The following paper was originally published in the
Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)
Monterey, California, June 1995.

Generic Containers for a Distributed Object Store

Carsten Weich
Institut fur Informatik, Universit at Klagenfurt
Universitatsstr. 65, A-9020 Klagenfurt, Austria

Generic Containers for a Distributed Object Store

Carsten Weich
Institut für Informatik, Universität Klagenfurt
Universitätsstr. 65, A-9020 Klagenfurt, Austria
e-mail: carsten@ifi.uni-klu.ac.at
www: http://www.ifi.uni-klu.ac.at/cgi-bin/staff_home?carsten

May 18, 1995

Abstract

In this paper we report of an experiment about how generic container classes can be used to build up a distributed main memory object store. This approach is inspired by the C++ standard template library (STL) but uses set-like associative structures instead of the array-like sequential structures mainly used by the STL. The basic structure is a container with only fundamental functionality. The only way to access the data of the elements is to apply a function to all the elements. We show how such containers can be extended to indexed sets and distributed object stores.

We use Modula-3 to implement the generic classes. Experiments show that such containers form powerful building blocks especially suitable for implementing distributed containers.

1 Overview

Object stores and object oriented databases usually use the file system as main storage area. Operations on such databases are designed to limit the number of accesses to disc blocks. Sometimes they use large main memory areas as buffer to accelerate operations. In [1] we have presented a different approach: Why not use main memory as main storage area and use the file system only to back up data. This should make it much easier to implement database operations. Since main memory is limited on a single compute node, we make use of distribution and parallelism in order to scale well with a growing database. If you add a compute node to your database you not only get more room for storage but also additional computational power.

We want to use sets as our main container type. Sets are not ordered by definition—this makes it much easier to parallelize operations automatically. The programmer must not make assumptions about the order of performing operations on elements of a set [2]. Consequently the operation can be parallelized by applying it on distributed subsets. Initial observations presented in [3] have shown that with this scheme you can scale up the size of a set without making operations slower by adding compute nodes to the system. The additional communication time needed can be made up by the additional computational power from the new nodes.

The main part of this paper presents generic containers and a generic distributed object type. They will be used to implement sets. We express them in Modula-3 [4]. With generic programming, container structures can be developed independently from the structure of the elements. The standard template library [5, 6] has shown that it is possible to go even further: You can develop algorithms independently of the container structures they work on. This is possible by designing one basic generic type with which container structures can be built. In case of the standard template library this basic type is an abstract notion of a pointer. As we will explain in the second section of the paper, we choose a different approach. Our basic type is less low level, we use whole containers as basic types.

Our containers provide a very simple interface with basic operations, which can be implemented efficiently. You can build up a variety of data structures by subtyping containers and by combining different kinds of containers. Due to the simplicity of the interface, they can be used as object store, as index into objects stores or even as building blocks for more complex, distributed structures. As examples we show how indexed containers and distributed containers can be built using our basic container classes.

We show how *distributable objects* can be made using containers. Distributable objects have an object identifier which is independent of the address space they currently reside. They can be accessed from remote compute nodes, and they can move from one node to another. Again we use generic code to develop such objects.

Finally we show how this approach meets the requirements of our distributed object store architecture. We report about our experiments and give an outlook on the future investigations planned.

Goals

We want to achieve a distributable object store. In order to make parallelization of operations on the stored data simpler, we use sets as storage structures. A large set can be distributed by installing subsets of it on several object store nodes. Many basic operations can be parallelized easily by just applying them on these subsets in a first step. Then the results of the sub-operations (which are hopefully much smaller in size) have to be collected in a second step. Selecting objects that meet certain criteria, calculating sums, maximums or averages and many more fall in that category. So distributable sets are needed as our basic structure.

On the other hand, many clients need to access data according to a certain ordering. This is not possible in sets. So sorted indexes into the sets are needed. They can be used to iterate through the set. Indexes are also needed if a client wants to access the data by means of a key value. Obviously indexes must be distributed as well if they become large.

Let us summarize the required structures: We need set structures for small, large and huge sets. For sets with frequent update accesses as well as for sets which are only readed. And we need sorted index structures for all these. There is no data structure to meet all the needs. We will have to provide a variety of data structures the client can choose from.

We can do this by developing generic building structures, which can be combined to support certain requirements of a particular client. Let us take a look at a well tested library of generic data structures first:

2 The Standard Template Library

The Standard Template Library (STL) [5] was designed to provide generic building blocks for general purpose programs. It also provides set containers. Indexed containers are not directly supported but we could certainly develop generic code to build them.

In order to construct a unified view to every supported data structure, STL defines a set of operations with which all data structures are accessed. Together these operations are called *iterator*. This leads to generic code which not only takes the element type as a parameter. It also parameterizes the whole data structure. Thus, algorithms working on a certain structure will work for a broad range of different structures as well. The operation to switch from one element to the next in the structure must be built in the iterator, not into the algorithm. The iterator must also provide a way to return the element itself (called *dereferencing*). The element must provide a way to compare its value with another element.

STL algorithms see containers as pairs of iterators (*first* and one beyond *last*). They access the structure by iterating through it by means of the predefined operations. By this, STL algorithms see every data structure as an array. Some algorithms are only allowed to iterate through the array by advancing from on element to the next, some algorithms may jump back or forth to the n-th element of the array. It is the job of the implementor of iterators to provide this view. Linked lists, trees or files can be mapped to iterators. A sorting algorithm for example works for arrays as well as for a file of records—the iterators are pretty different, the names and parameters of the operations accessing the data is always the same.

We prefer to consider sets as the most primitive view on aggregations of data. Instead of iterating through the elements, functions are applied to every single element—in an undefined order. By this, direct access to every element is encouraged, sequential processing is discouraged. We hope that this view will lead to parallelizable datastore operations. It is not easy to adopt this view to the STL. Since iterating through the structure sequentially is STL's most basic operation, it is not easy to develop a distributed structure.

But we can make use of the ideas behind STL to develop a special purpose generic library for distributable object stores. We also have to transfer the STL mechanisms to Modula-3 [4], because this is our implementation language. The generic Modula-3 containers presented here have a different view than the STL. We do not distinguish between the abstraction of iterators and containers. By not specifying how containers are processed, we make parallelization much simpler.

3 Generic Containers

Containers are data structures which store elements of a certain kind. You can insert and remove elements. You can apply a function to all elements stored. You can also test whether an element is in the container or not (fig. 1). Some implementations may rely on hints about the maximum or average number of elements the container can hold. You can pass this number when initializing the container. A container can tell the number of elements it currently stores.

Set oriented containers are associative in a sense that they only provide access to already known elements. You can not identify elements by position nor can you scan from one to the next. You can search elements by stating

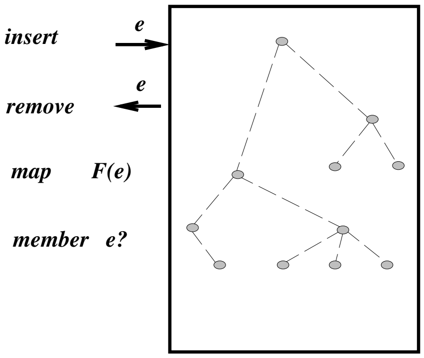

Figure 1: The basic structure

a search function which is applied to *all* elements. If you already have an element, you can test, whether it is in a certain container or not.

We define table containers which store tuples (key, element pairs) as well. If you insert an element into a table container, you also have to provide the elements key value. You later access the elements via this key value.

3.1 The Interfaces

We need two basic interfaces, the associative containers called `Cntr` and the table containers called `TableCntr`. The first one looks like the following[1]

```
GENERIC INTERFACE Cntr(Super, Elm);
  TYPE T <: Public;
    Public = Super.T OBJECT METHODS
        init (sizehint: CARDINAL := 0): T;
        member  (e: Elm.T): BOOLEAN;
        insert  (e: Elm.T);
        remove  (e: Elm.T);
        map     (p: Closure);
        size    (): CARDINAL;
      END;
    Closure = OBJECT METHODS
      apply(e: Elm.T)
    END;
END Cntr.
```

Table containers are defined by the interface:

```
GENERIC INTERFACE TableCntr(Super, Elm, Key);
  TYPE T <: Public;
    Public = Super.T OBJECT METHODS
        init   (sizehint: CARDINAL := 0): T;
        member (k: Key.T): BOOLEAN;
        insert (k: Key.T; e: Elm.T);
        remove (k: Key.T);
        get    (k: Key.T): Elm.T;
        map    (c: Closure);
        size   (): CARDINAL;
      END;
END TableCntr.
```

[1] We do not list the exception handling and other details here.

Both `Cntr` and `TableCntr` are generic interfaces. In Modula-3 this means, that they have to be instantiated with explicit interfaces. The parameters are the super type of the container, the type of the elements and the keys. The module defining the element or key type must export a type `T` and at least two of the following procedures:

```
PROCEDURE Hash (e: T): Word.T;
PROCEDURE Equal (e1, e2: T): BOOLEAN;
PROCEDURE Compare (e1, e2: T): [-1..1];
```

These procedures define how different instances of elements or keys can be compared with each other. `Hash` returns a hash code which can be used to identify elements and to store them in hash tables. The other two are used to compare the value of instances. Which ones are needed in a certain case depends on the implementation of the container. For modules describing types, it is a common convention in the Modula-3 library [7] to provide these procedures.

Containers can be used in very different ways. Implementers might want them to be subtype of `Netobj.T` [8]—to install the container on a remote server—or of `MUTEX`—to be able to lock the container to synchronize multiple accesses to it. So not only the element type is a parameter but also the super type.

Applying functions

The way to access the elements of a container is to apply a function to all of them. The function is passed to the container with the `map` method. This method takes a `Closure` as parameter which contains the function (as `apply`-method). If the function needs parameters or if it produces a return result, the `Closure` must be subtyped. The customized closure contains parameters and return fields as attributes. For instance a function calculating the sum of the `salary`-fields of all `Person.T` elements of a container could be applied using the following closure:

```
SumCl = Closure OBJECT
          sum:= 0;
        OVERRIDES
          apply:= SumOfSal;
        END;
```

This closure can be passed to the `map` method of the container, the `SumOfSal` procedure can access the `sum` field, which will contain the final result after `map` has terminated. The same mechanism is used in connection with threads in Modula-3 [4].

As another example let us define a closure for selecting all `Person.T` objects that have a `salary`-field greater than a certain value:

```
SelectCl = Closure OBJECT
          value : CARDINAL;
          result: PersonCntr.T;
        OVERRIDES
          apply:= SelectPersons;
        END;
```

The value is passed to the `SelectPersons` procedure with the `value` field. The procedure will test each element individually and insert those meeting the condition into the `result` container.

3.2 Different kinds of containers

The exact specification of the container operations are left to the implementor of the generic interfaces. E.g. we might have containers allowing the insertion of duplicates (like in bags) and others, which will ignore such insertions (like in sets). For simultaneous access to containers by different clients: Some implementations might require the client to lock the whole container before any update action, others might lock individual elements.

Very often it is necessary to store the data in a sorted fashion. For this purpose we define subtypes of the basic container classes. The sorted associative container class looks like the following, the same technique can be used for table containers:

```
GENERIC INTERFACE OrdCntr (Cntr, Elm);
  TYPE
    Direction = {Left, Right};
    T <: Public;
    Public = Cntr.T OBJECT METHODS
        init (sizehint: CARDINAL := 0): T;
        iterate (first: Elm.T;
             dir := Direction.Right): Iterator;
      END;
    Iterator = OBJECT METHODS
        next (VAR e: Elm.T): BOOLEAN;
      END;
END OrdCntr.
```

We do not want to change the semantic of the `map` method: The order in which it is applied to the elements is still not defined. To access the elements sequentially, we provide an additional `iterator` method. It returns a cursor object which can be used to scan through the elements of the container in the order defined by the `Elm.Compare` function. Note that code written for basic containers can still be used for ordered containers, because the latter are subtypes of the first.

We will use subtyping for a number of other purposes as we will see in the following.

3.3 Subtypes of the Container Classes

The operations performed by containers are rather fundamental. There is only the `map` method to access the elements, e.g. for tasks like searching or working on subsets. Algorithms which are of interest for more than one client are implemented as generic modules which define subtypes of some container type. Obviously this means that they do not have to be reimplemented for whatever container they should work on. As first example let us look on set operations. A generic module containing the common set operations could have an interface like the following:

```
GENERIC INTERFACE Set(Container, Elm);
  TYPE  T <: Public;
    Public = Container.T OBJECT METHODS
        init    (): T;
        select  (c: SelectClosure): T;
        union   (s: T): T;
        intersect (s: T): T;
        equal     (s: T): BOOLEAN;
      END;
    SelectClosure = OBJECT METHODS
        test  (e: Elm.T): BOOLEAN;
      END;
END Set.
```

The interface is called `Set`, but if the super type is a container that allows multiple insertions it will act like a bag. The key point is that the (generic) implementation of the `Set` module uses only operations of the container interface.

The method `select` returns a new instance of the same set type container, which contains all elements that have met a certain criteria. The criteria is tested by a `test` method passed to `select` as parameter. The methods `union` and `intersect` can be used to generate new sets containing the union or the intersection between the set itself and another set passed as parameter to the methods. Finally `equal` can be used to test whether the set contains the same elements as another one passed as parameter. All these can be implemented using the `map` method defined in the super type. If you want to print out all elements of the set that meet a certain condition, you write:

```
set.select(cond).map(print)
```

Where `set` is an instance of type `Set.T`, `cond` provides a `test` method checking the condition and `print` provides the method that prints individual elements.

3.4 Rearranging containers

If you need alternative access to some container data, or if you want to rearrange the data it contains, all you have to do is to copy it. E.g. sorting a container means to copy it to an ordered container. A trivial generic module will do this job:

```
GENERIC INTERFACE CntnrCopy (Cntnr1, Cntnr2);
  PROCEDURE Copy(c1: Cntnr1.T): Cntnr2.T
END CntnrCopy.
```

Its generic implementation only needs the `map` method of the source container and the `insert` method of the destination.

4 Putting containers together

Now we are able to show the efficiency of our containers. Due to the simplicity of their interfaces, it is rather easy to use them as building blocks of more complicated structures (which might them self be containers). We will present two examples: Containers with indexes attached to them and distributed containers. Both use containers again to implement their features. So it is possible to control their behavior, efficiency and complexity by passing suitable container instances as parts when instantiating the final structures. For instance you can pass a distributed container as index structure to the generic indexed container—which will lead to a container with a distributed index without one single additional line of code.

4.1 Indexed Containers

One of the most important structures in large data stores are indexes. They provide alternative access to large amounts of data to accelerate certain repeatedly needed retrievals. Suppose we have a huge container containing all student data records of an university. If a certain algorithm needs access to all students attending a course we need an index pointing to only that data. Another need might be to access the student data by their name and alternatively by their "matriculation number".

Using generic containers this is a very simple task. We first define a subtype of a generic container class containing additional methods to attach and detach indexes to it. Note that it is left to the instantiation whether the super type is a ordered container or not (see 3.2).

```
GENERIC INTERFACE IndexedCntr
                  (Cntr, Index, Elm, Key);
  TYPE
    GetKey = PROCEDURE (e: Elm.T): Key.T;
    Test   = PROCEDURE (e: Elm.T): BOOLEAN;
    T <: Public;
    Public = Cntr.T OBJECT METHODS
        init (sizehint: CARDINAL:= 0): T;
        addIndex    (index: Index.T;
                     getKey: GetKey;
                     test: Test:= NIL);
        removeIndex(index: Index.T);
      END;
END IndexedCntr.
```

The method `addIndex` attaches a new index structure to the container. `Index.T` is a table container mapping `Key.T` values to `Elm.T` instances (which are the elements of the indexed container). You have to provide a `getKey` procedure which calculates the key value of a particular element. As an option you can also pass a `test` procedure which decides whether an element should be contained in the index or not.

So if we want a structure containing all students attending a course, we do the following: We provide a container for that structure, pass it to the indexed student set using the `addIndex` method together with a `test` procedure (which selects the corresponding students). This will generate the index and the indexed container will keep it up to date until we detach it with the `removeIndex` method.

You can add more than one index to an indexed container. Still all have to have the same key type. If this is not convenient, you can subtype an indexed container again with the same generic interface but a different key type parameter. The subtype will have two `addIndex` methods, one for each key type.

At this point we list a fragment of the generic code of the indexed container. It is the implementation of the `insert` method.

```
PROCEDURE Insert (self: T; e: Elm.T) =
  BEGIN
    Cntr.T.insert(self, e);
    VAR ind:= self.indices;
    BEGIN
      WHILE ind # NIL DO
        IF ind.test = NIL OR ind.test(e) THEN
          ind.index.insert(ind.getKey(e), e)
        END; (*IF*)
        ind:= ind.next
      END (*WHILE*)
    END
  END Insert;
```

The statement `Cntr.T.insert(self, e)` is a super call to insert the element in the container itself, the line `ind.index.insert(...)` updates the index. This is all we have to do. The complete generic implementation of indexed containers counts little more than 100 lines of code.

4.2 B-Trees

B-trees and B*-trees were developed to minimize access to disc blocks when searching data identified by a key. While balanced binary trees perform very well in memory, it makes sense to store more than one key in a tree node if accessing the node is expensive [9]. The same is true if the data is distributed among several compute nodes (see fig. 2). Since the amount of data stored in a compute node will be much larger than in a disc block, we need a special kind of B-tree: The root tree node is much smaller than the tree nodes in the machines. The nodes containing the data can be huge. Looking up elements on a single node must also be efficient.

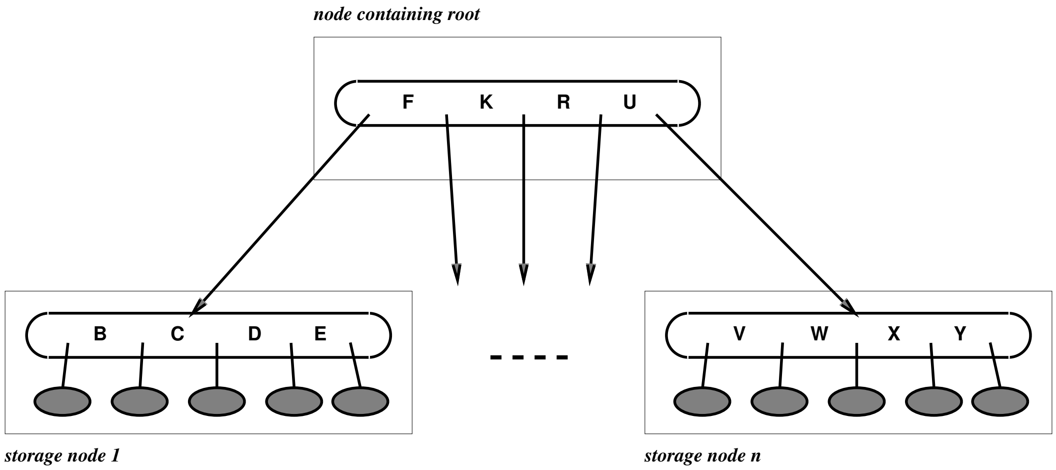

Figure 2: A distributed B*-tree object store

Again we try to develop such a structure using containers as building blocks. We need three kinds of containers:

- *Node containers*
  They store the elements of the distributed container. They must be able to split themselves: This generates a new node if necessary, when the amount of data increases. They also have to be able to merge themselves with another node. If the amount of data decreases, several nodes can be combined in this way.

  The split and merge methods can be implemented using the `map` method of the generic containers. So any container can be transformed into a node container.

- *Root node*
  This is a table mapping elements to node containers. It should be a sorted table in which the smallest element (according to some element ordering) stored in a node is mapped to a pointer to the node containing it.

  The root node can be implemented with our basic table containers.

- *Superstructure*
  We call the structure containing the root table container-container. It is a subtype of some container class with additional methods (see below).

  The implementation of a container-container is responsible for the distribution strategy. The client can choose between different implementations to get containers which distribute the elements over equally sized substructures, or to get one which replicates the data to accelerate certain client operations, and so on.

Let us take a look at the container-container interface:

```
GENERIC INTERFACE CntrCntr (Cntr, Elm);
  TYPE
    T <: Public;
    Public = Cntr.T OBJECT
      METHODS
        init(maxPerNode, minPerNode,
             sizehint: CARDINAL:= 0): T;
        map (fct: Closure;
             coll: CollectClosure:= NIL);
      END;
    CollectClosure = OBJECT METHODS
        collect (cls: ARRAY OF Closure);
      END;
END CntrCntr.
```

We have to redefine the `init` method in order to pass additional initialization information to the distributed container: How many nodes will be available? At what size does the client want to split a node into two? What is the minimum size of a node, i.e. when does the client want a node to be merged with others?

The `map` method needs a second closure: The first one defines the function which has to be applied to all elements (in all sub nodes). Since we want to do this in parallel, we get several provisional results. These can be computed to a final result with the `collect` method of the second closure. It can be left nil, if there is no such result.

As we have seen it is possible to build a variety of storage structures with the container interface (and with little changes to it). A formal description of a certain instance of a container must be supplied by the implementor. It must define the operations exactly and describe their complexity. A system of generic implementations of various kinds of containers is being developed at our department.

5 Generic Distributable Objects

The generic containers support access to arbitrary elements. They do not address the problems of accessing fields and methods of remote stored objects. Especially they do not solve the problems of simultaneous access to the data of a particular element. In this section we demonstrate how access to remote objects can be organized using containers. The task of looking up the location of a remote object and retrieve its data is very similar to the task of retrieving an element from a container. If we could express the access to remote objects using container mechanisms, we could make use of distributed containers to implement remote data access.

5.1 Dereferencing with Methods

In Modula-3 objects are identified by their reference, which points to their memory address. This is not suitable for objects which can reside on several compute nodes. Especially path expressions are difficult to implement if the location of the data is not known. Think of a person record storing the car the person owns. The car record might have a reference to the manufacturer of the car, which is a record containing the name of the company. Determining the car builder of the person's car would require an expression like the following:

```
person.car.manufacture.name
```

The person record, the car record and the manufacturer record might reside all in different address spaces. Further on, an assignment like

```
person.car := bmw_525;
```

which is called when the person buys a new car, again might require write access to several storage nodes. This is not possible if the objects above are ordinary Modula-3 objects.

Before we can access the object's data, we have to insure that we have an up-to-date version of it. This could be done by calling a method before reading data. The above path expression should look like:

```
person.r().car.r().manufacture.r().name
```

The `r` method checks if the data is locally available. If not it has to retrieve it. Finally it returns the actual value of the data. Note that changing this value should not change the actual value of the object.

The expression for setting the persons car attribute should look like:

```
person.w().car := bmw_525.r();
```

The effect is that the `w` method requests write access to the data object. It returns a reference of the data's physical location. This reference points to a writable location of the data which now can be changed.

The Modula-3 library contains the so called *network objects* [8], which provide the possibility to access remote objects. Network objects can be used to implement containers on remote nodes. But they were mainly designed to provide access to remote *services*. It is not possible to read or write object fields directly, you can only call the object's methods. And then, network objects were not made to support hundreds of thousands of objects, which might change their location frequently.

So we propose a different scheme. We need a construct which can be used instead of object references. The following interface describes a distributable object pointer:

```
GENERIC INTERFACE DistObj (Container, Elm);
  IMPORT Word;
  TYPE T <: Public;
     Public = MUTEX OBJECT METHODS
       init(c: Container.T; e: Elm.T): T;
       copyref(): T;
       r(): Elm.T;
       w(): Elm.T;
     END;
  PROCEDURE Equal(o1, o2: T): BOOLEAN;
  PROCEDURE Compare(o1, o2: T): [-1..1];
  PROCEDURE Hash(o): Word.T;
END DistObj.
```

Obviously this interface is suitable as element type for our containers—it contains a `T` and the necessary comparing procedures. This type can be used instead of a main memory address pointer. Dereferencing a distributable object can be done with two methods: `r` dereferences for reading, `w` dereferences for writing. Obviously, read and read/write access can be granted to clients by showing all or only part of the distributable object's interface. The `r` method returns the actual *value* of the object. The `w` method returns a pointer to the a writable location of the object. If the contents of this location is changed, the actual value of the object changes.

When initializing a distributable object, we pass the objects data and a container to the `init` method. The container serves as an abstract representation of the physical location of the object.

The `copyref` method returns a pointer to the object, not the object's value. This representation is of the same type and points to the same instance than the original. It is not necessary to look up the current value of the object. The method acts like assignment of pointer values. It is not necessary in a single address space. But the representation returned by `copyref` can be sent to another node.

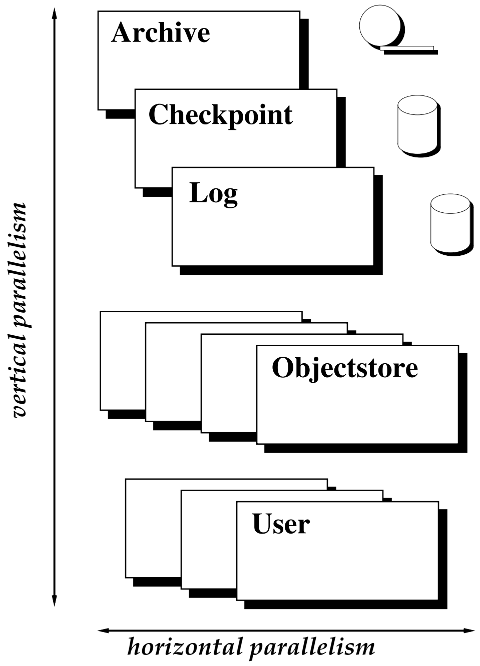

Figure 3: The components of PPOST

5.2 Implementing Distributable Objects

In a distributed object store with simultaneous access of several clients, we have to deal with the following problems.

1. We have to get an actual version of the data when the `r`- or `w`-method is called,

2. We have to deal with locking and consistency problems if simultaneous read and write accesses occur. If there are several copies of an objects values around, we have to ensure consitency after every write access.

The physical location of the objects data is represented by a container. You pass the container which can be used to retrieve the data when initializing an instance of a distributable object. The code implementing the `DistObj` interface delegates the problem of physically distributing the data to this container. It concentrates on implementing the locking scheme and on keeping consistency.

By using containers as abstraction for physical locations, we are able to separate the two main problems of remote object access. Keeping consistency is not a problem which a container can solve. But it can solve the problem of retrieving the elements. Different locking strategies can thus be combined with different storing solutions.

6 Containers for PPOST

The architecture of PPOST was presented in [1, 3]. In this section we would like to show, that the presented generic container scheme can be used to implement such an architecture. PPOST's main components are (figure 3): *object store* (consisting of a number of object *storage machines*), *log machine*, *checkpoint machine*, *archive machine* and *users* (consisting of a number of user machines). All the data of the stored objects (i. e. their attributes and methods) lie in the memory of the storage machines. PPOST is transaction-oriented. Every transaction that reads or changes the data is executed on those machines. Transactions are initiated by the user machines and processed by the object store. Changes of the data in the object store are reported to the log machine which saves the information onto a log file in non volatile memory. This can be done by for instance by writing to a sequential file with maximum disc speed.

The checkpoint machine reads the log produced by the log machine and saves all committed changes to the disc-based database. It produces a structured image of the database. If this requires more time, only the log file on the log machine will grow. The user transaction can go on as soon as the information about the changes is transmitted to the log machine.

The archive machine saves the disc-database to a secondary storage, like a magnetic tape. This is considered as a normal activity of the data-store and again is done in background without interrupting the user-transactions.

We call this pipeline-like way to decouple user-transactions from issues of persistence *vertical parallelism*. Operations on the stored data can often also be done in parallel, we call this *horizontal parallelism*.

With our container templates we are able to offer several structures to organize the object store. The users will get containers as super structures. With our distributable objects we have explicit control over all write accesses to the data: All users have to use the `w` method described in the last section. This method has all the necessary information to produce the log records.

7 Results and Future Work

Generic programming turned out to be an efficient technique to implement many variants of object stores. Once you have decided to view your data in an uniform way, the solutions you find for a particular problem can always be reused for many more problems. Since the parts which are developed all have similar interfaces, they can be tested easily and partly automatically. Thus more complicated structures are made with robust building blocks. Modula-3 proofed to be suitable for this programming philosophy.

The technique is especially powerful when building distributed object stores. It makes it easy to switch between different internal structures say for node subsets or indexes. Since the result of the combination of sub containers again forms a container, the technique can be applied recursively. Once we have a distributed container, we also have distributed sets, indexes or even nodes to super-super structures.

Because of this results, we are currently working on library of different generic containers (it is available via links from the authors WWW home page). They seem to provide a very flexible framework for experimenting with different distribution strategies.

8 Related Material

This paper, the Modula-3 code of the generic containers and the PPOST papers are available via world wide web. They can be accessed via links from the authors home page (the www-address is listed at the beginning of the paper).

References

[1] L. Böszörmenyi, J. Eder, and C. Weich. Ppost, a parallel database in main memory. In *Proceedings of the Fifth International Conference on Database and Expert Systems Applications*, 1994.

[2] L. Böszörmenyi and K. H. Eder. Adding parallel and persistent sets to modula-3. In *Proceedings of the Joint Modular Languages Conference*, September 1994.

[3] L. Böszörmenyi, K. H. Eder, and C. Weich. Ppost, a persistent parallel object store. In *Proceedings of the Second International Conference Massively Parallel Processing Applications and Development*, 1994.

[4] Greg Nelson. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[5] A. A. Stepanov and M. Lee. The standard template library. Doc no: X3j16/94-0095, wg21/n0482, ISO Programming Language C++ Project, 1994.

[6] D. R. Musser and A. A. Stepanov. Algorithm-oriented generic libraries. *Software—Practice and Experience*, 24(7):623–642, July 1994.

[7] Jim Horning, Bill Kalsow, Paul McJones, and Greg Nelson. Some useful modula-3 interfaces. Research report 113, Digital Systems Research Center, Palo Alto, 1994.

[8] A. Birell, G. Nelson, S. Owicki, and E. Wobber. Network objects. Research report 115, Digital Systems Research Center, Palo Alto, 1994.

[9] R. Sedgewick. *Algorithms in Modula-3*. Addison-Wesley, 1993.