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 for high-performance computing. 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. Abstract building blocks can be combined to create a concrete DSM protocol whose components can easily be varied to change the consistency model, the object granularity, the sharing resolution or certain operational modes. A sample implementation demonstrates that three classes of less than 100 lines of code each suffice to implement sequential consistency. In addition, 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.

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 single data item in an array 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 since the idea of shared memory is orthogonal to objects

* Clustered Linux

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 many 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 platform is based on a guest-level implementation of ARTS[5], a distributed object platform for parallel machines and the VAST[21] framework for volatile and non-volatile user-level memory management.

The outline of the paper is as follows: First we describe the basic features of the ARTS platform. Then we present the design and implementation of our integrated platform in detail. Code fragments are presented to show how the integrated platform can be used. Finally, we compare our work to other approaches and provide some concluding remarks.

2 The ARTS Platform

Originally, ARTS was a remote object invocation (ROI) [16] system designed as an integral part of the parallel PEACE operating system family [22]. In PEACE, system services as well as C++ applications are described by means of annotated C++-classes. These classes are pre-processed by a tool called DOG[19] that produces stub routines for remote object invocation and uses any C++ compiler as a back-end. Efficiency and minimal abstractions were major design goals such that even frequently used low-level system services could apply high-level cooperation paradigms instead of crude message passing.

Today, ARTS is an open object-oriented framework, which provides the basic services for distributed and parallel processing within a global object space. Both synchronous and asynchronous remote object invocation as well as collective operations on distributed object groups are supported.

Object groups are implemented as distributed linked object sets. These sets can be addressed as a whole by issuing a multicast operation to a *group leader* representing the group. For example, if the group is organized as a tree the root acts as the group leader and multicast operations issued to the root are executed on each object within the group. In principle, a multicast group can have any user-defined topology such as n-dimensional grids, lists or trees.

2.1 Language-Level Support

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.

All annotations are specified as pseudo comments to retain a strong backward compatibility with standard C++. A `/*!dual!*/` annotation is used to mark classes whose instances need to be accessed remotely:

```
class X {...} /*!dual!*/;
```

Those classes are referred to as *dual* classes. The following statement then creates a new remote object of type `X` at node number 5.

```
/*!dual!*/X a(...) /*!@ Node(5) !*/;
```

Note that the notation `/*!dual!*/X` 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 remote object invocation techniques. Pointer notations and `new()` expressions can also be used to create remote objects.

```
/*!dual!*/X *p = new /*!dual!*/X(...) /*!@ Node(5) !*/;
```

Cloning an existing object of type `/*!dual!*/X` only copies the proxy instance. Thus, the global sharing of objects is possible through passing of proxy objects, e.g., as parameters to remote methods.

Remote method invocation is synchronous by default. An `/*!async!*/` annotation can be applied to mark methods without return values that shall be executed concurrently with the caller.

```
void foo(...) /*!async!*/;
```

Multiple asynchronous methods are potentially buffered. When this behavior is not desired or might cause buffer overflows, a `/*!trigger!*/` annotation can be used instead for flow-controlled one way invocation.

Triggered and asynchronous methods are the basic way in ARTS to initiate parallel computations. However, although this is both simple as well as flexible, asynchronous methods are not very comfortable for high-level parallel programming. Collective operations on distributed object groups are more suitable for this purpose. Object groups are described by group classes that define the topology of a distributed object group. 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

```
class X: public GroupMember{...} /*!dual!*/;
```

it is possible to create and address whole object groups using a `GroupOf` template.

```
GroupOf</*!dual!*/X> a(5);
```

This statement will create a group of five objects that can be collectively addressed by means of collective operations. Collective operations are declared by means of a `/*!multi!*/` annotation.

```
void foo () /*!multi!*/;
```

Parallel global reductions are also possible when a binary reduction method is specified for methods with results:

```
int foo () /*!multi:+!*/;
```

In this example, the results are reduced using the standard addition operator. It is also possible to specify any commutative and associative binary reduction method that is compatible to the return type.

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!*/`). Furthermore, permanent copies of passed parameters on the server's heap can be created using a `/*!copy!*/` annotation. Note that there is no remote call by reference. When call by reference semantics is desired proxy objects have to be passed by value instead.

2.2 Concurrency Control and Synchronization

Remote operations on the same object are executed sequentially and do not interrupt each other by default. However, often a method can only be executed when the state of the object meets certain conditions. A typical example often found in literature is a bounded buffer object that requires to delay writing operations until buffer space becomes available and to continue reading operations when data becomes available. The ARTS platform provides a rich variety of synchronization mechanisms to solve such kind of problems. However, for space considerations, we will only decribe 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 any time it is 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 when everything is completed.

3 Distributed Shared Memory Abstractions

In principle, it is very easy in ARTS to create shared memory abstractions on top of collective operations without any hardware support. For instance, assignments to replicated shared variables can easily be implemented by means of parallel collective write operations issued to a group of replicas. Reading can be done directly in each address space. We can also hide the details of these mechanisms in a template class that encapsulates our shared memory variables. The following declaration then declares an integer variable in a logically shared memory

```
Shared<int> i;
```

that can syntactically be accessed like any local variable. Reading access is efficient but writing can be expected to be very slow when we assume, e.g., sequential consistency semantics for our DSM. We might introduce other classes with relaxed memory semantics such that we can use asynchronous collective operations to write to all replicas. But even in this case, each write operation ends up in at least one message-passing operation. We can hardly tolerate this for fine-grained writing access to arrays and matrices. Consequently, we apply MMU-based hardware support to maintain consistency on a page-based level.

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.

We can also use intermediate separators to avoid false sharing. In the following case, we avoid that accesses to the matrix are not falsely intermixed with accesses to other variables, just because the beginning of the matrix might be located in the same page as the other variables.

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

This might waste some memory space but can improve performance significantly.

Declaring shared variables within shared regions is straightforward but fairly limited in practice because the handling of dynamic data structures is not easy. A heap-based approach like in VAST[21] is better suited for dynamic data-structures. However, we did not implement shared heaps in the first experimental implementation since many numerical shared memory algorithms do not necessarily need dynamic data.

Shared regions are controlled by memory control objects. Each region is associated with a local control object and all control objects form a distributed object group that can be collectively addressed using the collective operations of the ARTS platform. Tasks like global page invalidation (see Fig. 2) or update protocols (see Fig. 1) are, therefore, very easy to implement.

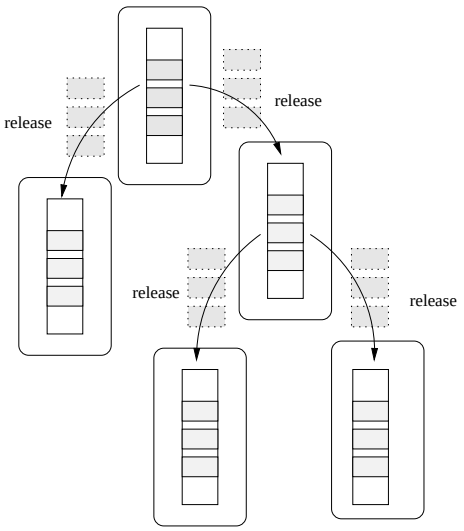

Fig. 1. Update on Release

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[20] 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.

Consequently, each memory control object is usually composed of a segment control object and a consistency engine that implements the desired memory consistency protocol. Additionally, synchronization objects like semaphores or barriers might be associated with the control objects to implement shared memory protocols that associate memory consistency with synchronization points. For example, a simple update protocol might propagate all modified pages to all members of the segment group upon a `release()` operation by means of a collective operation (see Fig. 1). Likewise, an invalidate protocol might globally invalidate all modified pages upon a `release()` and each group member registers the (remote) control object that holds the actual modified page such that the actual version of the page can be fetched remotely when an access occurs (see Fig. 2).

Generally, it is hard to determine if, e.g., an update or an invalidate protocol is more appropriate since this decision depends on application patterns and needs to be specified somehow. Our primary aim was, therefore, to provide a

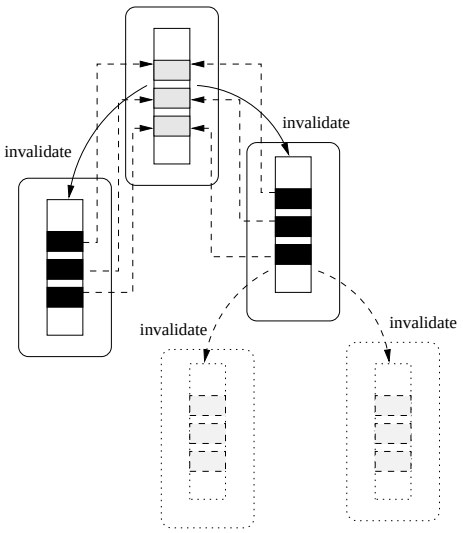

Fig. 2. Invalidate on Release

framework of basic bricks that can be combined easily (e.g., by inheritance) to tailor memory semantics and consistency protocols as desired.

4 Object-Oriented Design of the Clix-DSM

The design of this DSM system was motivated by a number of goals. A component approach was used to separate distinct functionality within the framework. Simplicity of each component combined with extensibility provides the means to apply object-oriented techniques of software design to derive more complex protocols out of base protocols or combine a set of components to a protocol. This approach supports the selection of common attributes of DSM protocols provided as an abstraction (i.e., a building block) to yield a concrete protocol. It also provides a convenient method to replace or switch between certain aspects of a protocol on-the-fly, possibly even during runtime, i.e., between distinct stages of execution.

The properties of a DSM protocol include the following aspects:

The Consistency can be sequential, release, entry or scope, just to name the most popular ones [17, 14, 4, 12].

The Granularity of a DSM object can be at the level of a byte, a word, a page or an arbitrary size of an object. This is the level at which the internal DSM protocols handle objects.

The Sharing Resolution within a DSM object may be at the level of a bit, a byte, a word or a double word. This is the smallest level at which the user may access data consistently and varies only for weak consistency models.

The Directory determines for each DSM object who handles requests for access to the object from the remote node. Typical variants are home-based protocols where requests are handled by a static home node or migration-based systems where requests are forwarded to a dynamically changing node, possibly in a chain of nodes.

The Modification Mode is either update or invalidate, i.e., a remote object reference will either receive the latest values upon local changes or invalidate the remote object, respectively.

The Communication Mode can be either synchronous or asynchronous per memory object, i.e., a node may queue incoming requests while waiting for the acknowledgement of a prior request (of the same memory object) or may already handle the next request meanwhile, respectively [18]. (Distinct memory objects may always be handled independently.)

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 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. Notice that the sharing resolution is irrelevant since the consistency model is not weak, i.e., the user may access data down to a bit in a consistent state. A later example then illuminates the handling of different sharing resolutions.

Sequential consistency is implemented by two classes that are tightly coupled: SequentialObjectControl (SOC) and SequentialConsistency (SC). Excerpts of the signature of these classes (omitting constructors, destructors and initialization) are depicted in Fig. 3 (a) and (b).

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 (see Fig. 3(c)). MS

```
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!*/;
```

(a) Sequential Consistency Class

```
class SeqObjectControl {
 public: ...
  void setConsistency( int objectId,
    /*!copy!*//*!dual!*/ SeqConsistency* pSeqConsistency );
  void setObjectControl( int nodeId,
    /*!copy!*//*!dual!*/ SeqObjectControl* pSeqObjectControl );
  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!*/;
```

(b) Sequential Object Control Class

```
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* pSeqConsistency );
};
```

(c) Memory Segment Class

Fig. 3. Signature of Classes for Sequential Consistency

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 aquireR() or aquireW() for read and write faults, respectively.

The example in Fig. 4 describes the operations for a read fault on node R. The fault handler of the MS (a regular class) synchronously calls aquireR

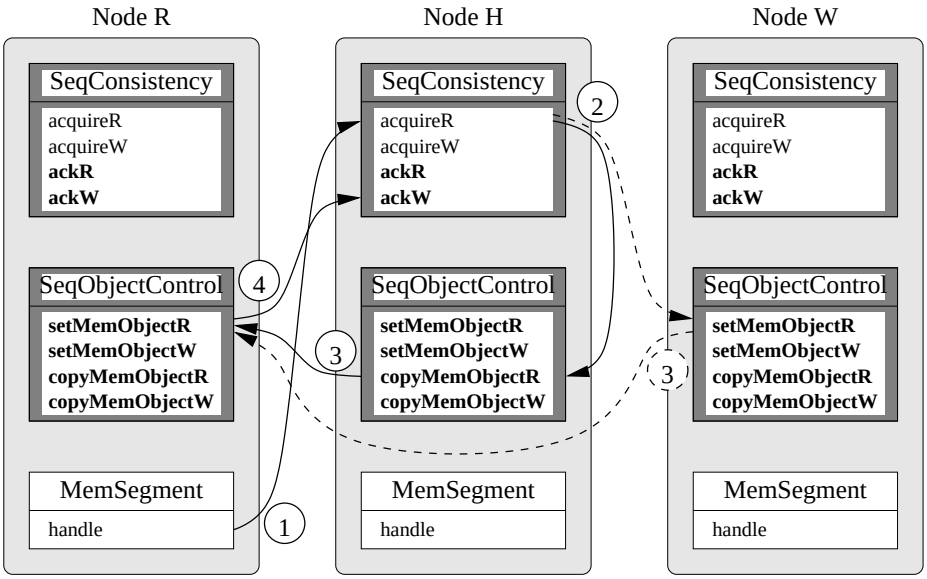

Fig. 4. Read Fault Handling

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 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 aquire() 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 example illustrates the necessity for mutual method invocations between the three players SC, SOC and MS. An instance of SC has to be aware of the SOCs of all nodes, which is realized during initialization through setObjectControl(). An instance of SOC creates its own MS upon construction and initializes it to the proper memory area via initSegment(). It also has to be capable to address the SOCs of other nodes, which are registered by setObjectControl(). The SOC and the MS also have to be aware of the home SC for each page, which is ensured by the setConsistency() method of the respective object.

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 [24] 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. 5) 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 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. The template class also contains a copy constructor and a destructor for correct handling of the ARTS modes for parameter passing during remote object invocations. With the constructor and destructor, a Diff object can be readily passed to a remote node where the diff is then applied to the corresponding page. An example for the

```
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. 5. Diffs with Arbitrary Sharing Resolutions

usage of the Diff class is given in Fig. 6. The class is instantiated with the data type character, which provides a sharing resolution of a byte. Differences between the twin of a page (i.e., a copy pf the unmodified page) and the modified page are recorded in the client and then transfered via remote object invocation to the server. The server then locally applies these changes to its own image of the page.

```
Client: Page twin(page);
        /* ... Aenderungen an page ... */
        Diff<char,Page> d(twin,page);
        for (register int i = 0; i < NUM_NODES; i++)
          dsm[i]->merge(&page,Stream(d.addr(),d.size()));

Server: void SoftDSM::merge(Page* addr, Stream& str) {
          char* buf=new char[str.size()];
          str.get(buf, invoker());
          Diff<char,Page> d(buf,str.size());
          d.merge(*addr);
        }
```

Fig. 6. Diff Usage for Byte Resolution

6 Future Work

Current plans include an extension to home-based scope consistency with the facilities already at hand. Lazy release consistency could also be supported but the current implementation needs to be enhanced by time stamps to support this model.

7 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 [17], were non-object based. The models of release and scope consistency typically do not build on object either [15, 12]. Entry consistency, on the other hand, is often associated with the object-oriented paradigm due to the granularity of the protocol [3, 7]. 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 [11] as well as parallel programming languages such as CC++ [8], ICC++ [10], Mentat [1] and MPC++ [13]. 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. Generally spoken, we might say that ARTS combines much of the flexibility of distributed system approaches with advanced collective operations for parallel computation and global synchronization.

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 [23], Shore [6] or Opal [9] 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. VAST basically provided all the features we needed for the implementation of our DSM system and CLIX is, therefore, an example for the successful integration of unrelated features from different frameworks.

8 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. We described how an object-oriented design approach provides a compact and flexible description of the system components. Abstract building blocks can be combined to create a concrete DSM protocol whose components can easily be varied to change the consistency model, the object granularity, the sharing resolution or certain operational modes. A sample implementation demonstrates that three classes of less than 100 lines of code each suffice to implement sequential consistency. In addition, distributed

synchronization facilities and facilities for weak consistency protocols are provided, which are also comparably small. The brevity of the description is mainly due to the ARTS environment for remote object invocation and the VAST memory manager. 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. N. Bershad and M. Zekausas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. TR CMU-CS-91-170, Carnegie-Mellon Univ., 1991.
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. M.J. Carey, D. J. DeWitt, M. J. Franklin, M. L. McAuliffe N. E. Hall, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring Up Persistent Applications. In *Proceedings of the ACM-SIGMOD Conference on the Management of Data*, Minneaopolis, May 1994.
7. J. B. Carter. Design of the Munin Distributed Shared Memory System. *J. Parallel Distrib. Comput.*, 29(2):219–227, September 1995.
8. 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.
9. J. S. Chase, H. M. Levy, M. J. Freely, and E. D. Lazowska. Sharing and Protection in a Single Address Space Operating System. *ACM Transactions on Computer Systems*, May 1994.
10. A. Chien, U.S. Reddy, J.Plevyak, and J. Dolby. ICC++ – A C++ Dialect for High Performance Parallel Computing. In *Proceedings of the 2nd JSSST International Symposium on Object Technologies for Advanced Software, ISOTAS'96*, Kanazawa, Japan, March 1996. Springer.
11. Object Management Group Document. The Common Object Request Broker: Architecture and Specification 2.0. Technical report, OMG.
12. 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.
13. Yutaka Ishikawa, Atsushi Hori, Mitsuhisa Sato, Motohiko Matsuda, Jörg Nolte, Hiroshi Tezuka, Hiroki Konaka, Munenori Maeda, and Kazuto Kubota. Design and Implementation of Metalevel Architecture in C++ – MPC++ Approach –. In *Reflection '96*, 1996.
14. P. Keleher, A. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *International Symposium on Computer Architecture*, pages 13–21, 1992.

15. 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.
16. Henry M. Levy and Ewan D. Tempero. Modules, Objects, and Distributed Programming: Issues in RPC and Remote Object Invocation. *Software—Practice and Experience*, 21(1):77–90, January 1991.
17. K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Systems*, 7(4):321–359, November 1989.
18. F. Mueller. On the design and implementation of DSM-threads. In *Proc. 1997 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 315–324, June 1997. (invited).
19. J. Nolte and W. Schröder-Preikschat. An Object-Oriented Computing Surface for Distributed Memory Architectures. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume 2, pages 134–143, Maui, Hawaii, January 5–8, 1993. IEEE Computer Society Press.
20. Jörg Nolte. Towards a Persistence Framework for High Performance Computing Systems. Technical Report TR-95007, Real World Computing Partnership, Japan, 1995.
21. Jörg Nolte, Hiroki Konaka, Atsushi Hori, Yutaka Ishikawa, Munenori Maeda, and Takashi Tomokiyo. Ocore and VAST — Integrating Persistence into a Parallel Object-Oriented Language. In *Proceedings of ISOTAS 96*, March 1996.
22. W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3.
23. 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.
24. Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)*, 1996. ANSI/IEEE Std 1003.1, 1995 Edition, including 1003.1c: Amendment 2: Threads Extension [C Language].