Clix* – A Hybrid Programming Environment for Distributed Objects and Distributed Shared Memory

Frank Mueller[1], Jörg Nolte[2], and Alexander Schlaefer[3]

[1] Humboldt University Berlin, Institut f. Informatik, 10099 Berlin, Germany
[2] GMD FIRST, Rudower Chaussee 5, D-12489 Berlin, Germany
[3] University of Washington, CSE, Box 352350, Seattle, WA 98195-2350, USA
`mueller@informatik.hu-berlin.de, phone: (+49) (30) 2093-3011, fax: -3010`

Abstract. Parallel programming with distributed object technology becomes increasingly popular but shared-memory programming is still a common way of utilizing parallel machines. In fact, both models can coexist fairly well and software DSM systems can be constructed easily using distributed object systems. In this paper, we describe the construction of a hybrid programming platform based on the ARTS distributed object system. We describe how an object-oriented design approach provides a compact and flexible description of the system components. A sample implementation demonstrates that three classes of less than 100 lines of code each suffice to implement sequential consistency.

1 Introduction

Object-oriented programming and distributed object technology are considered to be state of the art in distributed and as well as parallel computing. However, typical numerical data-structures like huge arrays or matrices are hard to represent in a distributed object paradigm. Such data structures usually cannot be represented as single objects because this leads to extremely coarse-grained programs thus limiting parallel execution. On the other hand, it is not feasible to represent, e.g., each array element as a remote object because remote object invocation mechanisms are typically not targeted for fine-grained computation.

Distributed shared memory can, in principle, cope better with typical numerical data structures: The idea of shared memory is orthogonal to objects and coarse-grained data-structures can transparently be distributed across multiple machines. This scenario is specifically attractive to networks of SMPs and legacy software in general since the shared memory paradigm is not tied to a specific programming language. However, for performance reasons it is usually necessary to relax memory models and the programmer potentially has to cope with different memory consistency models. Object-oriented programming helps to structure memory models and a distributed object platform eases the implementation of consistency models and associated synchronization patterns significantly.

In this paper, we describe the design and implementation of an experimental integrated platform for distributed shared memory and distributed objects. This

* Clustered Linux

platform is based on a guest-level implementation of ARTS, a distributed object platform for parallel machines and the VAST framework for volatile and non-volatile user-level memory management [5].

2 The ARTS Platform

Originally, ARTS was a remote object invocation (ROI) system designed as an integral part of the parallel PEACE operating system family [11]. Today ARTS is an open object-oriented framework, which provides the basic services for distributed and parallel processing within a global object space. The ARTS platform provides language-level support for proxy-based distributed and parallel programming. Standard C++-classes are extended by annotations to specify remote invocation semantics as well as parameter passing modes similar to an IDL. These annotations are preprocessed by a generator tool called DOG [5], that generates proxy classes and stubs for synchronous and asynchronous ROI. In addition, collective operations on distributed object groups are also supported.

All annotations are specified as pseudo comments to retain a strong backward compatibility with standard C++. Figure 1 shows an example from the DSM system. A `/*!dual!*/` annotation is used to mark classes whose instances

```
class SeqConsistency {
 public: ...
  void setObjectControl (int node,
    /*!copy!*//*!dual!*/SeqObjectControl* ctrl);
  ...
  void setHome (int objectId);
  void acquireR (int objectId, int sender);
  void acquireW (int objectId, int sender);
  void ackR (int objectId, int sender) /*!async!*/;
  void ackW (int objectId) /*!async!*/;
}/*!dual!*/;
```

Fig. 1. The Dual Sequential Consistency Class

need to be accessed remotely. Those classes are referred to as *dual* classes. Note that the notation `/*!dual!*/SeqObjectControl` is treated as a type specifier that actually refers to two objects: a local proxy object in the client's address space and a remote object encapsulated in some remote server's address space. Any time the client performs a method on the proxy, the corresponding operation is executed on the remote object using ROI techniques. Thus, the global sharing of objects is possible through passing of proxy objects as parameters to remote methods. Parameters to remote methods are passed by value by default. Pointer or reference parameters can be annotated to be treated as input parameters (`/*!in*!/`), output parameters (`/*!out!*/`) or both (`/*!inout!*/`). Permanent copies of passed parameters on the server's heap can be created using a `/*!copy!*/` annotation as in the `setObjectControl()` method (Fig. 1).

Remote method invocation is synchronous if not specified otherwise. An `/*!async!*/` annotation as in `ackR()` can be applied to mark methods that shall be executed asynchronously.

Object groups are described by group classes that define the topology of a distributed object group. Groups are implemented as distributed linked object sets that can be collectively addressed by multicast operations. In principle, a multicast group can have any user-defined topology such as n-dimensional grids, lists or trees. We provide a default tree-based implementation of a distributed object group called `GroupMember` and apply inheritance mechanisms to specify a group membership. When a `GroupMember` class is inherited by a dual class it is possible to create and address whole object groups using a `GroupOf` template.

ARTS provides numerous synchronization mechanisms. However, for space considerations, we will only describe the continuation-based approach here. Whenever a method detects that it cannot be executed immediately because the respective object is, e.g., not yet in a suitable state, the method can block the actual caller and generate a `Continuation` instance for the call. These continuation objects can be stored and used later to reply a result to the client, thus deblocking the client when appropriate. Continuation objects can also be passed as parameters to remote methods. Consequently, it is very easy to forward invocations asynchronously to other objects that might perform activities in parallel and finally reply the result to the initial client at completion of an action.

3 Distributed Shared Memory Abstractions

For performance reasons, we need to support several memory consistency models. Therefore, we adopt a NUMA approach and divide the logical address space into separate memory regions that are each associated with a specific shared memory semantics. Thus, the memory semantics of each shared variable is determined by the coarse-grained memory container in which the variable is located.

The easiest way to achieve this is to mark memory blocks of variables declared in the file scope (global variables):

```
BEGIN(strict); int x; double m[512][512]; END(strict);
```

`BEGIN` and `END` declare the beginning as well as the end of a shared memory area and the argument of the macros specify the memory consistency semantics for this area. Both macros implicitly declare an array that has exactly the size of a physical page. Therefore, we are able to align the address of the area in which the variables are declared to a page boundary. This is necessary to use the MMU support. Likewise, we need to declare an object of page size at the end of such a region to ensure that regions do not overlap.

Memory control objects refer to the beginning and the end of the region they control and set up memory protection mechanisms accordingly. Since we assume SPMD programs, each region is locally mapped to the same address. On Linux systems, we apply the `mprotect()` system call to protect and unprotect pages, page faults are propagated to the standard bus error or segmentation fault signals. These basic mechanisms are implemented by a `ControlledArea` class provided by the VAST framework. This class emulates a complete page table and provides the basis to all segment types that need to catch page faults and handle them according to a specific strategy.

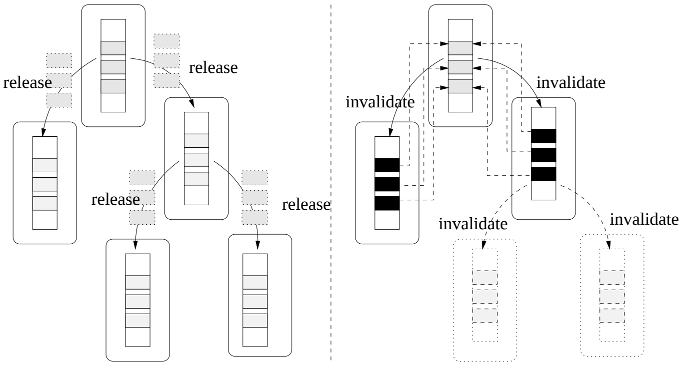

Fig. 2. Collective Operations for Different Modification Modes

All locally mapped regions can be joined to a distributed group that can be collectively addressed. Tasks like global page invalidation or update protocols are, therefore, very easy to implement (Fig. 2).

4 Object-Oriented Design of the Clix-DSM

The properties of a DSM protocol typically include the following aspects: The **consistency model** (sequential, release, entry, scope, etc.), the **granularity** of an object (byte, word, page, any), the **sharing resolution** that still guarantees a consistent view (bit, byte, word), the **directory** to determine who handles requests for accesses (home-base or migration-base), the **modification mode** for propagating changes (update or invalidate) and the **communication mode** for request handling (synchronous or asynchronous).

Each of these aspects has its own abstraction: Memory objects provide the proper sharing resolution, memory controls to oversee access to an object at the granularity level and directory items record the internal state of an object. In addition, aspects of the implementation of a concrete consistency model are isolated into separate classes to allow their reuse in the context of other consistency models, e.g., copy sets record nodes who have copies of a memory object and a scheduler queues and issues incoming requests for synchronous communication. Finally, the abstractions are combined to realize a concrete consistency model by defining each of the aspects above. A concrete consistency model also provides the operational semantics for message exchanges, which then determine the modification and communication mode. The design also supports multi-threading for each node both on the implementation level and the user level. The next section provides more details for an actual implementation of a DSM protocol.

5 Sample Implementation

In the following, several aspects of a sample implementation of sequential consistency of a page-based protocol with invalidation, a static home node per page and synchronous communication shall be given. Sequential consistency is implemented by two classes that are tightly coupled: `SeqConsistency` (SC) (Fig. 1) and `SeqObjectControl` (SOC) (Fig. 3).

```
class SeqObjectControl {
 public: ...
  void protect(int objectId);
  void unprotect(int objectId);
  void invalidate(int objectId);
  void setMemObjectW(int objectId,/*!in!*/MemPage* pMemObject)/*!async!*/;
  void setMemObjectR(int objectId,/*!in!*/MemPage* pMemObject,
    /*!in!*/ Continuation* pContinuation) /*!async!*/;
  void copyMemObjectW(int objectId, int node) /*!async!*/;
  void copyMemObjectR(int objectId, int node,
    /*!in!*/ Continuation* pContinuation ) /*!async!*/;
  void unprotectW(int objectId) /*!async!*/;
  void invalidateW(int objectId) /*!async!*/;
}/*!dual!*/;
```

Fig. 3. The Sequential Object Control Class

The two classes are separated due to their communication structure. SC implements an external server, i.e., it accepts requests from other nodes and handles them. Requests are handled in mutual exclusion since it is a dual class whose instances resemble a monitor on the current node. The actions for handling a request may include modifications of the access right for a memory objects, which are handed off to the corresponding SOC instance. The SOC delegates the task of modifying the MMU access rights to the MemorySegment (MS), an instance of a non-dual class derived from the VAST framework (Fig. 4).

MS overloads the handler of page faults defined by VAST. The handler is an integral part of the consistency model but cannot be transfered into SC and SOC since it is triggered by a page fault controlled through VAST. Upon activation of the handler, it simply distinguishes read and write accesses and issues a remote object call to the corresponding method of the owner's SC instance. In the case of a home-based system, the SC instance of the memory object is selected that corresponds to the home node to delegate the request. The called method is `acquireR()` or `acquireW()` for read and write faults, respectively.

```
class MemSegment: public ControlledArea {
 public: ...
  void initSegment(caddr_t area, size_t size, size_t psize);
  void protect(int objectId);
  void unprotect(int objectId);
  void invalidate(int objectId);
  void setConsistency(int objectId, /*!dual!*/SeqConsistency* pSeqCons);
};
```

Fig. 4. The Memory Segment Class

The example in Fig. 5 describes the operations for a read fault on node R. The fault handler of the MS (a regular class) synchronously calls `acquireR()` of the SC (a dual class, depicted as a shaded box) on the page's home node H in step (1). The request is queued on the home node and a continuation is generated, which blocks the synchronous caller for now but allows further requests to be received on H. If no prior request was active, the next request

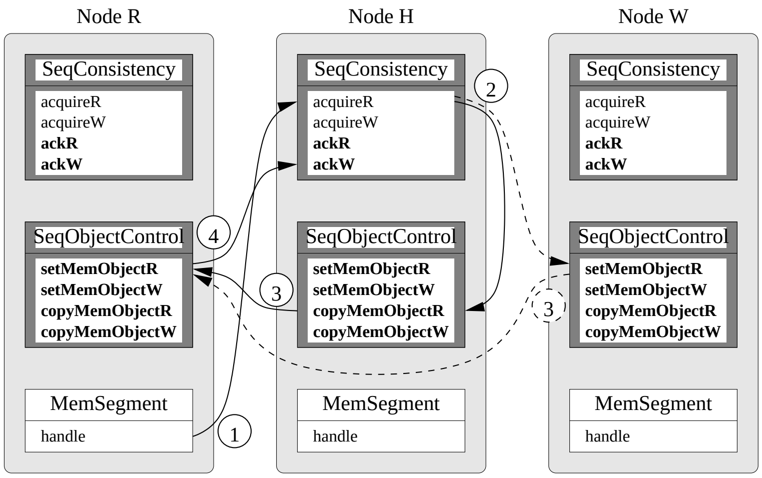

Fig. 5. Read Fault Handling

is dequeued and handled, which would be R's request in this case. As a result, R is added to the copy set. If no writer was active, the SOC of H is requested to hand over the page to node R via `copyMemObjectR()` in step (2). Notice that this request involves an asynchronous remote object invocation (depicted as a bold face method) to prevent the SC from blocking since it may have to accept further requests. The page is transfered from H's SOC to R's SOC via `setMemObjectR()` through another asynchronous call in step (3) and the faulting thread on node R is reactivated through a continuation. Finally, an acknowledgement is sent asynchronously from R's SOC to H's SC in step (4), which would result in handling the next request if H's queue is not empty. Had some other node W had write access to the page, then W would have been instructed by H's `acquireR()` method to reduce page rights to read-only and pass a copy to node R via `copyMemObjectR()`, depicted by the dashed arcs of steps (2) and (3), respectively. Write faults are handled in a similar fashion by the orthogonal methods for write access.

The advantages of the object-oriented design become apparent when the size of the implementation is regarded. SC, SOC and MS consist of slightly under 100 lines of code (excluding comments), out of which 10 lines comprise the page fault handler. This remarkably small implementation facilitates its validation considerably during the software development cycle since testing and debugging are restricted to a small amount of code. The prerequisite for such abstractions are underlying components that have been tested independently and can be reused for other protocols. In fact, this implementation of sequential consistency also illustrates how other protocols can be implemented.

Clix also provides synchronization in a distributed environment. The mechanisms range from FIFO-Locks and condition variables in a POSIX-like style to semaphores and barriers. All synchronization mechanisms are implemented as a separate class over ARTS' communication mechanisms or on top of each other and comprise about 20 lines of code each.

Other consistency models can be supported in much the same way as sequential consistency. Weak consistency models of pages can already fall back onto a class `Diff` (see Fig. 6) that creates a run-length difference between two memory pages or applies a difference to a page by updating it with the new values. This

```
template<class Granularity, class MemObj>
class Diff {
 public:
  Diff(MemObj& obj1, MemObj& obj2) { ... } // create diff of 2 objects
  Diff(char* buf,size_t buf_size) { ... }  // byte stream constructor
  Diff(const Diff& source) { ... }         // copy constr. (for Arts ROI)
  ~Diff() { ... }                          // destructur (for Arts ROI)
  void merge(MemObj& obj) { ... }          // apply diff object locally
  char* addr() { ... }                     // address of byte stream
  size_t size() const { ... }              // length of byte stream
}
```

Fig. 6. Diffs with Arbitrary Sharing Resolutions

class is provided as a template class and can be instantiated to support arbitrary sharing resolutions (called granularity in the figure) down to the level of a byte. Thus, different sharing resolutions may be supported for each memory object under the same consistency model. The user has the privilege to decide whether a coarse resolution is sufficient for certain tasks of the application, thereby yielding better performance, or if a smaller granularity is required due to the choice of data structures within other parts of the application.

6 Related Work

Software DSM system have been realized for a number of consistency models, sometimes even with an object-oriented approach. Implementations of sequential consistency, such as by Li and Hudak [10], were non-object based. The models of release and scope consistency typically do not build on objects either [9, 8]. Entry consistency, on the other hand, is often associated with the object-oriented paradigm due to the granularity of the protocol [3, 6]. The Panda system [2] used similar design goals but focused on transaction-based DSMs and persistent objects based on C++ language extensions that were not as powerful as those of ARTS. None of these approaches take advantage of facilities such as remote object invocation on the ARTS environment or VAST's memory management. Consequently, the implementation of their protocols tend to be more complex than in our CLIX approach. The ARTS platform incorporates many features similar to those found in common middleware layers like CORBA as well as parallel programming languages such as CC++ [7], pC++ [4] and Mentat [1]. However, system programming still needs significantly more control over runtime issues than languages designed for application level programming usually provide. ARTS originated in the area of parallel operating systems and, therefore, it does not rely on complex compiler technology but extensible OO-frameworks. Thus most remote object invocation mechanisms as well as high-level collective operations can be controlled by standard C++ inheritance mechanisms. The

VAST framework originated as a platform for persistent containers to ease checkpointing and incremental parallel computations. While many OO-databases and object-stores like Texas [12] exist, the organization of VAST as an OO framework made it possible to reuse many parts of VAST without change for environments that VAST was originally not designed for.

7 Conclusions

We have shown that parallel programming with distributed object technology can be combined with shared-memory programming by constructing a software DSM using distributed object systems. A sample implementation demonstrates that three classes of less than 100 lines of code each suffice to implement sequential consistency. In addition, per node multi-threading, distributed synchronization facilities and facilities for weak consistency protocols are provided, which are also comparably small. The description of such compact protocols facilitates the development, maintenance and validation of DSM systems.

References

1. A.Grimshaw. Easy-to-use parallel processing with Mentat. *IEEE Computer*, 26(5), May 1993.
2. H. Assenmacher, T. Breitbach, P. Buhler, V. Hubsch, and R. Schwarz. PANDA - supporting distributed programming in C++. In O. Nierstrasz, editor, *Object-Oriented Programming*, number 707 in LNCS, pages 361–383. Springer, 1993.
3. B. Bershad, M. Zekauskas, and W. Sawdon. The midway distributed shared memory system. In *COMPCON Conference Proceedings*, pages 528–537, 1993.
4. Francois Bordin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. *Scientific Programming*, 2(3), Fall 1993.
5. L. Büttner, J. Nolte, and W. Schröder-Preikschat. ARTS of PEACE – A High-Performance Middleware Layer for Parallel Distributed Computing. *Special Issue of the Journal of Parallel and Distributed Computing on Software Support for Distributed Computing*, 1999.
6. J. B. Carter. Design of the Munin Distributed Shared Memory System. *J. Parallel Distrib. Comput.*, 29(2):219–227, September 1995.
7. K. M. Chandy and C. Kesselman. CC++: A Declarative Concurrent Object-Oriented Programming Notation. In *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.
8. Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: a bridge between release consistency and entry consistency. In *Symposium on Parallel Algorithms and Architectures*, pages 277–287, June 1996.
9. Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. An evaluation of software-based release consistent protocols. *J. Parallel Distrib. Comput.*, 29:126–141, September 1995.
10. K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Systems*, 7(4):321–359, November 1989.
11. W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3.
12. V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An Efficient Portable Persistent Store. In *Proceedings of the 5th International Workshop on Persistent Object Systems*, San Miniato, Italy, September 1992.