# Cooperative Data Sharing: A Portable Scalable Runtime

David C. DiNucci (dave@elepar.com)
Elepar (www.elepar.com)
Beaverton, OR 97006

*Cooperative Data Sharing (CDS) is a simple but expressive hybrid of DSM and message passing which permits the application to dictate policy, leaving the runtime to flexibly choose between efficient mechanisms. The logical semantics, potential physical implementation approches, usage hints, and a few performance characteristics are presented.*

## Motivation and History

Message passing paradigms have significant advantages when entities communicate over a high-latency channel. The program can explicitly manage the channel's use, latency can be often be hidden by moving the data toward its next user even before it is requested there, per datum overhead can be minimized (pipelined) by moving the data in large chunks, and tight synchronization can be avoided by queuing unexpected data at the receiver. In a low-latency environment, a shared memory paradigm helps preserve memory bandwidth and reduce overhead by minizing copying. Shared memory also frees the program from explicitly managing data movement, and facilitates a demand-driven style in irregular cases, where the next entity that will need data is not known until that entity needs the data.

The amount of latency that is tolerable for a shared memory model is relative to many factors, and is only worsened by the increasingly large gap between memory access frequency and latency. To help, shared memory, and especially DSM, integrate message-like features between computing entities and/or caches, such as "home processor" declarations for some or all memory addresses (e.g. where they will be accessed most often), explicit prefetch (to hide latency), and page-based trannsfers (to pipeline the overhead). Still, these solutions are largely heuristic, rarely manage channel latency as effectively as message passing, and can lead to additional problems like false sharing. Implementing DSM systems may impose overheads from use of memory management hardware and "diffing" operations.

Cooperative Data Sharing (CDS) takes a different approach. Instead of starting with either message passing or shared memory, its design was motivated by looking at the advantages of each (like those above) in different environments and working toward a common interface with all of the advantages of both. The result is a small set of more general primitive communication operations which can look (and perform) like message passing, shared memory, or a hybrid. As shown in table 1, CDS's expressiveness serves to convey specific knowledge from the source program to the runtime system, where it can be combined with current hardware configuration and status to perform the communication most efficiently under those circumstances.

| Features | CDS | DSM | MPI | SOCK | LINDA |
|---|---|---|---|---|---|
| Some data can be traded/ shared in place (true 0 copy!) | x | x | | | |
| Consumer can pull (get) data from passive producer | x | x | 2 | | x |
| Consumer can prefetch/ prepull data to hide latency | x | ? | 2 | | |
| Producer can push (send) data to passive consumer | x | | 1 | x | ? |
| Data can be queued at producer waiting for pull | x | | 1 | x | ? |
| Pushed data can be made to overwrite previous value | x | x | | | x |
| Producer can retain access rights to communicated data | x | | 1 | | x |
| Producer can relinquish access rights to communicated data | x | x | 1 | | x |
| Dynamic memory allocation for shared memory | x | ? | | | |
| Consumer can specify timeout for waiting | x | ? | | | |
| Supports heterogeneous data formats | x | | 1 | | |
| Allows multiple readers | x | x | | | |
| Simplicity (~number of function + macro interfaces) | 42 | tens ? | !!! | 13 | 5 |

Target platforms differ not only in communication architecture, but also in the number of processors. If software entities running on the same processor can exchange data with very little overhead, they can be viewed very much like subroutines or objects in a single entity, rather than as separate entities. In other words, a "portable thread" approach like CDS allows the granularity of the program to effectively adapt (i.e. decrease) to the platform topology.

The ideas here originated during consideration of a runtime system based on LGDF2 [1] at OGI c. 1989. At

NASA Ames, they were implemented as Message Passing Kernel 1 (MPK1), a more architecture-independent alternative to MPI-1 (then being developed) c. 1993. It was later renamed CDS1 (for accuracy, and to avoid confusion with other MPKs) [2], and was subsequently proposed as a form of one-sided communication for MPI-2. The CDS described here is now produced by Elepar as BCR [3], and has greater heterogeneous support, more mature implementation, and more multi-paradigm interfaces than its predecessors.

CDS encompasses features found in the Reactive Kernel (RK) [7], which predated it, and most found in Treadmarks[5], CVM [6], and CRL [4], developed contemporaenously..

## The CDS Approach

CDS accomplishes its seemingly-conflicting goals by providing a fairly simple set of programming semantics, the logical model, which has an efficient implementation on a wide variety of target architectures, the physical model. This is similar, in some ways, to the approach used by high-level (compiled or interpreted) languages, but the call-based interface used here restricts both the expressiveness and the amount of preprocessing available to achieve these goals. This section will present the abstract semantics, or the logical model, of CDS. Approaches to efficient implementation on various architectures, or the physical model, will be described in the next section.

Each CDS computing entity is similar in some ways to a thread, sharing information with other entities, and in some ways similar to a process, independent enough to run on a remote processor, so here it will just be called a *CCE* (for "CDS computing entity"). Each CCE possesses a logically-private *communication* (or *comm*) *heap*, which is just a special heap of memory that CDS knows about (figure 1). A CCE allocates a memory region from this heap by using the **`rgalloc`** operation, which is analogous to the standard C **`malloc`** routine (e.g. it takes a length in bytes), but instead of returning the region's address like **`malloc`**, **`rgalloc`** returns a CDS *region ID*, which can be thought of (by the user) as a pointer to the region's address—i.e. the data can be accessed by using "`**rgid`" instead of just "`*rgptr`" (as one might use with traditional **`malloc`**). There is also an **`rgfree`** operation in CDS that is analogous to **`free`** in standard C, that effectively declares that the CCE will no longer access that region. Additional operations exist to determine the length of an already-allocated region (**`rglen`**), and to try to change a region's length without moving it (**`rgrealloc`**).

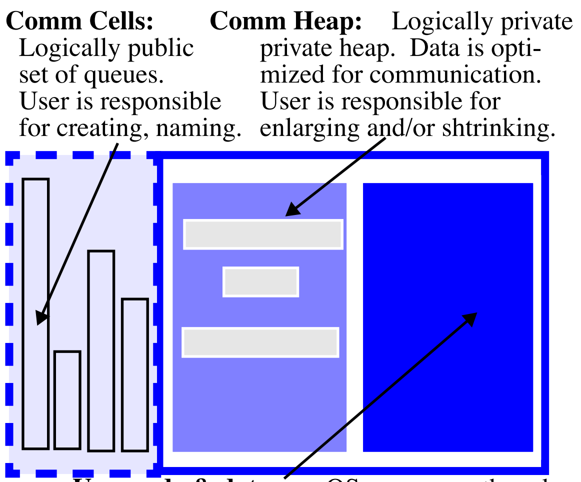
Comm Cells: Logically public set of queues. User is responsible for creating, naming.
Comm Heap: Logically private private heap. Data is optimized for communication. User is responsible for enlarging and/or shtrinking.
User code & data: OS process or thread

**Figure 1. Anatomy of a CCE**

Along with a comm heap, each CCE can also have any number of *comm cells* (usually just referred to as "cells") which are visible to other CCEs. Each cell has an integer name (unique within that CCE), so any cell can be addressed by any CCE by naming the CCE that holds it and the cell number within that CCE. Each cell can hold any number of distinct regions, in a queue-like fashion. There are four basic communication primitives that operate on cells (figure 2):

1. **`zap`**, which takes a cell address (i.e. CCE name plus integer cell name), and deletes any regions which might be sitting in it.
2. **`put`**, which takes a region ID and a cell address, and produces a copy of that region at the tail of the cell, optionally **`zap`**ping the cell first, and optionally **`rgfree`**ing the region after. This is also known as **`write`** if the **`zap`** is performed, **`enq`** if not.
3. **`putm`**, which is the same as **`put`**, but takes a list of cell addresses and produces a copy into each of them.
4. **`get`**, which takes a cell address, and produces a copy of the first region from the cell into caller's comm heap (and optionally removes it from the cell), return-

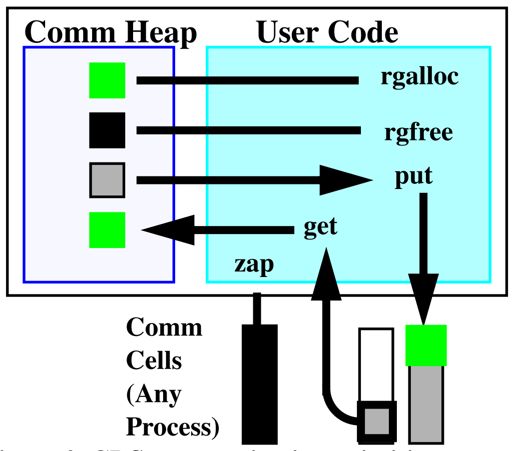

**Figure 2. CDS communication primitives**

ing a region ID to the new comm-heap region. A timeout value (in microsecs) can also be provided—or specified as PENDING, in which case a "pending" region ID will always be returned immediately, but must subsequently be verified before use (or canceled) using **`rgwait`** (also with timeout). **`get`** is also known as **`deq`** if it is removed, **`read`** if not.

So, CCE's in CDS communicate by creating regions in their comm heaps using **`rgalloc`**, shuffling those regions between their comm heaps (via cells) using **`put`** and **`get`**, and eventually relinquishing them using **`rgfree`**. There is one more basic rule covering communication: Before changing data in a region that was obtained with a **`get`**, or one which has already been **`put`** somewhere since calling **`rgalloc`** (or **`rgmod`**), then **`rgmod`** must be called (with the region ID) first. **`rgmod`** may update the region pointer stored in the region ID (i.e. in `*rgid)`, so care must be taken not to save an old value of this pointer into a temp variable across a call to **`rgmod`**.

A CCE declares itself as a CCE (allowing it to call CDS routines) by calling a special CDS **`init`** routine, but this will block until or unless the CCE has been *enlisted*. It can be enlisted by another CCE (calling the **`enlist`** routine), or by the user directly (using the **`enl`** program). If **`enlist`** (or **`enl`**) cannot find an appropriate CCE blocked at **`init`**, it may try to create one, so enlisting a CCE will often serve both to create it, and then to unblock the **`init`** when it is reached. This allows one programming approach to be used, regardless of whether the target platform requires all processes to be started at once (in which case they will all block at **`init`** until enlisted) or allows/requires them to be created piecemeal. Each CCE is identified with an integer CCE ID, which is similar to a file descriptor in that the specific integer used to identify a CCE may depend not only on the CCE being identified, but on the CCE holding that identification. After a CCE has called **`init`**, it can find its own ID, and the ID of the CCE which enlisted it. The only other way to find the ID of a CCE is to receive it through explicit communication—e.g. **`enlist`** does *not* return the ID of the CCE enlisted. The **`init`** routine itself creates a small comm heap and one comm cell (named 0), and **`enlist`** optionally takes a short region (as an argument) to be **`put`** into cell 0 of the CCE being enlisted. The new CCE must explicitly create any further cells or larger comm heap if/when needed, by calling **`cagrow`** (vaguely like **`sbrk`**).

Sometimes, the data to be moved from one place to another will reside (or need to reside) somewhere in the CCE's program memory instead of in the comm heap. It is usually possible to just copy the data to/from the comm heap in traditional ways (e.g. using the pointer in the region ID, and perhaps **`memcpy`**), but CDS provides three more flexible routines to help; one to copy data *to* a region from other memory (called **`copyto`**), one to copy *from* a region to other memory (called **`copyfm`**), and one to copy data between regions (called **`copytofm`**). All of these routines take a set of instructions called a *copy descriptor* (or sometimes likened to a "datatype"), expressed as a specially formatted integer vector, that allows data of different types and relative positions to be copied and packed or unpacked. Another routine, **`copytosz`**, can be used to find out how much space a **`copyto`** will use in the destination. For those somewhat-common cases when one needs to **`rgalloc`** a region (after first determining the size needed with **`copytosz`**), **`copyto`** data to it, and then **`put`** it into a cell, CDS offers a single composite routine called **`send`** which does all of these. Similarly, there is a **`recv`** routine that does the same as a **`get`**, a **`copyfm`**, and an **`rgfree`**. These conform roughly to send and receive in message passing paradigms. Both **`send`** and **`recv`** are allowed to succeed even if there is not really sufficient room for the region within the comm heap.

Use of the copy routines is mandatory when moving CCE IDs to/from the comm heap, to translate them between the CCE-dependent integer representation and a CCE-independent one. The copy routines are also extremely helpful when moving data between heterogeneous processors, because CDS endows each region with an integer code (called an *archtype*) that tells how that data in the region is represented—i.e. the kinds of processors that represent their data that way. This code gets passed along with the region from one CCE to the next. If the CCE is running on a processor type that matches the archtype in a region, then that CCE can safely access the region data directly (using the `**rgid` approach). Otherwise, the data should probably only be accessed using the copy routines, which translate appropriately based on the archtype. The archtype for a region is set when it is **`rgalloc`**'d—i.e. the archtype defaults to the same as the processor creating the region, though this can be overridden by a second argument to **`rgalloc`**. The archtype for a particular region can be queried with the **`rglen`** routine. The archtype for the current CCE is available in a global variable, **`archtype`**, but the archtype corresponding to any CCE ID can also be determined with the **`arch`** routine (e.g. to be supplied to **`rgalloc`**).

To make CDS more natural in shared memory situations, a set of operations (often implemented as macros) is provided. They are based on the principle that **`put`**ting a region into a cell makes it available to other CCEs, which is just what releasing a lock does in a traditional shared memory environment, and likewise, **`get`**ting a region from a cell provides the taker with access to it, just as acquiring a lock does. So, there's an "acquire read lock" (**`acqrl`**) operation which is really just another name for **`read`**, an "acquire write lock" (**`acqwl`**) operation which is really just a **`deq`** followed by an **`rgmod`**, a "release read lock" (**`rlsrl`**) operation which is really just **`rgfree`**, and a "release write lock" (**`rlswl`**) operation which really just **`put`**s the region ID back into a cell while **`rgfree`**ing it. An additional operation to convert a write lock to a read lock (**`wl2rl`**) is the same as atomically releasing a write lock and acquiring a read lock, in this case accomplished by just **`put`**ting the region ID into the cell without **`rgfree`**ing it. Note that a write lock will block readers and writers, but a read lock will block neither: If a writer comes along, it will write to a different "version" than the readers are reading.

Finally, each cell can be endowed with a handler (i.e. user-supplied subroutine) using the **`handler`** routine, to be requested whenever the number of regions in the cell exceeds a specified high- or low-water mark before a **`get`** or after a **`put`**. Even if it is requested, the handler will only invoked ("upcalled") if the currently executing coroutine (i.e. handler or main CCE) notices and has a lower priority than the handler. (At this writing, it will only notice if an operation blocks, but this is likely to change in future versions.) The priority of a coroutine can be set using the **`priority`** routine, and can be temporarily adjusted down using the **`allow`** routine.

## Physical Model

The previous section conveys very few of CDS's advantages over shared memory in a low-latency environment. In fact, from that description, CDS appears to require even more copying (and therefore more overhead) than message passing. The *physical* model underneath is (or at least, can be) quite different, and is described here. This description can serve as the basis for a performance model, but in the end, any implementation which conforms to the simpler logical semantics already described would be considered valid.

Four optimizations rely on these differences between the physical and logical models:

1. In the logical model, each comm heap is private to its CCE. In the physical model, each comm heap is stored in shared memory, where it can be accessed directly from any CCE that the hardware allows (all such CCEs will be called a "CCE group" here), or from a special daemon process that helps access that memory for CCE's outside of the group.

2. The comm cells don't really hold regions that have been copied from the comm heap, they hold pointers to regions that are still sitting in the comm heap.

3. Related to 2, data is rarely copied from place to place. Instead of copying a region to or from a cell belonging to the same CCE group, a pointer to the region is just moved to or from the cell. Reference counts are kept with each region to keep track of how many such pointers point to it. Data is only actually copied in three cases: when accessing a cell in a different group (so the data needs to be moved); when modifying (i.e. calling **`rgmod`** for) a region targeted by other pointers (as indicated by the internal reference count), and when the copy routines are called explicitly. (The second case here is effectively implementing "copy on write" without use of memory management hardware.)

4. The **`send`** and **`recv`** operations can be optimized. For example, they don't really need to go through the comm heap at all in some cases: The region that they purport to use is never visible to the caller, so the data can sometimes be moved directly between the communication channel and the CCE's memory addresses.

As a result of these, CDS has many similarities to shared memory. Different processing entities communicate through some common memory area that they can all access, and are careful when accessing that area to ensure that they cooperate properly, and do not interfere with each others' intentions. The shared area is partitioned into regions which are managed separately—i.e. a computing entity obtains access to an entire region at one time, uses it as desired, and relinquishes it. Multiple entities can read the same region at the same time.

CDS also has similarities to message passing. A computing entity (the sender) specifies some data to move and the computing entity (the receiver) where it should moved to (because it will likely be used there). The sender and receiver use a common tag to ensure that they are both working on the same data. The data can be queued on the receiver until it is needed.

## CDS as Target for Code Generation

The **`send`**, **`recv`**, and locking routines/macros within CDS provide a simple and natural evolutionary path for back-ends already utilizing these paradigms. This section will more fully describe a more "CDS-native" programming style.

Planning a CDS program is similar to planning any other sort of concurrent program—i.e. figuring which computations can execute concurrently with which others, and what data needs to move between those pieces. It differs even here, though, because it is less important to match the number of computational pieces—CCEs—to the number of processors. In this sense, it is more instructive to think of CCEs as “portable threads” than as processes like one would use in a message-passing program: The more CCE’s, the more flexibility, since a larger number of processors can be productively accomodated, but inter-CCE communication will be minimal in cases when they must “double up” on a single processor. (Such a view will be even more valid in the future, when CCEs may be implemented literally as OS threads, thus minimizing scheduling overhead.)

In traditional message passing, the program is generated using a “push” mindset, where each data producer must know who will consume. In traditional shared memory, the program follows a “demand” mindset, and a producer cannot provide information about the eventual consumer even if known. In CDS, the program provides the information it has about the consumer’s location when that is known. If the producer doesn’t know what will consume, the producer generally **`put`**s the data into a local cell to minimize the likelihood that the **`put`** operation will result in latency, even if the eventual **`get`** from the consumer might. If a potentially-distant consumer becomes awayre of its need, it can issue a `PENDING` **`get`**, effectively prefetching the data. Of course, maximum latency-hiding opportunities still result from **`pu`**tting the data directly to into a cell at its eventual destination, if this is known at **`put`** time.

The CDS program gets extra points for leaving data in the comm heap (i.e. within the region), and for regions that are only read, not written, by some CCEs (at least for some period of time). These will provide maximum opportunity for communication without copying, and for having multiple CCEs accessing the same region at the same time. Trade-offs are still involved. For example, to determine whether logically-related data should be split into separate regions because some of it will be modified often while other rarely,one may consider expected usage frequency and latency probabilities.

Programs that benefit greatly by having the data in their private (non-comm heap) memory, perhaps in an unpacked/scattered form, should use the **`send`** and **`recv`** operations whenever they apply (instead of the corresponding primitive operations) since the combined operations provide the most opportunity for CDS to optimize out superfluous copying. (Elepar’s current product does not highly optimize these operations.)

This leaves the very basics of programming, and managing the resulting complexity. For the most part, managing cell numbers is very similar to managing tags in a message passing program. That is, a given cell usually accomodates a particular kind of region, though there are cases when a single cell can be used as a collection point for many different kinds of regions which are identified programmatically by data within each region. When using a shared-memory paradigm (e.g. using the macros), each cell corresponds roughly to a lock, but its location also represents a “home” for the covered region. This, unfortunately, has many of the same sorts of disadvantages that home processors in DSM systems—i.e. the data may end up in a different place than you know it well be needed next. For this reason, if a shared region is known to go through different phases where it will be needed in different CCEs, it is often productive to assign it several cells (“locks”), each designating an upcoming phase, each within the CCE which will be processing the data in that phase.

**`copyfm`** and **`copyto`** (or **`send`** and **`recv`**) can be used to automatically translate data when programming heterogeneous machines, but its overuse thwarts many of the efficiencies offered by CDS (e.g. copy avoidance). On the other hand, making each individual access to region data conditional on whether or not conversion is required can be very complex and error-prone. To achieve a middle ground, the program can check the archtype for each region once, shortly after **`get`**ting it, and then either (a) **`copyfm`** the data to a buffer in user space and create a pointer to it, if conversion is required, or (b) set the same pointer to point to the region itself (i.e. to `*rgid`), if not. The pointer can thereafter be used to access correctly represented data, regardless of circumstance. This protocol can be further formalized using macros/routines. It becomes somewhat less effective when the region in question contains some CCE IDs, unless all of them are at the end of the region and the caller doesn’t need to access them, since CCE IDs must be converted through the copy routines to be useful, and the room taken by a native CCE ID in the region will be very different than the size of an integer (i.e. the converted CCE ID in user space).

The CDS **`enlist`** and **`init`** operations, which are used to start the CCEs of a program, are designed to be independent of specific details of how processors are allocated or how processes are initiated. Because of the resulting flexibility, most user programs will not use them directly, but will instead call library functions customized to interface with specific processor allocation systems (if any), and to create CCEs in a certain topology, or with certain startup conditions.

An existing CCE cannot address the cells in a new CCE unless it knows the new ID, and it can’t know this until the new CCE (or a CCE it has already told) reveals it. Likewise, the new CCE cannot inform any other CCE of its ID unless it knows *their* ID. This can be used to implement information hiding, but must also be considered during program initiation. Since a new CCE always knows its enlistor's ID, one common user-enforced protocol is for the enlistor to **`get`** a region (the so-called "birthcry") from the new CCE (in one of the enlistor's cells) containing the new ID. This **`get`** should have a timeout value specified to account for the case where the new CCE never succeeds in producing the birthcry. (If the enlistor can do productive work between the **`enlist`** and the **`get`**, it may be able to completely hide CCE creation overhead.) Alternate startup protocols are possible by having the enlistor provide one or more other CCE IDs within the enlistment region (i.e. placed into cell 0 of the enlisted CCE). In any case, the new CCE is anonymous between the time that it calls **`init`** and the time that it **`put`**s its ID into another CCE's cell, so the intervening interval is precisely when it should call **`cagrow`** to create its comm area (i.e. comm heap and comm cells): Waiting until after it performs the **`put`** runs the risk of other CCEs attempting to access the cells before they exist. Note that the comm heap can become fragmented over time, so to ensure that sufficient space will be available for any particular region, it may be necessary to create it significantly larger than the total regions that may actually be stored at any one time.

Handlers can cause action to occur when a particular cell gets too empty or too full, or can even facilitate a purely demand-driven programming methodology, where the main program effectively stops and all subsequent activity occurs within handlers. One of the most interesting operations is to install a handler with a low-water mark of 0, meaning that the handler will be requested only if a **`get`** is performed on the empty cell. The handler can therefore **`put`** a region into the cell to satisfy the **`get`** which caused it to be requested.

## Performance

Although CDS's performance advantages come primarily from copy avoidance when CCEs share memory, it also enjoys advantages over message passing systems (e.g. PVM, MPI) for other reasons. Housekeeping messages, used in MPI to ensure a ready buffer on the receiver, are rarely needed in CDS because the user is responsible for always ensuring sufficient comm heap. Also, while PVM and MPI require that all transfered data be processed through a "type", CDS allows access to the raw payload, and makes translation (i.e. **`copyfm`** and **`copyto`**) optional. As a result, CDS can deliver more of the raw fabric bandwidth to the application.

Elepar's current CDS implementation (called BCR) uses standard SysV shared memory segments, UDP/IP communication, and custom low-level locking. It does not implement lock-free queues at this time. We have run some simple benchmarks on a small heterogenous cluster. The "ring" benchmark passes regions of varying sizes around CCEs using simple **`enq`** & **`deq`**, and we ran it on a 2-processor Pentium III 850MHz running Linux and a Mac Powerbook 266MHz G3 running LinuxPPC communicating over 10baseT ethernet. Each pass (put+get) took 2.8 μsecs with 2 CCEs running on the PC (independent of region length), or 5.6 μsecs with 3 CCEs. Across the ethernet, bandwidths of 58.6KB/s, 293KB/s, 957KB/s, and 1.05MB/s were observed for region sizes of 10, 100, 1000, and 10000 bytes, respectively—nearly native hardware capacity even at relatively small region sizes.

## Future Plans and Conclusions

Elepar considers this an early version for CDS, hence calling it "Before CDS Redesign" (BCR). Further features under consideration include cell contexts, more forms of non-blocking **`recv`** and blocking **`send`**, and potentially collective operations. Standards are feasible.

CDS provides an expressive and flexible subroutine-based interface, together with simple semantics and comprehensible performance model, to serve as a target for scalable languages, and to provide a runtime with the information it needs to make wise dynamic performance decisions based on information extracted from the source program and physical conditions.

## References

[1] D. DiNucci, R. Babb II, "Design and Implementation of Parallel Programs with LGDF2", *Digest of papers from Compcon '89*, IEEE, pp. 102-107.

[2] D. DiNucci, "A Simple and Efficient Process and Communication Abstraction for Network Operating Systems", *Proceedings of CANPC '89*, LNCS volume 1199, Springer-Verlag, pp.31-45, D. Panda and C. Stunkel, Eds., Feb 1997.

[3] D. DiNucci, "BCR Principles and User Guide v1.1", Elepar, Jan 2002.

[4] K. Johnson, M. F. Kaashoek, D. Wallach. "CRL: High-Performance All-Software Distributed Shared Memory", *Proceedings 15th Symposium on Operating System Principles*, Dec 1995.

[5] P. Keleher, S. Dwarkadas, A. Cox, W. Zwaenepoel, "Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems", *Proceedings 1994 Winter Usenix Conference*, pp. 115-131, Jan 1994.

[6] P. Keleher, "CVM: The Coherent Virtual Machine", University of Maryland, November 1996.

[7] J. Seizovic, "The Reactive Kernel", Masters Thesis, CSTR tr-88-10, Caltech, October 1988.