O.M. Nierstrasz (ed.): Proc. 7th European Conference on Object-Oriented Programming ECOOP'93, LNCS vol. 707, 1993, pp. 361-383

PANDA — Supporting Distributed Programming in C++

H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, R. Schwarz

Department of Computer Science, University of Kaiserslautern
P.O. Box 3049, W - 6750 Kaiserslautern
Germany

Abstract: PANDA is a run-time package based on a very small operating system kernel which supports distributed applications written in C++. It provides powerful abstractions such as very efficient user-level threads, a uniform global address space, object and thread mobility, garbage collection, and persistent objects. The paper discusses the design rationales underlying the PANDA system. The fundamental features of PANDA are surveyed, and their implementation in the current prototype environment is outlined.

1. Introduction

Systems for parallel and distributed object-oriented programming can be classified into two basic categories. Firstly, there is a variety of programming languages developed especially to serve experimental purposes. Different object models for parallel and distributed programming can be investigated by designing and working with such systems. Some examples of languages in this area are Emerald [Jul el al. 88], Pool [America and van der Linden 90], Sloop [Lucco 87], and Orca [Bal et al. 92]. In contrast, systems belonging to the second category provide support for parallelism and distribution in the environment of an existing object-oriented programming language. In this type of systems, the corresponding mechanisms must fit into the framework of the underlying language, while in the first case no restrictions exist for system development. An advantage of a system embedded into an existing language environment is that the programming tools of the environment and existing software components can still be used. Furthermore, with the increasing acceptance of C++ [Stroustrup 86] — especially in the area of system programming — there is a growing need for systems providing support for programming of parallel and distributed applications in this language. Representative systems in this research area are, among others, Amber [Chase et al. 89], Cool/C++ [Habert et al. 90], and DC++ implemented on top of Clouds [Dasgupta et al. 91].

In C++, programming support for parallel and distributed applications can be realized by a library of classes providing the required functionality, and a run-time system to execute these functions. Important qualities of such systems which do not rely on compiler extension are their portability, extensibility, and ease of modification, as has been shown, for instance, by Presto [Bershad et al. 88]. The behavior of every system object can be redefined, allowing to adapt it to changing requirements. On the other hand, however, several mechanisms proposed to handle parallelism and concurrency control in the object-oriented setting can be implemented only by introducing special language constructs [Gehani and Roome 88, Nierstrasz and Papathomas 90, Saleh and Gautron 91]; thus, they are not applicable in this case.

Furthermore, providing distribution support in C++ is — to a large extend — a problem because of its “C-isms”. Restricting the language would facilitate this work but it would conflict with the user requirements concerning software re-use. Full language support ensures that software developed for centralized systems can be used in a distributed environment with an acceptable amount of adaptation.

The goal of PANDA is to provide an environment for parallel and distributed programming in C++, imposing as little as possible restrictions on the use of the language. This work is based on the experience which we obtained in previous projects where experimental distributed programming languages and environments were developed [Wybranietz and Buhler 89, Buhler 90]. PANDA’s basic library consists of classes which address the issues of parallelism, distribution, object persistence, and garbage collection. These classes can be used directly for building an application, or — alternatively — as a basis for more specialized programming systems, since the user is free to define new abstractions by customizing the PANDA classes. A fundamental part of PANDA is a thread package which provides mechanisms for expressing and handling parallelism in the user space. Here, the crucial issue is performance. The cost of parallelism determines its grain and has thus a strong influence on the programming style. As has been shown by FastThreads [Anderson et al. 89] and Presto [Faust and Levy 90], high performance and flexibility of parallelism support can be achieved by a *user-level* thread implementation.

PANDA has been designed for a hardware platform consisting of a network of homogenous processors. It addresses the issue of distribution by the concepts of single global address space, distributed shared memory, and thread migration. Similar to Amber [Chase et al. 89], all nodes of a PANDA system share a single virtual address space managed globally. Thus, a virtual address can be used as a system-wide unique identifier. References transmitted across node boundaries do not lose their meaning. This concept has serious limitations in the case of 32-bit addressing, which makes virtual addresses to a scarce resource. The wider virtual addresses of emerging 64-bit architectures, however, will support such a system organization, as stated in [Chase et al. 92, Garrett et al. 92].

This paper surveys the PANDA system. Section 2 describes the PANDA interface from the application programmer’s point of view, and it discusses the underlying design rationales. The implementation of the fundamental features of our programming environment is briefly outlined. In particular, the thread package, distribution and mobility concepts, as well as the integration of persistence and garbage collection are presented. Section 3 describes the system layers of the PANDA run-time environment. In Section 4, the current status of our prototype implementation is presented. Finally, the main results are summarized, and the direction of future work is indicated.

2. Programming in PANDA

The PANDA system architecture is based on a C++ class hierarchy, and it provides an object-oriented application interface. The system levels of PANDA are especially designed to get high performance at the application level. Our design has been influenced by new trends of hardware architectures and operating systems.

The system supports standard C++; no extensions to the original language are required. This decision was influenced by our goal to be highly portable. However, we use C++ precompiling to simplify the user’s task of encoding. This precompiler inserts source code fragments at specified places. This is, for example, useful when a class should be protected against concurrent activation. Without a precompiler the programmer would have to insert a mutual-exclusion semaphore in every method. Instead of this, only a hint in the class definition is necessary, and the precompiler can do this instrumentation. The precompiler frees the programmer from the burden of such routine work, but it is not essential for the concepts that are described in the sequel.

2.1 Thread Package

Activities in a system are called processes or *threads*. In a usual C++ program there exists only one thread. Figure 1 depicts what happens during program execution. A single thread “runs” sequentially through objects and their methods.

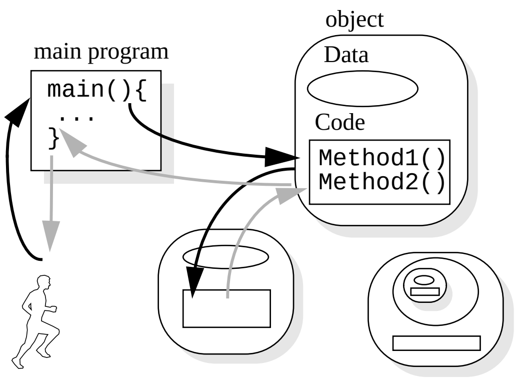

Fig. 1. ”Running” thread in a sequential C++ program

To create a parallel program where several threads operate in one application, the programming model has to offer certain functionality to handle concurrency. In PANDA, the basic mechanisms for thread management are provided by the system class `UserThread`. An instantiation of this class generates a new activity. The advantage of this approach is that concurrency support can benefit from the object-oriented environment. By deriving new classes from `UserThread`, various types of active objects may be defined. An example illustrates the usage of PANDA threads:

The following declaration specifies a thread which periodically increments its private variable `number`:

```
class Increaser : UserThread {
  public:
    Increaser (int StartValue) { number = StartValue;}

    void code() {
      for (;;) {
        Delay(1sec); //method of UserThread
        number = number + 1;
      }
    }
  private:
    int number;
};
```

Creation of threads is now performed by:

```
main{
  incPtr1 = new Increaser (10);
  incPtr2 = new Increaser (500);
  incPtr3 = new Increaser (1000);
}
```

Just before the closing bracket three parallel threads exist in the system, each represented by an `Increaser` object (Figure 2).

Note that the `Increaser` class has a special method `code`. On instantiation, the invocation of `Increaser`'s constructor creates a new thread which executes the `code` method. The constructor is executed in the creator's context and it is guaranteed that this constructor has finished before the new thread is actually activated, so that parameters may be passed safely.

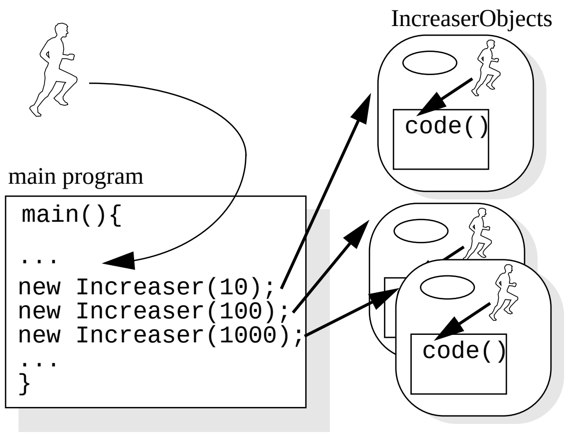

Fig. 2. Active objects obtained by derivation from *UserThread*

PANDA threads are realized by a thread package implemented in user space [Assenmacher 92]. The underlying operating system kernel has no notion of PANDA threads. Management and scheduling of threads is done in non-privileged user mode. Threads which are managed in this way are called *user-level threads* [Bershad et al. 88, Marsh et al. 91]. Our implementation offers very fast thread management. Creation and deletion of threads is extremely fast, causing an overhead within only one order of magnitude compared to a procedure call, see Table 1 in Section 4.

In a system with concurrent activities which act on non-disjunctive object sets, mechanisms for synchronization are required. PANDA offers synchronization objects such as semaphores, distributed read/write semaphores, and signals. To protect objects against concurrent access, a class may be turned into a monitor. For this purpose, the PANDA precompiler may be instructed to add mutual-exclusion semaphores to every method, thus saving a lot of encoding work.

2.2 Distribution

Recall that one of our main goals was to support programming in a distributed environment. Consequently, the question arises how PANDA manages to link the remote components of a distributed application together. In particular, what provisions are taken to ensure that a thread can visit and enter not only local, but also remote objects? This question is central to our model. In this chapter, we address this issue and discuss the different ways to solve the problem.

Three fundamental mechanisms can be employed to provide activities which are able to span several nodes:

As a first alternative (Figure 3), we may simply rely on message passing to cross node boundaries if an invocation is made to a remote object. In essence this type of remote object invocation corresponds to the well-known *remote procedure call (RPC)* mechanism which has been discussed in great detail in the literature [Birrell and Nelson 84]. In order to provide a transparent interface, each caller and

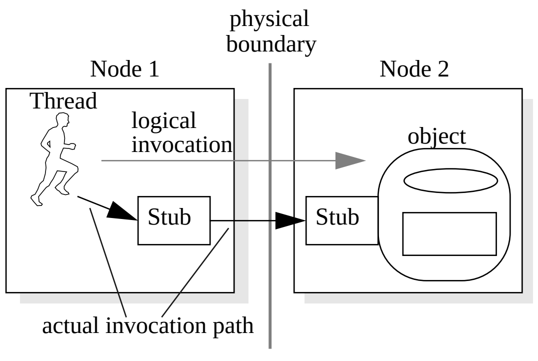

Fig. 3. Supporting remote object invocation

each remotely accessible object has to be equipped with an appropriate RPC stub, and parameter marshalling has to be done on every invocation of a non-local method. In the object-oriented setting, the functionality of RPC stubs may be provided by so-called *proxies* [Shapiro 86], i.e., local representatives of remote objects. Typically, marshalling is based on some preprocessing of the source code. However, providing perfect transparency in a language like C++ is rather difficult. For instance, it is difficult to handle pointers which may occur in the parameter list of a method. Also, it should be noted that RPC supports a slightly asymmetric model of computation. Most applications based on RPC make an explicit distinction between *clients* on the one hand, and *servers* on the other — at least at the conceptual level. In some applications, such an asymmetry appears to be artificial, and is only an artifact of the underlying RPC mechanism. These subtleties diminish the location transparency which can be achieved with RPC.

Therefore, a more natural approach to the distribution problem is to create the illusion of a single, global object and thread space, i.e., to hide the node boundaries from the application programmer. To this end, the system has to ensure that all operations — local or remote — are eventually carried out locally. On the conceptual level, this can be achieved with two different strategies [Jul et al. 88, Chase et al. 89, Dasgupta et al. 91], one based on object mobility, the other on thread mobility. In Figure 4, the system migrates the object to its invoking thread as soon as a non-local invocation is encountered. After the object has been moved to the node on which the thread resides, the remote object invocation turns into a local call. Figure 5 shows the dual solution. Instead of moving the object, the system migrates the invoking thread, again turning the remote invocation into a local one.

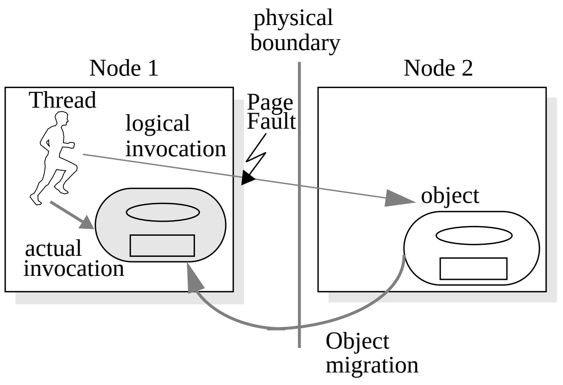

Fig. 4. Object migration based on page-fault mechanism

The solutions depicted in Figures 4 and 5 are very attractive from a programmer's point of view. They do not distinguish between the caller (client) and the called (server), and they need not be reflected by the program code. In principle, the program could be completely unaware of the distributed nature of the underlying hardware. This is a highly desirable property. For example, an application could be im-

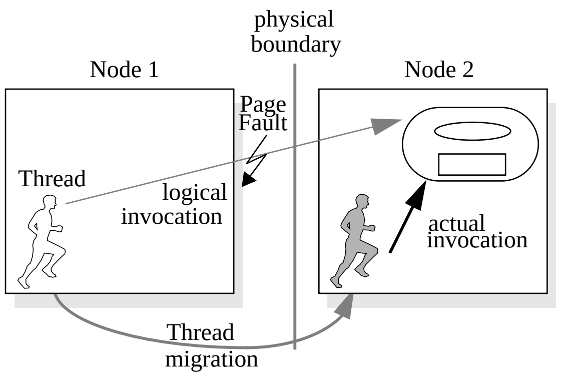

Fig. 5. Thread migration based on page-fault mechanism

plemented and tested in a centralized system; distribution may then be realized by simply spreading the different components of the application to different nodes of the distributed system, without any need to change the code of the various program modules. In most application domains, this view is, of course, too optimistic — efficiency considerations typically preclude a thoughtless distribution and require a careful decomposition of the system into (relatively independent) components. Nevertheless, location transparency is at least a first step in the direction of this ideal, and it helps to reduce the complexity of the application code.

Actually, both object and thread mobility have proved their value in practical implementations [Ananthanarayanan et al. 90, Bennett et al. 90, Chase et al. 92b, Garrett et al. 92, Nitzberg and Lo 91]. The approach depicted in Figure 4 is generally described in more abstract terms, disregarding the object structure of the data. Transparent access to remote memory locations based on moving the respective data to the requestor is usually referred to as *distributed shared memory* (DSM). In PANDA, DSM is provided to support object mobility.

Providing Distributed Shared Memory

One possible way to achieve perfect location transparency in a distributed environment would be to completely re-implement the memory management layer of the underlying C++ run-time system, and to base the execution of all programs exclusively on a DSM mechanism. Unfortunately, however, in many situations DSM turns out to be a rather expensive mechanism; a thoughtless use of shared memory objects may lead to permanent thrashing, intolerably reducing the efficiency of the program execution. Therefore, PANDA provides DSM only if the programmer explicitly requests for it.

From the programmer’s perspective, the allocation of an object in the DSM address space is straightforward. The following code fragment illustrates how it is done:

```
class Any; //Arbitrary user class
...
Any *pointer = new DSM Any;
```

All that has to be done is to insert the preprocessor directive 'DSM' between the `new` operator and the object class name; the preprocessor replaces this by a special parameter which is handed to the `new` allocation routine. The PANDA run-time support system takes this as a hint, and ensures that the newly created object instance resides in the DSM where it is accessible from all nodes of the system.

Implementation of the DSM Mechanism

Basically, the PANDA implementation of DSM is as follows. A global virtual address space is assigned to the Panda application. As shown in Figure 6, this address space is statically partitioned into several, disjoint memory partitions, (at least) one private partition for each node of the distributed system, and (at least) one shared partition per node. The private partition assigned to a node contains all private objects of that node. Such objects are not shared but can only be accessed by the threads running on the owner node of the respective partition. Consequently, the pages of the private partitions are only mapped into the address space of their respective owner node; remote nodes are not allowed to reference these memory locations. The second partition, however, comprises the DSM address space and contains those objects which are shared by all nodes; the pages of all shared partitions are mapped into the address space of all processes. Thus, every node may access any shared location.

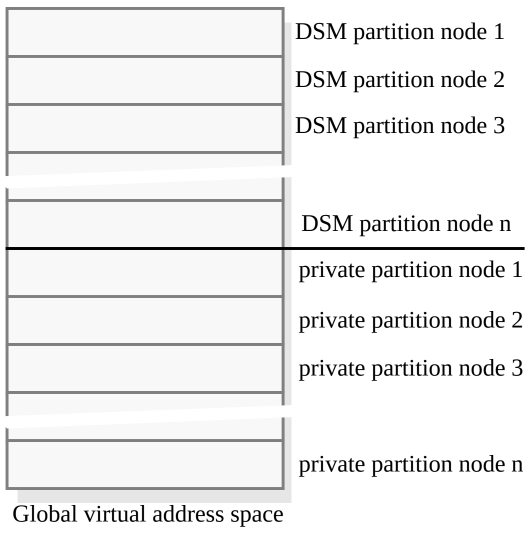

Fig. 6. Statically partitioning of memory

Note that a static partitioning of the address space is only feasible if the address space is sufficiently large. With the advent 64-bit architectures which is the emerging technology, spending some available memory addresses (recall that the virtual addresses of the private partitions assigned to the remote nodes are lost on the local host) in order to obtain a simpler and more efficient realization of the memory management mechanisms seems reasonable [Chase et al. 92, Chase et al. 92b, Garrett et al. 92].

Having partitioned the address space as described above, DSM memory management is now straightforward. All pages which are currently not available are read-write protected by the memory management unit. Whenever a thread tries to access an object which is not locally available, a page fault occurs. This page fault is handled by an appropriate page fault handler. If the address that caused the page fault refers to a private memory partition, this shows that a remote private partition has been referenced. As objects within such a non-local partition are not remotely accessible, access cannot be granted locally, that is, an exception has to be raised to indicate an access violation, or — depending on circumstances — the system may try to migrate the thread that caused the page fault to the node which owns the referenced partition (see below).

If the address that caused the page fault refers to a DSM partition, a different strategy applies. As DSM partitions are shared between all nodes, an access to an arbitrary DSM partition is always permissible, and should therefore always be granted. To this end, the page fault handler follows some protocol which links together the distributed physical DSM memory pages to a single virtual shared memory.

In the PANDA system, managing distributed memory and reacting to page faults is particularly easy, due to the static partitioning strategy that is applied. Note that for each virtual memory address, the system can immediately determine the corresponding partition. This knowledge helps to simplify the DSM protocol. For example, we can easily determine the node which is responsible for each partition, and that node can coordinate all activities related to the management of the partition's virtual pages, for example, their retrieval, and the invalidation of read copies.

In principle, the PANDA system is open to all DSM protocols that are known from the literature [Nitzberg and Lo 91]. However, an appropriate page fault handler has to be provided which implements the chosen strategy. In the current PANDA prototype system, a simple approach (exclusive-read-exclusive-write) has been chosen, mainly because we were able to provide this functionality with a trivial DSM manager (approximately 150 lines of code). Currently, we are about to realize several more sophisticated DSM protocols and to study their performance benefits.

Providing Thread Mobility

An approach to achieve location transparency which is dual to DSM is based on thread mobility. Instead of moving the object to the thread where it is needed (*data shipping*), we may as well move the thread to the object (*function shipping*). None of these approaches is superior to the other in general; their appropriateness depends on the actual object and thread granularity, and also on the object reference patterns that occur in the application at hand [Ananthanarayanan et al. 90, Dasgupta et al. 91, Jul et al. 88].

In PANDA, thread migration is performed by a method of the already mentioned class `UserThread`. This class provides the method `migrate` at its interface:

```
class UserThread {
  public:
    …     // Constructor, code() etc. …
    void migrate (destination);
};
```

Conceptually, invoking this method interrupts the thread's execution, transfers its stack segment to the appropriate partition on the specified remote node, moves its thread control block to the same destination, and inserts it into the appropriate queue of the destination node's scheduler. After this has been done, the thread's execution can be resumed. In practice, slight variations on this scheme are feasible to increase the efficiency of thread migration. For example, one might postpone the transfer of the stack segment and provide its memory pages only on demand.

From an abstract point of view, thread mobility is a possible realization of function shipping. In Figure 5, we showed how function shipping may serve to provide location transparency, i.e., to create the illusion of a single, global address space which can be accessed from each node in the system. The basic idea is to provide a page fault handler which — on receiving a page fault interrupt — determines the current location of the referenced memory page, and then migrates the requestor to that location. This approach provides an alternative to the DSM mechanism discussed in the previous sections.

One could also imagine to combine data shipping with function shipping by simply providing both strategies within a single page fault handler. Based on the type of object that is requested, the type of thread which made the request, and on global load conditions in the system, the handler could decide on each page fault whether to move the object to the thread, or whether to move the thread to the object. We will further pursue such strategies in the future.

2.3 Persistent Objects

In conventional programming languages data is transient. Its lifetime maximally spans the program's execution. Very often, preserving data across multiple invocations of several programs is required. Moreover, some data should even survive system failures. Data having these properties is called *persistent*. In conventional environments, persistence is typically realized with the help of file systems or database interfaces. But the lack of integration makes these realizations rather difficult to handle. Using the file system to make data persistent imposes a lot of additional encoding work on the programmer. He has to "flatten" his data structures to make them storable and re-readable. If a database interface is used, the user has to learn a database language, and he has to fit his data structures into the model provided by the database. New approaches in object-oriented databases [ODeux et al. 91, Lamb et al. 91] try to better bridge the gap between programming languages and databases. A database, however, is rather resource intensive, and therefore probably not adequate for those applications which do not require its functionality to the full extent.

Thus, the integration of persistence mechanisms into the run-time environment of the language seems to be a reasonable alternative. Such an approach has been pursued — in particular — in systems designed for fault tolerant computing [Ahamad et al. 90, Liskov 88, Shrivastava et al. 91].

In PANDA, an integrated persistence mechanism has been realized. Persistence is modelled as an additional object attribute which is orthogonal to the existing features of the C++ language. Persistent objects are remotely accessible.

If a class is intended to be persistent in PANDA, the precompiler directive "persistent" has to be placed in front of its definition:

```
persistent class Confer {
  public:
    Confer (int year) { date = year; }
  private:
    int date;
};
```

Creation of a persistent instance of such a class is done by:

```
Confer *con = new “ECOOP” Confer(93);
```

Finding an existing persistent object is performed by:

```
Confer *con2 = new “ECOOP”;
```

One major problem concerning persistent objects is how to maintain their consistency in the presence of concurrent access and node failures. An abstraction which guarantees this requirement is the transaction principle.

PANDA offers distributed transaction support, restricted to persistent objects. By default, transaction brackets are automatically inserted by the precompiler at the beginning and the end of each method of a persistent class. However, there exists a precompiler directive which suppresses this default to allow manual instrumentation. Currently, nested transactions are not supported; recursive invocations are, of course, feasible. If a failure occurs, always the outermost transaction bracket is aborted. Only persistent objects are recovered. Our implementation of transactions is based on the page-fault mechanism. Shadow copies of modified pages are managed. Pages are stored on a simple block-oriented file system. Distributed transactions are based on two-phase-commit.

2.4 Garbage Collection

It is generally claimed that environments which support applications comprising a changing set of dynamically allocated objects have an urgent demand for automatic garbage collection. Distributed garbage collection has been intensively studied in the literature, in particular in the context of functional programming environments where dynamically allocated data is ubiquitous [Rudalics 88].

In imperative programming languages such as C++, however, an excessive use of dynamically allocated data is less common, and it is always explicitly requested by the programmer. Due to the language's block structure, most of the volatile data can simply be declared within the block scope, i.e., on the invocation stack; such *stack objects* are removed on block exit. In C++, true *heap objects* tend to be rather long-lived. Empirical studies have shown, however, that long-lived objects are rarely subject to garbage collection [Hayes 91, Sharma and Soffa 91]. As the application programmer is generally well aware of their existence, and of their lifetime, manual deletion is probably most effective.

The potential benefits of garbage collection may be further restricted if there is a shared access to complex dynamic data structures. Figure 7 shows an example. Sup-

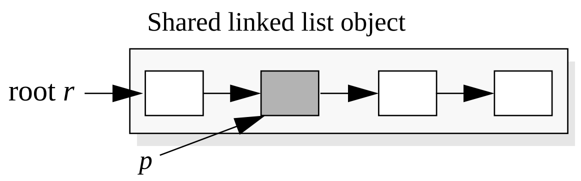

Fig. 7. Limitations of automatic garbage collection

pose that several threads share a common linked list, reachable via the root pointer *r*. One of these threads may call some list operation *drop*(*p*) to declare **p* as “no longer in use” from its local perspective. However, **p* cannot so easily be discarded, without knowing whether *all* other threads have also dropped *p*. Consequently, **p* has to remain in the list until it is certain that it has *semantically* turned into garbage. Unfortunately, the notion of semantical reachability is not adequately matched by the criterion of physical reachability on which all garbage collectors are based. Note that all list items will always remain *physically* reachable via *r* unless they are explicitly removed by the list management. Obviously, automatic garbage collection is of little help in such cases; instead, an explicit agreement protocol for the identification of garbage has to be provided by the programmer. Situations similar to our example are quite common in concurrent applications sharing linked data structures. Similar observations have been reported from systems based on an actor model [Kafura and Washabaugh 90].

Design Rationales

For efficiency reasons, PANDA supports a *co-existence* of ordinary objects with objects that are automatically garbage-collected. This means that a garbage-collectable object can be declared and used in roughly the same way as any other object. Being garbage-collectable should be mostly *orthogonal* to existing features of the C++ programming language [Detlefs 92, Edelson 90]. This rules out any form of added formalism deviating from normal C++ syntax.

We currently pursue a telecommunication exchange system application. There, *responsiveness* — i.e., the ability to react to incoming stimuli within a short and predictable interval of time — is one of the most stringent requirements, even more important than efficiency. Unfortunately, many of the most efficient distributed collectors known from the literature exhibit a stop-and-collect behavior which conflicts with the requirements of responsiveness. What is needed is an *incremental* garbage collection scheme which collects garbage on the fly. So-called *reference counting collectors* are well suited for that purpose [Bevan 89, Piquer 91, Rudalics 88], in particular because they can easily handle *references in transit* which are a major source of difficulties in distributed garbage collection schemes.

Another reason why stop-and-collect approaches are probably not an adequate choice is because they tend to leave garbage objects in the system for a substantial amount of time before they actually delete them. This may conflict with C++ *destructors*, methods which are called when their corresponding object is deleted. If (as it is often the case) a garbage collector treats garbage as a single, uniform memory space without considering the individual garbage objects, then this may completely preclude the invocation of any destructors [Edelson 90]. But even if the individual garbage objects *are* identified, but their deletion is delayed until the next detection phase, this conflicts with the semantics of destructors which are assumed to be *instantaneously* called on deletion (and may cause relevant side effects that must not suffer from any delays). These subtle requirements originating from C++ as well as the demand for responsiveness were our main motivation for choosing a simple reference counting scheme.

Language-Based Implementation

To support reference counting in C++, the method at hand is to employ so-called *smart pointers* [Detlefs 92, Edelson 92], i.e., reference objects which provide all basic functionality of an ordinary pointer, but transparently carry out additional semantic operations. This can be put nicely into practice, based only on features which are provided by C++.

We included two different garbage collection schemes in our PANDA run-time support system. The first follows the classical *reference counting* approach. The second strategy is based on positive weights which are assigned to each garbage-collectable object, and to each reference to such an object [Bevan 89].

The idea is to maintain the following invariant: The sum of the weight of an object and of the weights of all references that point to that object is a known constant *CONST*. To satisfy this invariant, an object and its original reference is initially equipped with a weight of *CONST*/2 each. Whenever a weighted reference to an object is copied, its weight is split into two shares. One share is handed back to the original reference, the other is assigned to the newly created reference. Accordingly, as soon as a reference is removed or overwritten, its current weight is returned to the object. Obviously, as soon as the weight of the object increases to *CONST*, all its references must have been deleted, and the object is garbage. For more details, the interested reader is referred to [Bevan 89] and [Schwarz 92]. The potential benefit of this scheme is that reference copying only involves the original reference and its copy, but not the (potentially remote) object. This may help to save communication bandwidth and copying delay.

PANDA supplies a base class `GC_obj` which provides the basic functionality required for (weighted) reference counting. An object class is made garbage-collectable simply by deriving it from `GC_obj`. All other objects (probably, the ma-

```
class UserType: public GC_obj {
  ...// add internals, e.g.:
  char *name;
};
UserType obj1, obj2;
GC_ref<UserType> ref1, ref2;
ref1 = ref2 = new UserType;
ref1->name = “AnyName”;
obj2 = *ref1;
if (ref2 == 0) cerr << “???\n”;
ref1 = 0;// implicit deletion;
ref2 = &obj1;// implicit deletion:
     // *** collection ***
```

Fig. 8. A code fragment showing the application of garbage collection

jority) are by no means affected by garbage collection. We regard this as an essential feature. Probably only a small fraction of all objects is subject to garbage collection. This will drastically reduce the computational overhead and the storage requirements of reference counting. Figure 10 shows a code fragment which illustrates how garbage collection can be applied. Note that garbage-collectable references — implemented by the template class `GC_ref<...>` — may be assigned, compared, and dereferenced like any ordinary pointer of the C++ language.

Potential Problems

Garbage collection based on smart pointers is subject to certain limitations. This general observation is particularly true for C++ which suffers from the burden of several C anachronisms. In [Edelson 92], the author discusses the weaknesses of smart pointers in C++. The three main sources of potential trouble are: Smart pointers to constant objects, implicit type casting and “pointer leakage”.

The first problem — the difficulty to provide the smart pointer equivalent of a `const Type` pointer — does not so much affect the functionality of smart pointers, but is just desirable for documenting and structuring reasons, and to provide a more rigorous compile-time checking of the application code. Edelson presents a solution to this problem which causes, however, some inconvenience for the application programmer.

The inability to extend the implicit type casting rules of C++ to smart pointers has more significant consequences. For example, ambiguous function calls are often made unambiguous by comparing the argument types with the signatures of the functions in question. This procedure depends on a number of implicit type conversion rules which the C++ compiler automatically applies to the function arguments. With smart pointers, such implicit casts are no longer feasible. In most cases, however, the programmer can simply specify an explicit type conversion to resolve ambiguous (or otherwise even illegal) situations. This may be inconvenient, but it is not a fatal handicap.

So-called pointer leakage [Edelson 92] is another potential pitfall. Note that the overloaded "->" operator must return an *ordinary* pointer. Therefore, once a temporary smart pointer has been dereferenced and "->" has returned an ordinary pointer, the smart pointer is no longer needed and may be destroyed. If it was the only reference to an object, then the object might be subject to garbage collection, although the ordinary pointer returned by "->" is still in use. The danger of pointer leakage can be reduced by careful encoding; unfortunately, aggressive optimizing compilers may sometimes try to eliminate “superfluous” intermediate variables, thus destroying the last smart pointer to an object.

Besides these shortcomings which are inherent to C++ and the smart pointer approach, the main weakness of our (weighted) reference counting strategy is its inability to identify cyclical garbage structures. On the other hand, a cyclical structure often represents *one* entity of the application domain which should rather be represented by an *atomic* object providing a specialized delete operation. A trick that might sometimes help to combine user-defined destructors with automatic garbage collection is illustrated in Figure 9. The basic idea is to exploit the compatibility between ordinary pointers and reference objects, and to break the cycle by inserting at least one simple pointer in the reference chain.

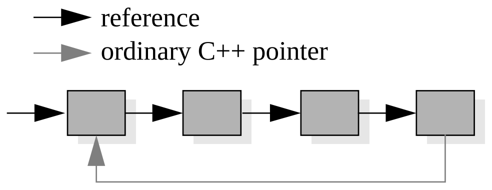

Fig. 9. Breaking cycles by mixing references with ordinary pointers

Despite of the problems mentioned above, we think that the advantages of our solution outweigh its potential drawbacks:

- Only garbage-collectable objects are penalized by the collector.
- Garbage is eliminated incrementally.
- The overhead is proportional to the access frequency to garbage-collectable objects.
- Our strategy complies with the semantics of C++ destructors.
- Our approach does not depend on any special system support, and it is applicable in a parallel and distributed setting.

3. System Architecture

The design of our system very much benefits from current hardware trends, which will have a major impact on future system architectures in general [Karshmer and

Nehmer 91]. We expect that multiprocessor workstations connected by a high-speed network capable of a bandwidth of 1 GB/s will soon become the standard working environment. The availability of 64-bit wide machine addresses will allow the use of a global address space shared by several applications [Chase et al. 92].

Object-Oriented System Design

In order to maximize extensibility, exchangeability, maintainability, and to minimize development time, an object-oriented design of our system has been chosen. We regard it as an adequate approach to structure the operating system kernel, the run-time support, and also the applications. Hence, we focused on a single programming language for both application and system software. This approach prevents the obstacles and the performance penalties entailed by interface mismatch. Moreover, structuring the system in an object-oriented manner simplifies the replacement of its components, thus facilitating experiments regarding various system configurations. Derivation techniques in combination with dynamic binding and polymorphism allow a gradual refinement of system components and improve structural clarity.

Main Components

As shown in Figure 10, the system is divided up into three different layers: pico-kernel, run-time package, and application layer. The objects within each layer form a set of exchangeable building blocks. This technique simplifies maintenance and the integration of new or customized components.

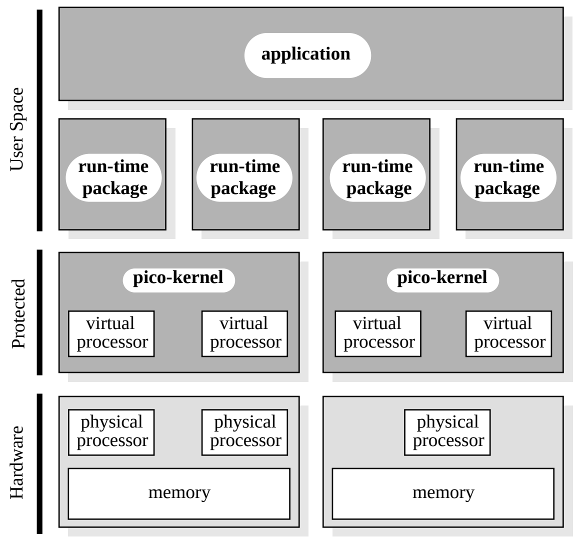

Fig. 10. PANDA system layers

A single application may span several run-time packages. There is a one-to-one mapping between run-time packages and virtual processors. Several virtual processors may share a single physical processor, whereby scheduling is done on a preemptive single-priority FIFO basis. This policy prevents an application from processor monopolization. The current scheduling policy can, however, easily be adapted to specialized application domains. For example, a fixed time slot may be assigned to a virtual processor in order to meet the requirements of real-time systems.

The pico-kernel offers only those hardware abstractions which are critical in respect to protection and monopolization considerations: protection domain management and virtual processor scheduling. The code size of the pico-kernel is reduced to an absolute minimum because only those functions remain in the kernel which actually require the privileged instructions of the processor.

All kernel calls are non-blocking. This allows the continuation of user-level threads within the run-time package. No thread will ever block within the kernel stopping the whole run-time package. For a further discussion of problems related to the implementation of user-level threads, the reader is referred to [Anderson et al. 92, Draves et al. 91, Marsh et al. 91].

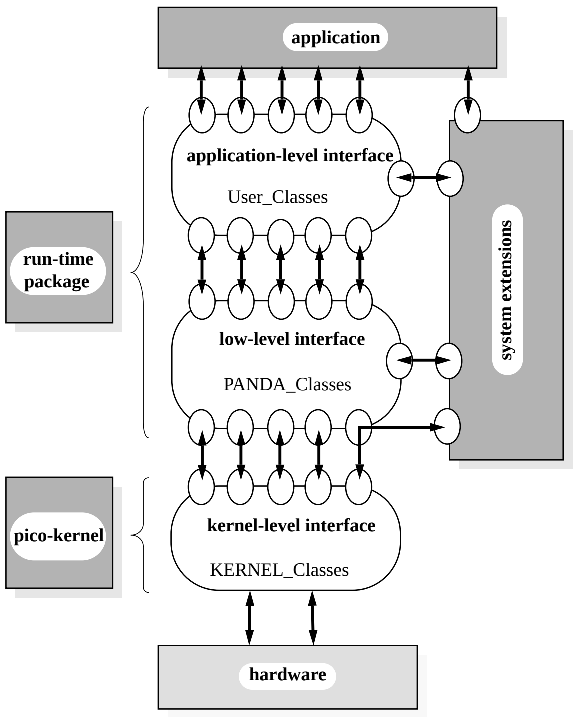

Fig. 11. PANDA interfaces

At the first glance, assigning only one virtual processors to each run-time package seems to be restrictive. Note that it precludes true parallelism between threads within the same run-time package. On the other hand, scheduling a single virtual processor can be accomplished without additional overhead for locking. This eliminates the need for special multiprocessor mechanisms, discussed, e.g., in [Anderson et al. 89]. By saving the synchronization costs in this critical code section, substantial performance gains are obtained. An application may still benefit from a multiprocessor platform by simply employing several run-time packages. One drawback of this approach is that it is up to the application to evenly distribute load over the available processors. However, global load distribution may be implemented by using thread migration between the run-time packages.

System support such as communication service, different DSM services, migration service, or persistence service are realized as exchangeable components of the run-time package. Most of these servers are hidden from the application by higher-level abstractions. In general, an application uses the *application-level interface*. It may, however, also access the *low level interface*, but only "applications" meant as system extensions — such as a load distribution facility — are encouraged to do so. The three different interfaces are provided by corresponding class families (see Figure 10).

A valuable tool provided by our programming environment is a C++ precompiler. It facilitates the instrumentation of programs with stub classes, statistical counters, monitoring hooks, transaction brackets, method add-ons etc. In contrast to a mere preprocessor, this precompiler is fully aware of the syntactical structure, which allows the integration of new language constructs, proposed, e.g., in [Gehani and Roome 88, Buhr and Stroobosscher 92]. But this is a technique which we try to avoid because the precompiler should not be a basic requirement for our system.

4. Status

We currently support the processor types SPARC (32 bit), Motorola 680xx, and Intel i386/i486. The main development stream uses SPARC processors, the others are updated and adopted from time to time. Currently, remote communication via a raw Ethernet and a TCP/IP or UDP connection is supplied. Our system is highly portable because the pico-kernel's hardware dependencies are limited to a context switch facility, MMU access, and the interrupt handler interface.

For convenience, most of the development is done by using a pico-kernel emulation on top of UNIX (SunOS 4.1.1). As this environment does not yet reflect new hardware trends like, for example, 64-bit wide address spaces and high-speed local area networks, we are currently restricted to prototype applications.

Performance

In order to put the PANDA system into perspective, some preliminary performance measurements are listed in Table 1. Our results compare quite well with the performance of FastThreads [Anderson et al. 92], Presto [Faust and Levy 90], and Psyche [Marsh et al. 91]. The measurements were carried out on Sun SPARC-2 workstations connected by a 10 Mb/s Ethernet, with a pico-kernel emulation on top of SunOS 4.1.1. Comparing the null procedure with a null process (creation, context switch, null execution, and deletion), it should be taken into account that the SPARC

Benchmark operation	Time
Thread creation, null execution, and deletion (with/ without lazy stack allocation)	15.1 µs / 10.7 µs
Context switch	17.7 µs
Remote message (8 bytes) transfer (ping / burst)	1.8 ms / 0.3 ms
Local memory allocation and deletion	1.5 µs
Remote memory access via DSM page transfer	9.6 ms
Null procedure call (with / without register window overflow)	7.1 µs / 0.31µs

Tab. 1. Performance of the PANDA prototype

architecture is particularly tuned to allow extremely fast procedure calls; context switches, on the other hand, are relatively slow, and only poor support for user-level threads is provided [Anderson et al. 91].

5. Conclusions and Future Work

The results gained so far are encouraging. PANDA has proved to be a practical and efficient support for parallel and distributed programming. Due to our compliance with standard C++, an industrial partner is seriously considering our system as a platform for telecommunication software, and is currently employing it for prototype implementations. Thus, we are able to obtain valuable feedback regarding our design choices.

We also explore PANDA's capability to serve as a base for various concurrent and distributed programming models. To this end, we implemented a run-time support layer for COIN [Buhler 90], a programming language especially designed for parallel and distributed applications. As expected, PANDA's low-level interface provided a suitable set of abstractions for this purpose, and no major performance penalties had to be payed due to interface mismatch.

Our experience with C++ is twofold. On the one hand, it is an emerging standard which provides access to a variety of software, and which allows very efficient system programming. On the other hand, it has definite disadvantages as far as concurrency and distribution are concerned. We are well aware of the fact that C++ is probably not an ideal language to address these issues. Therefore, our work is directed towards two main goals. First, we would like to enhance our environment by adding C++ classes to deal with problems related to — in particular — robustness and persistence. Second, we will also investigate how the aspects of parallelism, distribution, and persistence can be more suitably reflected in an object-oriented programming language.

Acknowledgments

We would like to thank Peter Sturm for many fruitful discussions, and Thomas Gauweiler for his technical help.

References

[Ahamad et al. 90] M. Ahamad, P. Dasgupta, and R.J. Leblanc, Jr. *Fault-tolerant Atomic Computations in an Object-Based Distributed System.* Distributed Computing, Vol. 4, pp. 69-80, 1990.

[America and van der Linden 90] P.H.M. America and F. van der Linden. *A Parallel Object-Oriented Language with Inheritance and Subtyping.* Proc. of European Conference on Object-Oriented Programming and ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (Ottawa, Canada, Oct. 21-25), ACM, New York, pp. 161-168, 1990.

[Ananthanarayanan et al. 90] R. Ananthanarayanan et al. *Experiences in Integrating Distributed Shared Memory With Virtual Memory Management.* Technical Report GIT-CC-90/40, Georgia Institute of Technology, College of Computing, Atlanta, GA, 1990.

[Anderson et al. 89] T.E. Anderson, E.D. Lazowska, and H. Levy. *The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors.* IEEE Trans. Comput. 38, 12 (Dec.), pp. 1631-1644, 1989.

[Anderson et al. 91] T.E. Anderson, H.M. Levy, B.N. Bershad, and E.D. Lazowska. *The Interaction of Architecture and Operating System Design.* Proc. ACM SIGOPS '91, 1991.

[Anderson et al. 92] T.E. Anderson, B.N. Bershad, E.D. Lazowska, H.M. Levy. *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.* ACM Transactions on Computer Systems, Vol. 10, No. 1, Feb. 1992.

[Assenmacher 92] H. Assenmacher. *The PANDA Run-time Package*. Technical Report, Department of Computer Science, University of Kaiserslautern, Germany, 1992.

[Bal et al. 92] H.E. Bal, M.F. Kaashoek, and A. S. Tanenbaum. *Orca: A Language for Parallel Programming of Distributed Systems.* IEEE Trans. Softw. Eng. 18(3), pp. 190-205, March 1992.

[Bennett et al. 90] J.K. Bennet, J.B. Carter, W. Zwaenepoel. *Adaptive Software Cache Management for Distributed Shared Memory Architecture.* Proc. 17th Annual Int. Symposium on Computer Architecture, Seattle, Washington, pp. 125-134, May 1990.

[Bershad et al. 88] B.N. Bershad, E.D. Lazowska, H.M. Levy, and D. Wagner. *An Open Environment for Building Parallel Programming Systems.* Proc. of the ACM SIPLAN PPEALS - Parallel Programming: Experience with Applications, Languages, and Systems, (July 19-21), pp. 1-9, July 1988.

[Bevan 89] D.I. Bevan. *An Efficient Reference Counting Solution to the Distributed Garbage Collection Problem.* Parallel Computing 9, pp. 179-192, 1989.

[Birrell and Nelson 84] A.D. Birrell and B.J. Nelson. *Implementing Remote Procedure Calls.* ACM Transactions on Computer Systems, 2(1), Feb. 1984.

[Buhler 90] P. Buhler. *The COIN Model for Concurrent Computation and its Implementation.* Microprocessing and Microprogramming 30, No. 1-5, North Holland, pp. 577-584, 1990.

[Buhr and Stroobosscher 92] P.A. Buhr and R.A. Stroobosscher. μ*C++ Annotated Reference Manual.* University of Waterloo, Canada, Aug. 1992.

[Chase et al. 89] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield. *The Amber System: Parallel Programming on a Network of Multiprocessors.* Proc. of the 12th ACM Symposium on Operating Systems Principles, pp. 147-158, 1989.

[Chase et al. 92] J.S. Chase et al. *How to Use a 64-Bit Virtual Address Space*. Technical Report 92-03-02, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1992

[Chase et al. 92b] J.S. Chase et al. *Lightweight Shared Objects in a 64-Bit Operating System.* Technical Report 92-03-09, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1992.

[Dasgupta et al. 91] P. Dasgupta, R.J. LeBlanc, M. Ahamad, and U. Ramachandran. *The Clouds Distributed Operating System.* IEEE Computer, pp. 34-44, Nov. 1991

[Detlefs 92] D. Detlefs. *Garbage Collection and Run-time Typing as a C++ Library*. Proc. USENIX C++ Conference, Portland, Oregon, Aug. 1992.

[Draves et al. 91] R.P. Draves, B.N. Bershad, R.F. Rashid, and R.W. Dean. *Using Continuations to Implement Thread Management and Communication in Operating Systems.* Proc. of the 13th ACM Symposium on Operating Systems Principles, Oct. 1991.

[Edelson 90] D.R. Edelson. *Dynamic Storage Reclamation in C++*. Technical Report UCSC-CRL-90-19, University of California at Santa Cruz, CA, June 1990.

[Edelson 92] D.R. Edelson. *Smart Pointers: They're Smart, but They're Not Pointers.* Proc. USENIX C++ Conference, Portland, Oregon, pp. 1-19, Aug. 1992.

[Ellis and Stroustrup 90] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[Faust and Levy 90] J.E. Faust and H.M. Levy. *The Performance of an Object-Oriented Thread Package.* Proc. of European Conference on Object-Oriented Programming and ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (Ottawa, Canada, Oct. 21-25), ACM, New York, pp. 278-288, Oct. 1990.

[Garrett et al. 92] W. E. Garrett, R. Bianchini, L. Kontothanassis, R.A. McCallum, J. Thomas, R. Wisniewski, and M.L. Scott. *Dynamic Sharing and Backward Compatibility on 64-Bit Machines*. TR 418, University of Rochester, Computer Science Department, Rochester, NY, April 1992.

[Gehani and Roome 88] N.H. Gehani and W.D. Roome. *Concurrent C++: Concurrent Programming with Class(es).* Software — Practice and Experience 18(12), pp. 1157-1177, Dec. 1988.

[Habert et al. 90] S. Habert, L. Mosseri, and V. Abrossimov. *Cool: Kernel Support for Object-Oriented Environments.* Proc. of European Conference on Object-Oriented Programming and ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (Ottawa, Canada, Oct. 21-25), ACM, New York, pp. 269-277, Oct. 1990.

[Hayes 91] B. Hayes. *Using Key Object Opportunism to Collect Old Objects.* Proc. OOPSLA'91, Phoenix, Arizona, Oct. 1991.

[Jul el al. 88] E. Jul, H. Levy, N. Hutchinson, and A. Black. *Fine-grained Mobility in the Emerald System.* ACM Trans. Comput. Syst. 6(1), pp. 109-133, Feb. 1988.

[Kafura and Washabaugh 90] D. Kafura and D. Washabaugh. *Garbage Collection of Actors.* Proc. ECOOP/OOPSLA'90, pp. 126-134, Oct. 1990.

[Karshmer and Nehmer 91] A. Karshmer and J. Nehmer (eds.). *Operting Systems of the 90s and Beyond.* LNCS 563, Springer-Verlag, 1991.

[Lamb et al 91] C. Lamb, G. Landis, J. Orenstein, D. Weinreb. *The Objectstore Database System.* Communications of the ACM, Vol. 34, No. 10, pp. 50-63, Oct. 1991.

[Liskov 88] B. Liskov. *Distributed Programming in ARGUS.* Comm, ACM, Vol. 31, No. 3, pp. 300-312, March 1988.

[Lucco 87] S.E. Lucco. *Parallel Programming in a Virtual Object Space.* Proc. of ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (Orlando, Fla., Oct. 4-8). ACM, New York, pp. 26-34, Oct. 1987.

[Marsh et al. 91] B. D. Marsh, M.L. Scott, T.J. LeBlanc, and E.P.Markatos. *First-Class User-Level Threads.* Proc. of the 13th Symp. on Operating Systems Principles, Pacific Grove (California), pp. 110-121, Oct. 1991.

[Nierstrasz and Papathomas 90] O.M. Nierstrasz and M. Papathomas. *Viewing Objects as Patterns of Communicating Agents.* Proc. of European Conference on Object-Oriented Programming and ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (Ottawa, Canada, Oct. 21-25), ACM, New York, pp. 38-42, 1990.

[Nitzberg and Lo 91] Nill Nitzberg and Virginia Lo. *Distributed Shared Memory: A Survey of Issues and Algorithms.* IEEE Computer, pp. 52-60, Aug. 1991.

[ODeux et al. 91] O. Deux et al. *The O_2 System.* ACM Communications of the ACM, Vol. 34, No. 10, pp. 34-48, Oct., 1991.

[Piquer 91] J. Piquer. *Indirect Reference Counting: A Distributed Garbage Collection Algorithm.* In: LNCS 505, E.H.L. Aarts, J. van Leeuwen, M. Rem (eds.), Springer-Verlag, pp. 150-165, 1991.

[Rudalics 88] M. Rudalics. *Multiprocessor List Memory Management.* Technical Report RISC-88-87.0, Research Institute for Symbolic Computation, J. Kepler University, Linz, Austria, 1988.

[Saleh and Gautron 91] H. Saleh and P. Gautron. *A Concurrency Control Mechanism for C++ Objects.* Proc. of the ECOOP '91 Workshop on Object-Based Concurrent Computing (Geneva, Switzerland, July 15-16), LNCS 612, Springer-Verlag, pp. 95-210, 1991.

[Schwarz 92] R. Schwarz. *Language-based Garbage Collection in the PANDA System.* Internal Report, Department of Computer Science, University of Kaiserslautern, Germany, 1992.

[Sharma and Soffa 91] R. Sharma and M. L. Soffa. *Parallel Generational Garbage Collection.* Proc. OOPSLA'91, Phoenix, Arizona, Oct. 1991.

[Shapiro 86] M. Shapiro. *Structure and Encapsulation in Distributed Systems: The Proxy Principle.* Proc. 6th Int. Conference on Distributed Computer Systems, pp. 198-204, Cambridge, MA, May 1986.

[Shrivastava et al. 91] S.K. Shrivastava, G.N. Dixon, G.D. Parrington. *An Overview of the ARJUNA Distributed Programming System.* IEEE Software, pp. 66-73, Jan. 1991.

[Stroustrup 86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1986.

[Wybranietz and Buhler 89] D. Wybranietz and P. Buhler. T*he LADY Programming Environment for Distributed Operating Systems.* Proc. of the PARLE'89 Conference, Eindhoven, Holland, Juni 1989. Springer-Verlag, LNCS 365, pp. 100-117, 1989.