task. This is why many distributed executive kernel [13, 10, 11, 3] have been developed to resolve this problems and offer basic functionalities to simplify the creation of threads on multiple nodes of a cluster of workstations and express their interactions by means of communications or synchronizations.

If these distributed executive kernel have succeeded to integrate communication and multithreading to efficiently overlap communication by computation and provide low latency remote thread creation mechanisms they do not support very well scalable high-performance communication network or heterogeneous environment that involve different processor architectures and systems.

The problem is to abstract the utilization of multiple communication devices in the same application using different protocol of communication. In fact, they usually rely on IP to resolve this problem but this solution is not acceptable for many communication sub-systems because the overhead involved in the stack IP is far to high in comparison of native (ad-hoc) communication library (the same argumentation is also valid for distributed executive kernel that use portable communication libraries such as MPI or PVM). Also many communication libraries do not support threads or are not thread aware. This lead to inefficient integration of thread and communication because every access to the communication layer has to be trapped and serialized and communication handling has generally to be done by a dedicated thread [13]

Another problem is that these distributed kernel do not provide support for fault-tolerance. As an example, many communication libraries like MPI [5] or distributed executive kernels that lye on top of MPI as to deal with the fact that if a single process crash all the other application processes have to be terminated.

To resolve these problems we propose a new model for a distributed executive kernel, *DECK*, that respects the following constraints:

- support for heterogeneity
  - offer a portable platform for heterogeneous distributed computing that abstracts the underlying communication devices, the architecture and the operational system.
- good efficiency
  - use efficiently the underlying architecture, the operational system, the communication and thread libraries using an abstract system interface. This include the support for SMP architectures via multithreading and new MSA architectures such as Myrinet or SCI.

- aim scalability and reusability
  - enhance the kernel facilities with basic services to extend the functionalities of the DECK kernel to provide a useful higher level services for the end user or a distributed programming environment.

This paper is divided in three parts and presents the global architecture of DECK, the programming model it offers and the design of a first prototype.

2 Heterogeneous architectures and fault tolerance

This section is dedicated to the presentation of the global DECK architecture. We focus this presentation on the properties needed in order to interconnect an heterogeneous network of computers through a distributed executive kernel and offer a minimum support for fault tolerance.

First, we present our approach to resolve the support of an heterogenous distributed system that is composed of different operating systems and hardware in order to build portable applications that could be compiled without modifications on every machine. We follow this presentation with the definition of the problems generally involved with the support of fault tolerance in distributed executive kernel system like DECK and finally conclude with our solution to integrate such heterogeneous distributed portable kernels and fault tolerance.

2.1 The heterogeneity problem

Support for heterogeneity has usually to be solved at two level in the case of distributed applications. The first level is linked with the interoperability of heterogeneous nodes of a distributed platform. This require generally the design of a common communication protocol or a same communication medium (the most famous examples are IP and Ethernet). The second level has to resolve the problems related to the heterogeneity of the different operational systems and processor architecture that are involved.

Thus, to be able to build a portable application and resolve the problems involved with the heterogeneity the programmer have to ensure the interoperability of the distributed system components and use common systems functionalities. This later point, could be achieved by building an abstract system library that offer a subset of functions that are available on all the systems (or can be simulated) and resolve problems like endianims conversion or 64bits architectures.

This abstract system library, in the case of DECK, could be viewed for example as a collection of module like a thread module, an I/O module or any other basic system functionalities module that are required by the implementation of DECK. Depending on the system, the thread module could use a native thread system library, a POSIX thread library (IEEE 1003.1c) or any other thread library but it exports at the API level a common set of functions on every systems. Using this solution, a programmer can make portable programs and put on the back of the abstract system library the work to solve the problems linked with the heterogeneity. This is the same approach that is used by communication library, like PVM, because they offer a common set of primitives but with different possible implementations whether the system is UNIX or Window/NT for example.

Another interesting point of this approach is that the abstract layer library could benefit from specific system library in order to achieve good performances. If we considered the Solaris systems that offer two thread packages, a native thread package and a POSIX thread compliant library, it is more efficient to use the native thread package [6] rather than the POSIX one (it offers more support for SMP architecture as it is included in the Unix kernel and the system libraries).

2.2 Support for fault tolerance

Support for fault tolerance in distributed systems may be considered from many point of view. In the case of DECK, we do not aim to provide support for fault tolerance directly but rather we want to offer functionalities and basic properties to be able to build on top of this kernel fault tolerant services.

Indeed, we estimate that services like reliable group communication using specific communication protocol and supporting fault tolerance have to be offered by a higher services layer [20]. However, this layer could expect that some basic support like reliable point to point message passing is available. Also, some services to create or restart a process of an application should be offered to leave the fault tolerant services concentrate its work on the implementation of a fault tolerancy rather than deal with the basic services that it needs and resolve all the problems that we have exposed for the support of heterogeneity for example.

In the case of DECK, we want to integrate support for at least these points:

- reliable point to point communication,
- dynamic creation or restart of process and the corresponding naming system facilities,
- basic system facilities to save and restore persistent data.

Other extensions such as reliable collective group communications or checkpoint and rollback facilities are not included in the DECK kernel but may be in the future offered as services by the service layer (cf. section The DECK services layer).

2.3 Integrating fault tolerance and heterogeneity

Support for native communication and specific hardware architecture are the key for performance issues in distributed systems specifically if they lye on top of SMP architectures and proprietary communication interfaces. If it is possible to abstract the low level hardware with an abstract interface like Chameleon [5] it is however more complex to provide at the same time the basic needs to support fault tolerant in distributed executive kernel.

The principal problem comes from the high performance communication libraries (MSA) that usually offer a SPMD programming model. According to this model a single process is usually executed on a node and if a process crash the entire application has to terminate. This is also the case for MPP computer like the IBM-SP2 or the Cray-T3D/E and their native communication libraries (cf. PVMe [2] for the IBM SP series or the shared memory facilities for the Cray).

It is very difficult in this case to build a crash tolerant application if each time a process fail the entire application stop and have to be executed again. The solution we propose to be able to support specific communication devices and their native communication library with fault tolerancy is a clustering approach. In this model each cluster is composed of multiple nodes that may use specific communication facilities and each cluster may communicate through hubs with the other clusters. An interesting point is that a cluster may use multiple communication devices to communicate. Thus, if a communication failure occurs with a network or communication interface a node may still be able to use another gateway to communicate. We also partially resolve the problem of the failure of processes by the fact that if a process fail it will in the worst cases only crash an entire cluster but not the whole application. The failing cluster could then be started again and reintegrate the application dynamically. This assumption however is possible only if a cluster could use multiple interface of communication at the same time and if one of them support failure of processes. To resolve this problem we suppose that the IP protocol is usually available and use it as a standard communication medium for control messages between clusters.

The figure 1 presents the proposed architecture model. The described configuration is composed of three clusters with their respective processes that are interconnected by an IP network for inter-communication and use different internal cluster communication network for intra-communication.

To provide a standard interface to the programmer and hide the underlying communication devices a standard API has been designed (abstract layer) and a multiplexer/demultiplexer is used

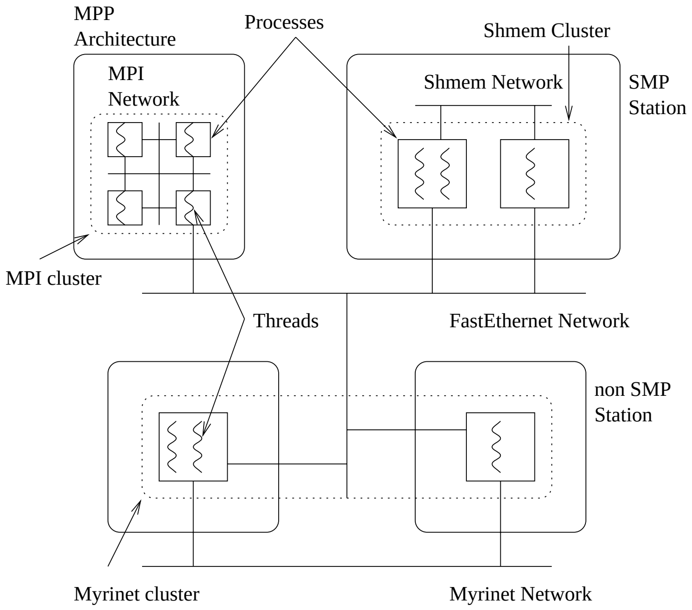

Figure 1: The cluster approach of DECK.

to received and send message according to their destination using the correct interface. The implementation and related problems are described in more details in the implementation of the DECK prototype.

3 A multithreaded architecture approach

SMP architectures and clusters of SMP are nowadays a common distributed platform for distributed applications. Even parallel supercomputers, like the SP3, integrates multiprocessors nodes in their last configurations. These architectures could be efficiently used if multiple activities of an application share the processors available. To do so, a process approach could be retained, however, the latency and the complexity involved in such approach for the local interactions between the processes are usually high. To resolve this problem, a second solution is to design multithreaded processes where different threads are included in a unique process that execute them concurrently over the physical processors and use the shared memory offered by the process to express their interaction (communications of data or local synchronization) as illustrated on figure 2.

Another motivation comes from the characteristics of some distributed applications and parallel or distributed languages that require efficient multithreading support in order to achieve good performance. This is the case for irregular applications [8] and distributed languages like DPC++ [4]. Parallel irregular applications for example are applications where the amount of activities depend from dynamic computations of the input data. Thus, the number of threads that may be generated could not be precalculated and their number could be very high. To encapsulate each thread of execution in a process would lead to poor performances due to their number and the fact that each time a thread of execution will have to be created a process have to be instantiated, also the life cycle of the process could be very short (the computation time could in some cases be lower than the time required to create a process).

This is why, current distributed executive kernels usually offer two level of concurrency. The first level is defined by the concurrent execution of processes on different machines while the second correspond to the creation and execution of multiple concurrent threads of control inside a same process. In this model, the first level of concurrency offers a coarse to medium grain of parallelism and the second a fine grain parallelism.

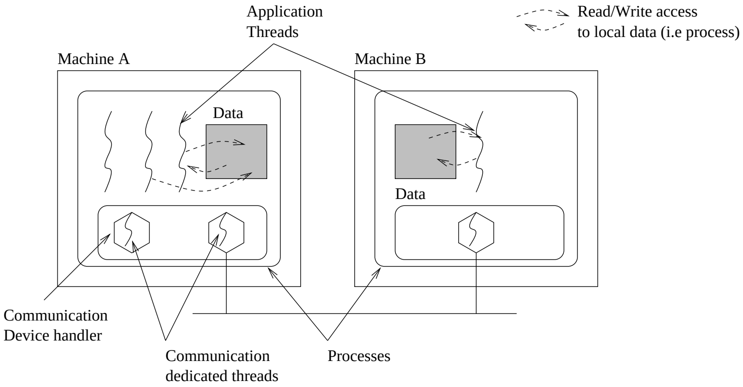

Figure 2: The two level of concurrency.

Multithreading capacities are also motivated in many cases by the possibilities to increase performance and the expression of the parallelism (it is sometimes much more easier to express different thread of control than build a complex automata).

The performance benefits that result from the integration of communication and multithreading come generally from the possibility to overlap communications by computations and the low latency associated with the creation of remote threads. The consumption of memory resources are lower as the latency of intra- process communications made by the threads. The complexity of these local communications are also greatly reduced as no conversions are needed (specifically pointer conversions).

However, many problems are involved in such integration. The first problem to resolve is to protect the communication libraries from the side effect involved in the execution of multiple threads that could call the communication libraries at the time. The second problem is to integrate efficiently the multithreading kernel with the communication kernel to provide a thread aware multithreaded communication environment that could handle directly the threads that are calling the communication library. For example, if a thread is suspended on a completion of communication the communication kernel could re-schedule directly the thread upon reception of the awaited message. Also, as the communication kernel is aware of the presence of the threads so it could support communications from a thread to another one and not only inter-process communications.

The last problem to resolve also known as the pooling problem [12] appears as the communication kernel could not be called for a while if no thread are calling it. However, communication messages may be received and processed. To avoid such situation it is important to ensure that communication kernel is called periodically or to trigger an interruption when a packet or message is received. Another solution is to associate a thread dedicated to the pooling task in each device modules (cf. figure 2).

4 The programming model of DECK

We have exposed in the previous section the two level of concurrency that DECK offers and also the cluster model adopted to use multiple communication devices and fault tolerance. In this section we present the programming model introduced by DECK to support efficiently this architecture at both level and manage the clusters of nodes.

4.1 Processes and clusters management

The processes and cluster of processes (or nodes) are the primary entities that DECK have to support. For this, it offers a variety of functionalities to create, destroy and permit basic interactions between processes and clusters. Indeed, a DECK application is constituted of processes that are distributed among a collection of nodes and attached to a cluster that represents a specific context of communication generally coupled to a specific hardware architecture (i.e MPP architecture) or ad-hoc communication libraries like (i.e. MSA Myrinet) as shown on the figure 1.

A cluster of communication could be compared as an open group in the sense that a process could dynamically integrate it even if it is not previously known by the cluster. The management of every cluster is centralized in dedicated process. This process is responsible to maintain the coherence of the group and process the request of processes that want to leave on enter the cluster.

In the case of MPI cluster for example, the *root* process is responsible for the creation of the node that made the cluster. If a process fail the entire cluster has to terminate. To restart the cluster, only the *root process* has to be restarted. In the case of a fault-tolerant cluster, if a process fail it could be restarted by a dedicated process (the cluster manager) or by third parties (manually by the user for example) however it is its task to recontact and integrate the cluster again. To accomplish this, any process could always contact a naming service to retrieve the address of a particular process or a cluster manager.

Every processes (or node) are given an identification address during their startup by the naming service. This address is unique and global and the naming service do not forge twice a same address. Thus, if a process crash it could be restarted and reuse its previous address if it had saved it.

4.2 A fine grain parallelism programming model

If a DECK application is primary a collection of distributed process, the computation and communication are done by thread of execution. This is why each time a DECK process is created a main thread of execution is also created. This thread could then create new thread locally or remotely and interact with the others processes of the application. This model is the model usually adopted by many other executive distributed kernel[13, 10, 11, 3]. However, they offer different programming model to express the thread activities and the interactions that are allowed between them.

The problems involved with distributed fine grain parallelism are coming from the fact that the life cycle of the threads could be very short and that they could be dynamically instantiated randomly in different nodes. How to instantiate a distant thread and how to interact with it are the first problems to be solved. If we adopt only a message passing programming paradigm (if we suppose that it is possible to create threads on every nodes) then the programmer have to keep track of each threads and its execution site and have to ensure that a remote thread is present on a distant node before accomplishing any communication with it. This complexity could lead to difficulties in the design of fine grain distributed application.

This is why these kernels offer usually functionalities to create remotely thread of control using a paradigm close to the RPC model or RSR model [11] if a remote thread is not allowed to return directly results to its creator (it usually has to use the same mechanism to do so). These mechanisms are also coupled with a message passing paradigm and in some case with other programming model like shared memory (TPVM [7] or Cid [14]).

The DECK programming model offer a remote service creation as a basic mechanism to execute remotely a thread. Each time a thread is created via such mechanism it inherits the address of its creator and could then communicate with it using a message passing operation or by executing a RSR in the process referenced by the address. A service in the case of a RSR, is a C function that has been declared to the DECK kernel as an entry point. A variant of this call is also proposed to execute a remote service as a regular function call and to avoid the overhead of a thread creation (in this last case the function calls are executed by the DECK kernel in sequential).

The DECK kernel offers also some functionalities to create locally a thread in a node and express local synchronization mechanisms (mutex) to realize mutual exclusion and resolve problem such as race condition [16]

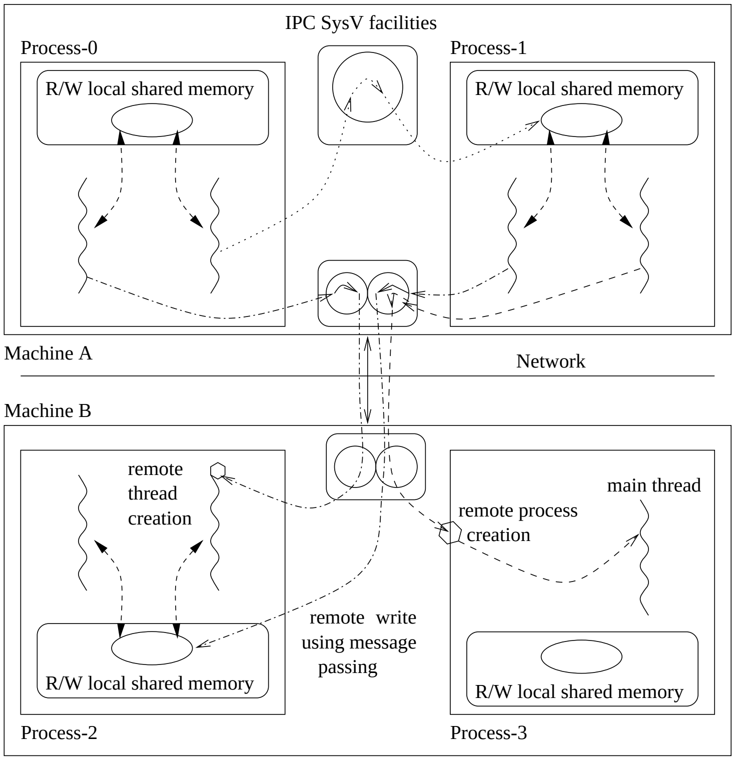

Figure 3: Interactions in DECK.

The figure 3 presents the different interactions that are allowed by the DECK model. Each thread of control could create thread locally or on a remote node. The exchange of data between two threads are done through point to point message passing operations. Each message are emitted to destination of a port (or a mail box). A port as to be declared by the user in order to accept incoming message. A port is reference by a *portid* that is only local to a node. A context of communication has to be created (like communicator in MPI or context in the last release of PVM) in order to dispatch the communications in different communication spaces that could be handled separately (a context per cluster for example). Thus, a valid address to issue a communication is constituted by the triplet (*node-id*, *port-id*, *context-id*) as it is global and unique. Each communication are asynchrone but associated to a request structure (like MPI) thus a thread could test if a message has been sent and if it can reuse the buffer allocated for a communication (send) and test or wait if a message has been received using the same kind of request structure.

The messages that could be sent and received are limited to the basic C data types and the communications are issued only between two processes. However, future extensions are planed to integrate as extension services support for complex data (list, complex structure, tree) and collective communications.

4.3 Extension of the programming model of DECK

The programming model offered by DECK itself is dedicated to support the two basic entities of DECK that are the distributed processes and the threads. The functionalities provided are limited to the life cycle of these execution units (creation and destruction) and the management of the interactions involved between them (message passing, synchronizations,...).

More complex operations involved between threads or processes like group communications, complex data management or dynamic load balancing of the thread are not offered directly by DECK itself. However, the philosophy of the DECK environment is to build an open environment where extension could be provided by means of services that lye on top of the basic functionalities enumerated in this section. Indeed, as the services are not included directly in the low level kernel of DECK it increases the modularity of the environment and contribute to parallel developments and portability.

5 The DECK services layer

Some basic services that may be required by some applications are tightly coupled to the DECK kernel and thus it may be difficult to a simple user to implement such services as they may have to interact with many components of the DECK architecture. To avoid this problem and extend the functionalities of the DECK kernel we propose a service layer that lie on top of the basic mechanisms offered by DECK. The kind of services provided may be dedicated to answer basic requirements of an application (group communication, complex data management support, etc.) or extend the functionalities of existing services (secure communications, fault tolerant group communications, etc.).

The service layer should indeed contains only the required functionalities needed by an application and also answer dynamically to specific needs. To provide such requirements, the DECK kernel proposes two kind of services. **Local services** are modules that may be built separately from the DECK kernel and have to be linked with the application during the compilation phase (if they are needed). The second kind of services or **Global services** are services that are independent from the application by the fact that they are provided by specific processes. This is useful if an application want to dynamically access specific services that are provided by third parties components and also to build fault tolerant services. This could be the case for a naming service facility that want to be implemented with two independent processes to ensure a backup if one the naming server crash.

To illustrate this architecture and present two of the basic services actually offered by DECK, we described in the following subsections the naming and group services.

5.1 The naming service

The naming service is a basic service module that has been implemented over the DECK kernel in order to resolve the main problem encountered by our model when dealing with multiple communication interface (or network) or involved with the threads support.

The main role of the naming service of DECK is to abstract the hardware or specific communication libraries addressing scheme in order to provide a uniform addressing model. This kind of services are essential in order to support the multiplexing and demultiplexing of packets or messages that could be received or send from different communication devices with specific addressing mechanisms. This kind of services also provide routing servicing if a node have to re-route a message if it plays a hub or gateway role in a cluster for example.

To achieve such requirements we have adopted a three level addressing architecture that abstract completely the physical addresses used by a communication device. This architecture is presented in detail in the subsection *The case study of the naming service* and may be compared to the approach use by [21]. Basically this architecture model permits to a process to reference another process using a high level address reference that could be a name (i.e. string) or a process id (i.e. a simple integer). This address could be translated into an abstract physical address which reference a physical address. As the logical address (or high level address) does not reference directly the physical address it is possible to change the execution node of a process (migration) or in a case of a crash to re-create it on a new node and still use the same logical address. This is also useful in some fault tolerant protocol when a backup process has to replace a primary process. The coherency of the data base that contains the different addresses and their relations is ensured by the naming service.

The last functionality offered by the naming service is the possibility to manipulate complex address that are not only related to a process or a node but also to a thread or a communication port. This kind of support has been also proposed via extensions by some communication libraries in order to support more efficiently multiple threads per process [22, 17]

5.2 The group service

The group services facility is the second module service that has been defined for DECK. It has been designed to offer the basic services to build collective communications. Thus, it does not, currently, offer collective communication functionalities but rather aims to provide basic mechanisms to build and manage group of processes and communication context. Theses structures could then be used to build collective communications like barrier, broadcast or reduction. To provide such kind of communication is also one of the main goal of DECK as there are few communication libraries or distributed executive kernels that offer such mechanisms at the thread level [3].

For example, Chant [10] proposes like DECK an extension (that could be viewed as a service module) that offer very basic group support at the thread level. The interactions are limited to the group members and only the barrier function is provided as a collective communication. TPVM [7] as also made an attempt to include thread support to the PVM communication libraries but like Chant the operations that could be achieved over a group are limited. Some distributed executive kernel as also proposed some functional extensions to the classic RPC and thread duplication in order to simulate group of thread and global synchronizations [13, 3].

In the case of DECK, the policy is to distinguish the group facilities from the operation that could be achieved over it. Indeed, group management (open or closed group) and structure properties (hierarchical, equality group,...) could be different and depends from the application needs. Also, protocol over these groups may differ and follow specific semantic [20]. This why, we have adopted for DECK a service approach where the basic group management functionalities are offered (and soon basic collective communications) instead of include directly theses services in the DECK kernel.

From the performance point of view, this should not be a real problem as most of the communication libraries use point-2-point communication library in order to build collective communication [5] and specifically if they have support fault tolerance as properties such as atomic broadcast are usually not offered by regular communication devices.

The services offered by the DECK environment will be extended in the future. And services such as complex data management, transaction or timing will also be added. These implementations will follow the same goal towards portability, hardware abstraction, interoperability and scalability.

6 The DECK prototype implementation

The DECK prototype is still in an early stage of development. Currently, the low level abstract device layer has been defined and implemented. Basic device modules such as a UDP device and an IPC SYS/V device are available. Other devices, like PVM and MPI devices will be implemented.

The DECK API offers the basic remote invocation mechanisms and message passing but the implementation of contexts and clusters functionalities are still under development. Support for heterogeneity is provided at the communication layer using a home made *a la* XDR data encoding and decoding facilities. However, further tests for 64 bits architectures and more exotic architectures has to be done and XDR services should also be included in the next releases.

The integration of the multithreading kernel has been made but fine tuning and implementation of different pooling policies have to be done in order to achieve good performance specifically for the case of SMP architecture. It should be transparent from the user point of view and be able to adapt itself dynamically upon the network state.

Support for fault tolerance is also in early stage of development. The UDP device has been built using a modified ACK/NACK protocol to handle unreliable communications and server crash. The protocol has to be tuned for specific architectures and systems dynamically. It is possible, in fact, to change the timing value of the communication protocol algorithm and interact with the buffering management policy at runtime. This part will be studied in the future to provide a dynamic and reconfigurable communication protocol at least for IP networks.

6.1 The Deck internal architecture

The internal modular architecture of DECK, presented on the figure 4, is based on an abstract communication device (**DEVICE_HANDLER** module) that may encapsulate several physical communication devices while offering a single (and simple API) to the upper layers of DECK. The abstract communication device could be compared to the Chameleon device of MPIch [5] but in the case of DECK multiple communication devices could be integrated at the same time in a same abstract device.

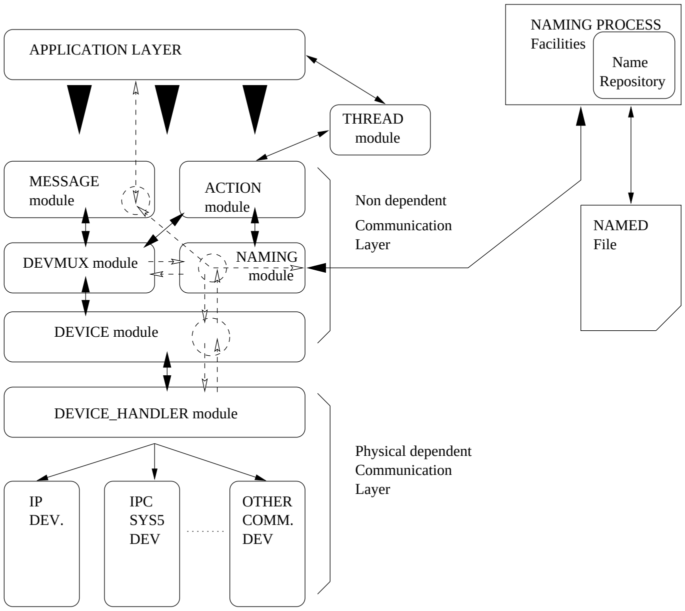

Figure 4: Internal view of DECK.

An important point to outline, is that in comparison of architecture similar to MPIch Chameleon or most communication libraries it is no more possible to rely on a uniform addressing mechanism (i.e. an integer number to reference every process). Also, general communication addressing schemes such as IP is also not usable because an IP address is not a standard addressing ID for communication library such as MPI or PVM (at least from the process point of view). To handle this problem we have built a three level addressing architecture that may be compared to the Globe resolution system [21].

The first level correspond to the *low physical address* manipulated by a communication device module (i.e. a *global pointer* for shared memory libraries, an *IP:PORT* address in the case of IP networks or a *taskid* for message passing communication libraries like PVM or MPI).

The second level is a *global physical address* that is manipulated only by the upper layers of DECK. It is useful to have two levels of physical addressing as a process referenced by a global physical address may crash and have to change its low physical address so if it is started on an another node it still could use the same global physical address.

The last level is a *logical process address* that is used to reference a destination process at the end-user level (i.e. application). The send and receive message passing functions accept such addresses. It could be compared to a task-id or process-id used by regular communication libraries and is implemented as integer number. In the next subsection we present the naming service that is responsible for the coherency handling and the management of these addressing architecture.

The abstract communication device layer is also referenced as *the physical communication layer* as it could be specific to an architecture or a system. However, the upper layer or *independent communication layer* is portable over any abstract communication device layer and this without any modifications.

The upper layers of DECK are composed of kernel modules that are responsible, in a nutshell, for the management of higher level communication functionalities like message passing, remote service requests and the management of the threads. Basically, it is composed of five modules that interacts to process incoming (or sending) messages from (to) the abstract communication device (as shown on the figure 4 according to the dotted lines).

- The DEVMUX module is reponsible for the dispatching of the different packets or messages that are received by the physical communication layer. This include also specific message such as *CONTROL* messages or *ACTION* messages. It is in fact the core communication multiplexer and demultiplexer of the DECK kernel.

- The NAMING module handles address resolutions that may occurred when a message has to be sent or is received.

- The MESSAGE module takes care of the message passing operations like the management of the communication contexts or the buffers associated to the messages.

- The ACTION module manages the handlers that could be triggered remotely to instantiate a thread of execution or a local function.

- The THREAD module offers the basic functionalities required to create, destroy or synchronize the threads. It could be called by any other module in order to create a new thread of control or acquire a lock over a shared variable of the DECK kernel.

If any physical device modules have to interact with the upper layers they have to call the **DEVICE_HANDLER** module that will then call the **DEVICE** module which finally will dispatch the call to the corresponding upper layer module. This process enhance the portability and the reusability of both layers. Indeed, the DEVICE and DEVICE_HANDLER modules offer an opaque interface to each others. To write a communication module for example, the programmer has simply to respect the interface of the DEVICE module without knowing the implementation details of the upper layer.

Another important issue that have to be considered in the implementation of communication systems is the policy of buffering management they offer. Indeed, new MSA architectures such as [15] try to reduce to the minimum the number of copies needed for the transmission of a message. The ultimate goal is usually to offer a zero copy architecture. However, in the case of DECK, such implementation may not be feasible as it is not possible to change the implementation of some underlying communication modules that it may have to support. This is the case for most IP communication implementation (TCP/UDP) and many communication libraries (MPI,PVM,P4). To be able to support MSAs that offer zero copy mechanism, DECK's communication buffers are always referenced through pointers from the upper layers to the device communication modules. In fact, it is up to the device communication modules to allocate the buffers when the data are received. The allocated buffer is then exported to the upper layers via a global pointer. The same mechanism is adopted by the *MESSAGE* module when a buffer corresponding to a message has to be sent.

6.2 The case study of the naming service

The service layer of the DECK environment consists of a set of independent and dependent modules that are implemented on top of the DECK kernel. Thus, they can not access directly the DECK modules and they have to use the standard API offered by the DECK kernel as a regular application has to do. Independent services are simple services that call only the functionalities offered by the DECK kernel (i.e. a complex data manager). A dependent service is a service that depend of the functionalities offered by other services to work properly. This is the case for example of most services that have to access the naming services. These last kind of services could also be local or global. They are considered as local if they produce no side-effects over distant processes and global if they interact with other processes or are implemented as a distributed services. Hybrid implementations could also be implement using a global service, that is distributed, and local services to process local requests given by the global services.

As an example, of such hybrid distributed services, we present in the next paragraphs the naming service.

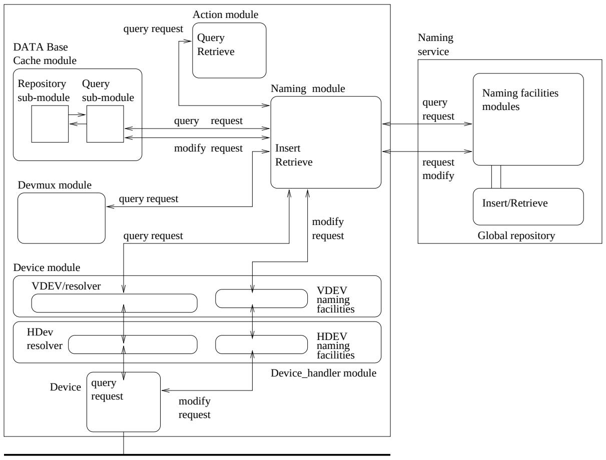

Figure 5: The naming service implementation.

The figure 5 presents the architecture of the naming service. The implementation of this service illustrates the modular architecture of DECK and the possibility to build and use dynamically services on demand. The name service described correspond to the naming service presented in the *DECK services layer* section of this article. The main goal of this service is to provide an abstract and fault tolerant naming service system that will be used by the DECK kernel and also by the application layer.

It is responsible, in a nut shell, for the management of a global name repository that is used to translate the different addresses manipulated by DECK (*low physical address), global physical address, logical process address*. As it has been pointed out, all the interactions from a device module to upper layers have to transit through the device and device handler module each time it wants to call services from upper layers.

The interactions of a device module with the naming module appear for example generally in two cases. The first case corresponds to the creation of a process. The device module reports to the naming module the address of the corresponding process (*low physical address*. The naming module records the address and allocate a *global physical address* for the new process if it is its first creation else it links the former *global physical address* to the new *low physical address*. The naming service stores all the entries of the naming module and could export (or import) to (from) other processes it's local data base (their local data base to update its global repository). The second case appears each time a message is received or has to be sent. The different module of the upper layers and also the involved communication device modules have to translate different kind of addresses (cf. figure 5).

The fact that a service module is independent from the DECK service permits that it could be changed and this dynamically. In the case of the naming services if a fault tolerant naming service is required, this could be achieved simply by making a re-compilation of application with the fault tolerant naming service library. Another policy that could be applied is to use *global services* that are independent from the applications. In the case of the naming service, a dedicated process could ensure the administration and the coherency of the different addresses manipulated by DECK (cf. figure 4). Also, multiple global naming services could be created to provide more support for fault tolerance. In the current DECK implementation a primary global naming service may be extended by a backup global naming service that will replace the former in a case of a crash.

6.3 DECK in the DPC++ environment

Currently, DECK is being integrating in the DPC++ distributed oriented object programming environment [4]. Some functionalities of DPC++ required more extensions from the DECK layer such as the management of complex data and for the services that implement fault-tolerant protocols for group of processes. However, a primary implementation is under development and the DPC++ environment currently used the multithreading facilities offered by DECK to encapsulate the execution of multiple active object in a single process and the internal concurrency of the active objects (parallel execution of multiple methods of an object). This implementation also benefits from the fact that DECK offers support for thread to thread communication and thus every distributed objects could be associated to a port of communication.

7 Conclusion

The first goal of the DECK distributed executive kernel was to design a portable and efficient environment to support efficiently new hardware architectures such as multiprocessor SMP architectures and high-performance communication networks and their specific MSA. The second objective of DECK was to provide a minimum support for fault tolerance and this without relying on a specific implementation or design dedicated exclusively to fault tolerance.

These primaries objectives have been successfully integrated in our architecture model for the DECK environment. We have proposed an open executive environment where multiple heterogeneous communication devices could be used in conjunction with multithreading facilities in order to efficiently support the characteristics of these new hardware architectures.

The programming model offered provides support for both coarse and fine grain parallelism and multiple paradigm of programmation in order to provide a convivial programming environment to express the parallelism and the involved communications or synchronizations required by distributed applications.

The current implementation of DECK is still in an early stage of development. The first step was to build the core components of the DECK architecture and validate the modular approach. This have been made with the implementation of a primary prototype that support basic communication devices such as UDP or IPC SysV shared memory facilities with a naming and group services.

The second phase of the implementation and validation of DECK will be to integrate high-performance MSA for local area network such as Myrinet, support more efficiently SMP architectures and to offer more complex services in order to match the requirements of the distributed applications we want to support and the constraints and functionalities needed by some higher-level specific services.

References

[1] F. Armand, F. Herrman, J. Kipkis, and M. Rozier, *Multi-threaded processes in chorus-mix*, in in 'Proc. EEUG Spring' 90 Conf', 1990, pp. 1–13.

[2] M. Bernaschi and G. Richelli, *Development and results of pvme on the ibm 9076 sp1*, Journal of Parallel and distributed Computing, 29 (1995), pp. 75–83.

[3] J. Briat, M. Christaller, and M. Rivière, *Athapascan0: Concepts structurants simples pour une programmation parallèle efficace*, Calculateurs Parallèles, 7(2) (1995), pp. 173–196.

[4] G. G. H. Cavalheiro, R. C. Krug, S. J. Rigo, and P. O. A. Navaux, *DPC++: an object-oriented distributed language*, in Conferencia Internacional de la Sociedad Chilena de Ciencia de la Computación, 1995, pp. 92–103.

[5] N. Doss, W. Gropp., E. Lusk, and A. Skjellum, *Using MPI, Portable Parallel Programming with the Message Passing Interface*, MIT Press, 1993.

[6] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. William, *Beyond multiprocessing - multithreading the sunos kernel*, in in 'Proc. of Usenix Conference', SunSoft Inc., 1992, pp. 11–18.

[7] A. Ferrari and V. Sunderam, *Tpvm: Distributed concurrent computing with lightweight processes*, in IEEE High Performance Distributed Computing 4, 1995, pp. 211–218.

[8] T. Gautier, J. Roch, and G. Villar, *Regular versus irregular problems and algorithms*, in In proc of IRREGULAR 95, 1995.

[9] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam, *PVM: Parallel Virtual Machine*, MIT press, 1994.

[10] M. Haines, D. Cronk, and P. Mehrota, *Chant: lightweight threads in a distributed memory environment*, in Journal of Parallel and Distributed Programming, 1996.

[11] F. I., K. C., and T. S., *The nexus approach to integrating multithreading and communications*, in Parallel Programming Env. for High Performance Computing, 2nd European School of Computer Science, ed., ESPPE96, 1996, pp. 53–67.

[12] K. Langendoen, J. Romein, R. Bhoedjang, and H. Bal, *Integrating pollling, interrupts, and thread management*, in Proc. of the 6th Symposium on the Frontiers of Massively Parallel Computation (Frontiers '96), 1996.

[13] R. Namyst and J. Méhault, *Pm2 : Parallel multithreaded machine. a computing environment on top of pvm*, in In 2nd Euro PVM UG Meeting, 1995, pp. 179–184.

[14] R. Nikhil, *Parallel symbolic computing in cid*, Springer-Verlag LNCS., (1995).

[15] S. Pakin, M. Lauria, and A. Chien, *High performance messaging on workstations with fast message for myrinet*, in In Proceedings of SuperComputing '95, 1996.

[16] A. G. R., *Concurrent Programming: Principles and Practice*, Benjamin/Cummings, Redwood City, CA, 1991.

[17] A. Skjellum, N. Doss, and K. Vishwanathan, *Inter-communicator extensions to mpi in the mpix (mpi extension) library*, tech. rep., Missipi State University, 1994.

[18] N. Suzuki, *Shared Memory Multiprocessing*, MIT Press, 1992.

[19] A. Tubtiang, *Un commutateur ATM pour le réseau numérique à intégration de service large bande*, PhD thesis, Univerisité de Paris 6, 1993.

[20] R. van Renesse, P. Kenneth, K. Birman, and S. Maffeis, *Horus, a flexible group communication system*, in Communications of the ACM, 1996.

[21] M. van Steen, F. Hauck, and A. Tanenbaum, *A model for worldwide tracking of disttibuted objects*, tech. rep., Vrije Universiteit, 1997.

[22] H. Zhou and A. Geist, *Lpvm, a step towards multithreaded pvm*, Concurrency - Practise and Experience, (1997).