Making Peer-to-Peer Computing Truly Computing Among Peers

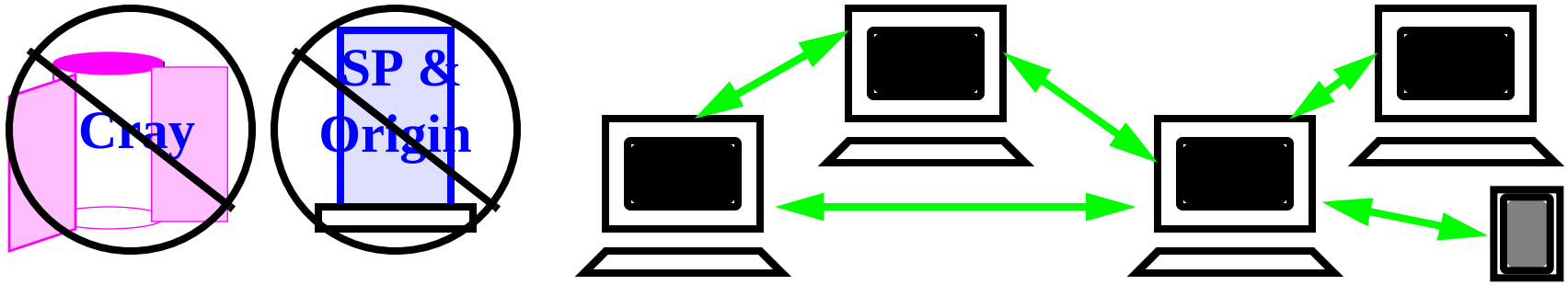
David DiNucci, PhD

Elepar: Working Together Independently

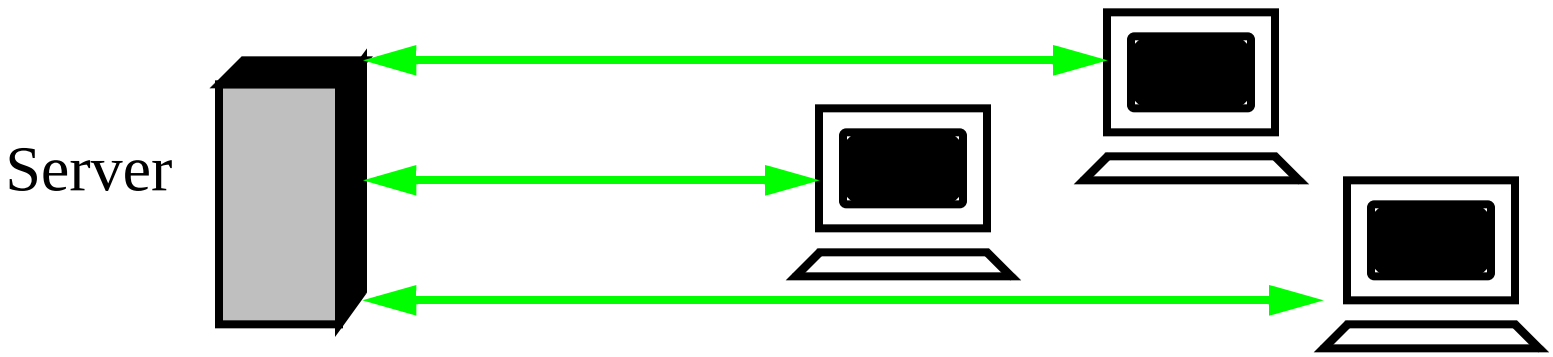
`dave@elepar.com` `www.elepar.com`

Peer-to-Peer Computing is...

Hype, partly implied by “Peer-to-Peer” moniker



Current reality (e.g. SETI@home, Intel/UD cancer cure)



So for now, only effective on data parallel—i.e. uniform calculations over huge domains and/or parameter spaces

Why Reality != Hype

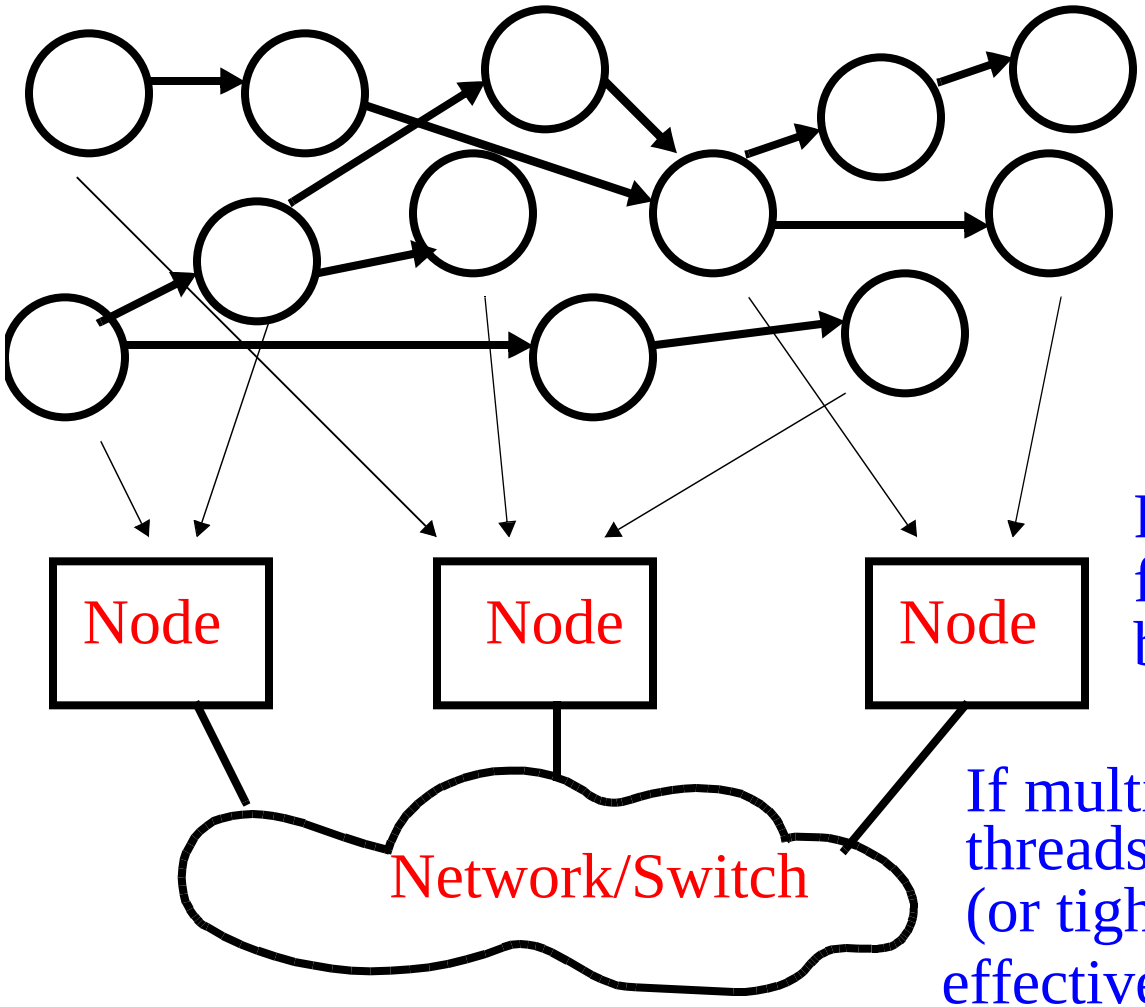
As usual, achieving the hype is too hard!

- **Dynamic topology -> Resource discovery/reservation/scheduling**
- **Heterogeneous speed/archit. -> Portability/Variable granularity**
- **Local or distributed comm -> Latency tolerance/Low overhead**
- **Many decentralized components -> Fault tolerance**
- **Complex & concurrent -> Formal analysis, debugging rules**
- **App still #1 -> Leveraging existing tools, languages, techniques**
- **Utilizing untrusted resources -> Privacy/Security/Anonymity**

...i.e. all of the hard problems of parallel and distributed computing over the decades. (May be reason hardware corp's so behind P2P!)

If you had a program with all of these traits, wouldn't you want to be able to run it in more environments than just peer-to-peer?

High-Level Plan of Attack...Dataflowish

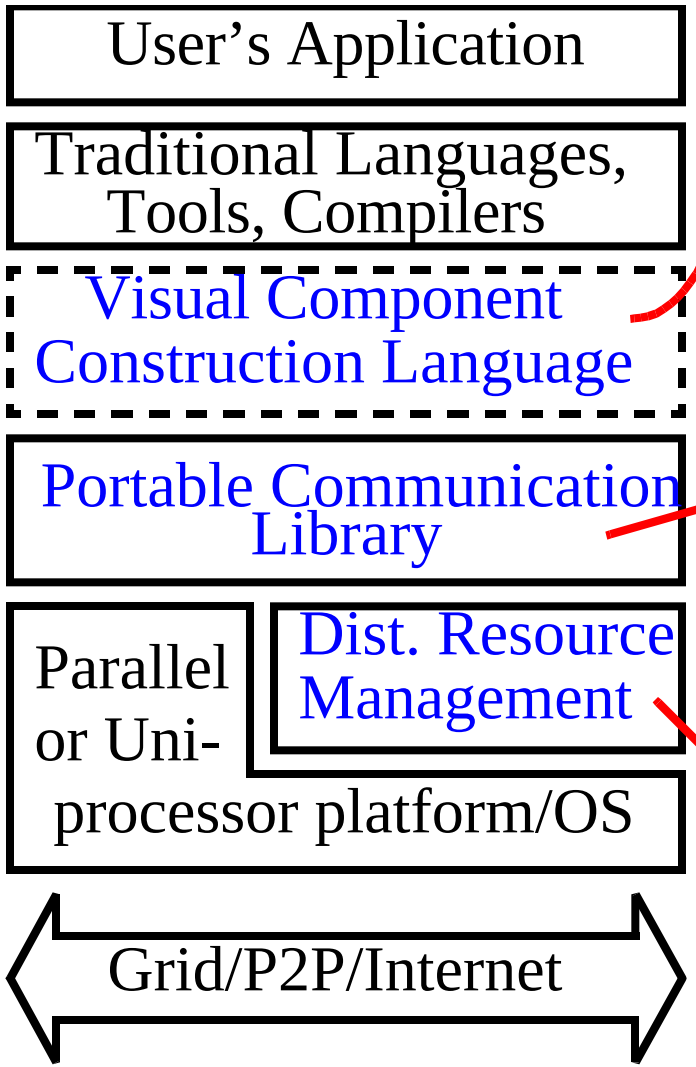


App is composed of "Portable threads" implemented in your favorite language. Stateless, atomic --> functional

Resources (processors) found & scheduled via bidding and brokers

If multiple communicating threads end up on same node (or tightly bound nodes), they effectively "merge" ovhd-wise

Elepar's Three Layers



Software Cabling (SC): Visual OO component coordination methodology for building & analyzing portable, distributable apps from modules implemented in traditional langs

Cooperative Data Sharing (CDS): Efficiently supports messaging (push), blackboard/shared (pull), & hybrid styles on variety of architectures

People, Instruments, Computers, and Archives (PICA): Rules and protocols for finding, bargaining for, and scheduling distributed, independently-controlled resources of all kinds

CDS: Cooperative Data Sharing

History: **Early musings at OGI, prototype implemented and published at NASA Ames, now under development at Elepar**

Approach: **Determine common features of shared memory and message passing, build subroutine interface around those features, include other expected features (process control, active messages, conversion/marshalling).**

Result: **Compared to other communication layers (e.g. MPI, sockets, DSM), it is:**

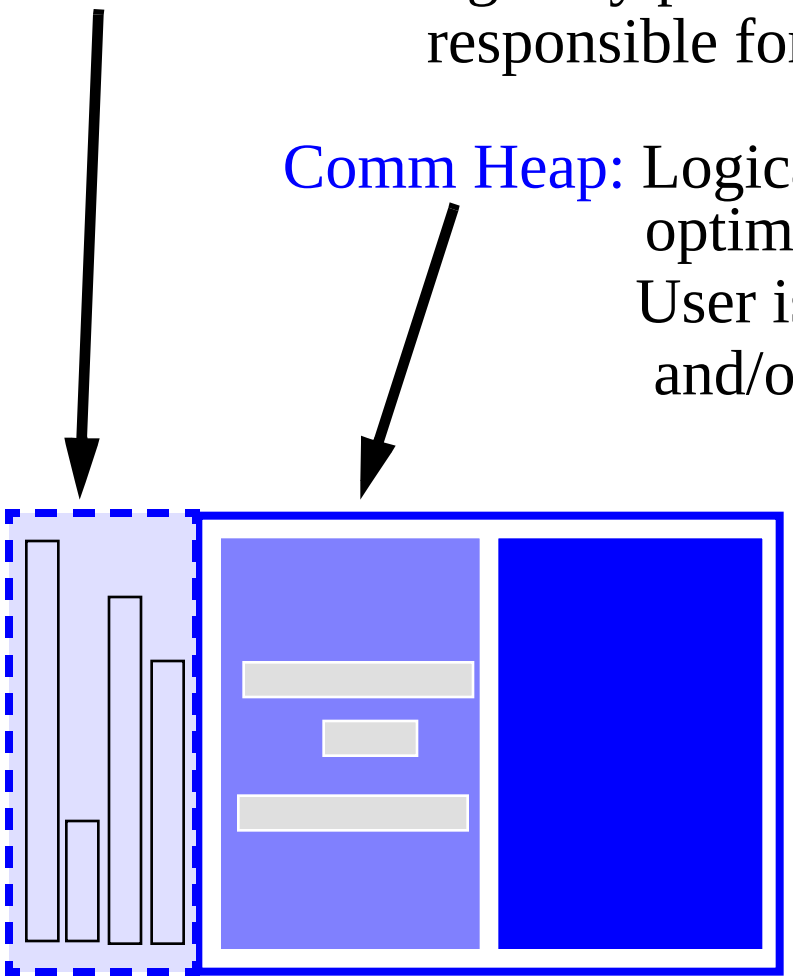
- **Relatively simple/Small**
- **Expressive/Powerful**
- **Very portable to different uniprocessor & parallel architectures**

CDS: Anatomy of a CDS Process

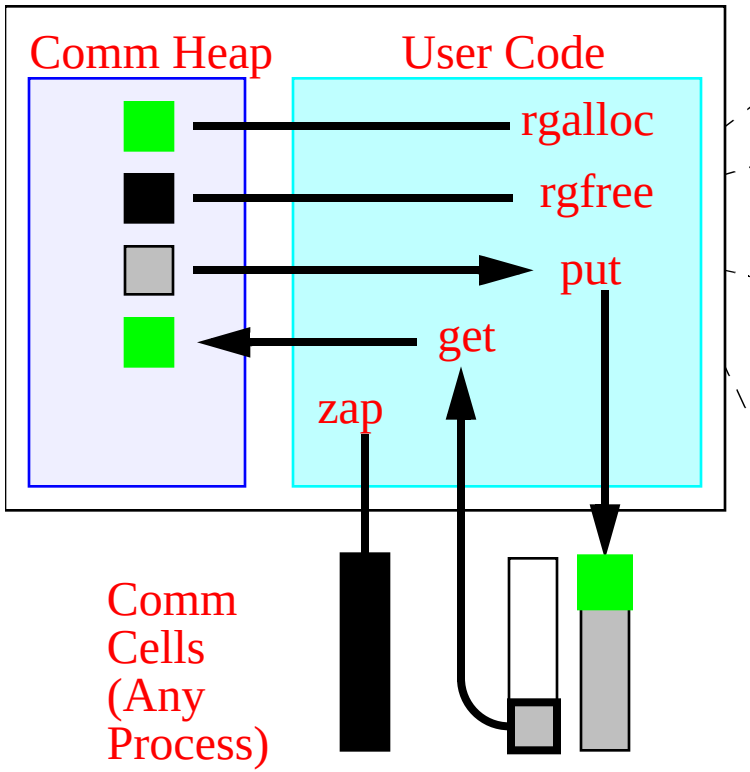
Comm Cells: Logically public set of queues. User is responsible for creating and deleting.

Comm Heap: Logically private heap. Data is optimized for communication. User is responsible for enlarging and/or shrinking.

User code & data: Standard Unix process.



CDS Basic Communication Operations



Allocates a region in the local comm heap

Frees up a region in the local comm heap

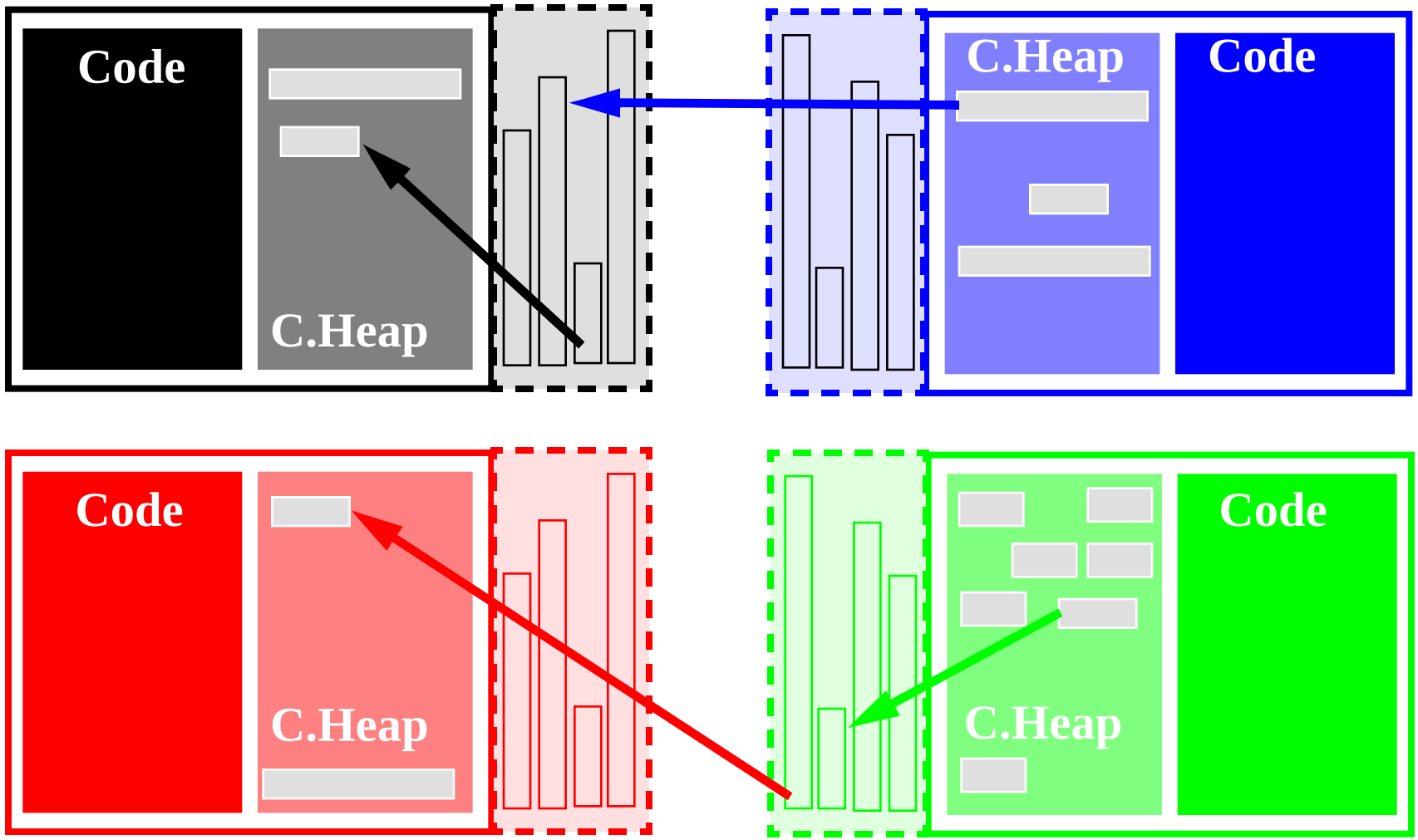
"Copies" region from local comm heap to end of any cell, optionally freeing region from heap and/or zapping cell before depositing new region. AKA "write" if cell zapped, "enq" if not. bput same, but blocks until cell empty and a get is waiting

"Copies" region from beginning of any cell to local comm heap, optionally removing it from cell after. AKA "deq" if removed, "read" if not.

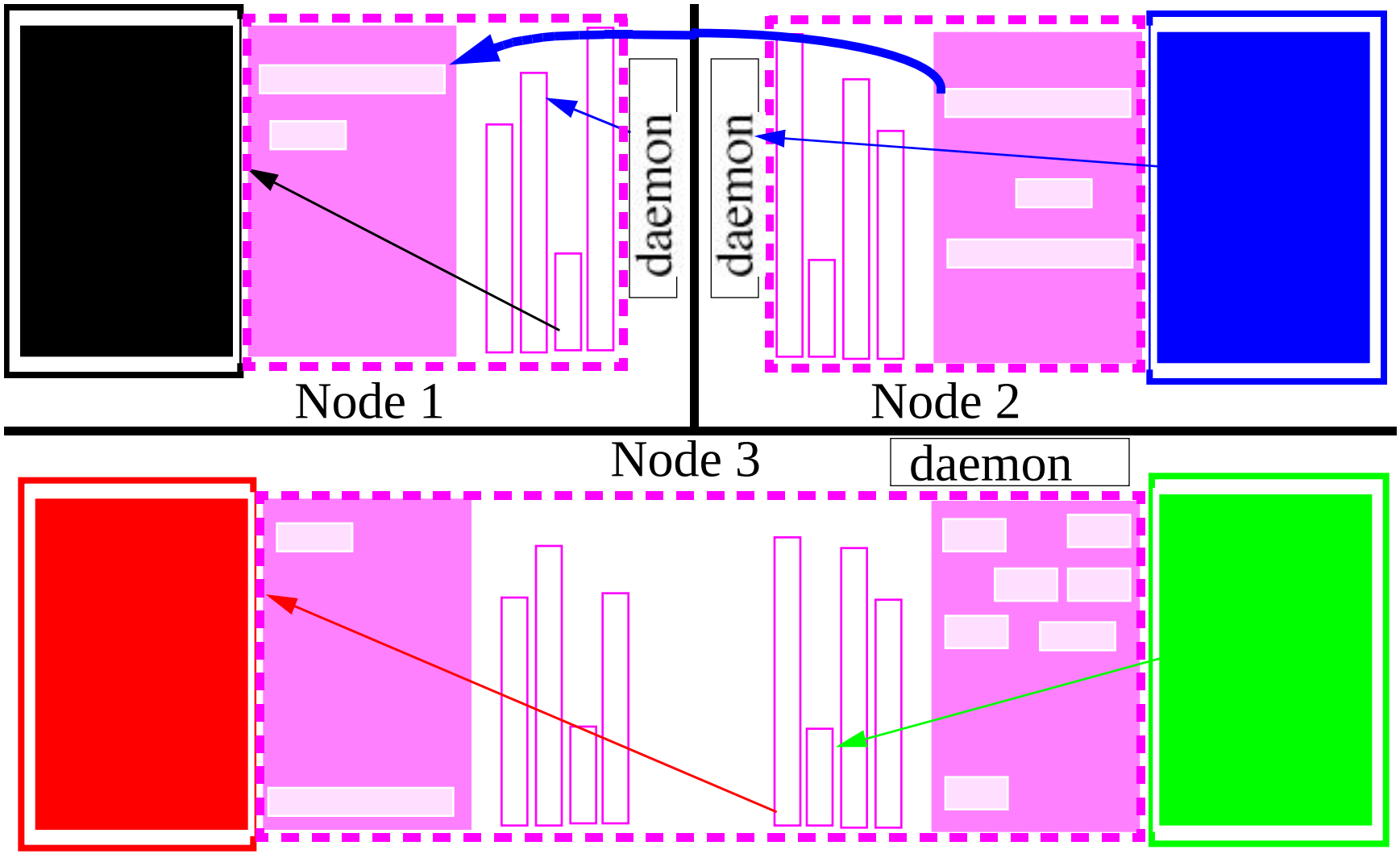
All ops that can block (i.e. `bput`, `get`, `deq` and `read`) take a time-out value, and also "i" versions (`ibput`, `iget`, `ideq`, and `iread`, respectively), resolved with a `wait` op.

"Copy" operation is virtual (i.e. usually copy on write), so these are usually just pointer ops. For portability, `rgmod` must be called before modifying any potentially-shared rgn.

CDS: Logical View



CDS: Physical View on 3 Nodes



CDS: Other Functionality

Process Initiation/Active Messages ("Handlers")

High and low water marks can be set on each cell

A "handler" function can be chosed to be invoked each time that watermark is exceeded.

Copying/Marshalling/Conversion

Although process can access regions in comm heap directly, "`copyfm`", "`copyto`" routines exist to pack, unpack, and/or convert data as it is being moved to or from region, based on internally-supported conversion tables.

CDS Shared Mem & Msg Passing "Macros"

enqing reqion ~= releasing a lock, deqing region ~= acquiring a lock.

"Macro"	Meaning	Translates Into
`acqwl`	Acquire write lock	`deq, rgmod`
`rlswl`	Release write lock	`write, rgfree`
`acqrl`	Acquire read lock	`read`
`rlsrl`	Release read lock	`rgfree`
`wl2rl`	Write lock -> read lock	`write`

Msg passing includes copy to/from comm heap, can be optimized out.

"Macro"	Meaning	Semantically identical to
`send`	Send message	`rgalloc, copyto, enq, rgfree`
`recv`	Receive message	`deq, copyfm, rgfree`
`sendx`	Destructive send	`rgalloc, copyto, write, rgfree`
`recvx`	Non-destructive receive	`read, copyfm, rgfree`
`bsend`	Synchro or ready send	`rgalloc, copyto, bput, rgfree`

Corresponding "i" ops: `iacqrl`, `iacqwl`, `irecv`, `irecvx`, `ibsend`

Comparing CDS Featureset

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		x	x	?
Data can be queued at producer waiting for pull	x		x	x	?
Pushed data can be made to overwrite previous value	x	x			x
Producer can retain access rights to comm'd data	x		2		x
Producer can relinq access rights to comm'd data	x	x	x		x
Dynamic memory allocation for shared memory	x	?			
Consumer can specify timeout for waiting	x	?			
Supports heterogeneous platforms	x		x		
Simplicity (~number of function + macro interfaces)	51	20	!!!	13	5

The CDS Interface

Managing comm heap and contexts/cells

`rgalloc rgmod rgfree rgsize rgrealloc`

`addcntxt delcntxt grwcntxt`

Communication Primitives

`read deq benq enq write zap enqm writem`

`iread ideq ibenq wait waitm ienqm benqm`

Copying and Translation

`copyto copyfm copytofm transtab`

Composite functions (shared mem and msg passing)

`recv bsend recvx send sendx sendm sendxm`

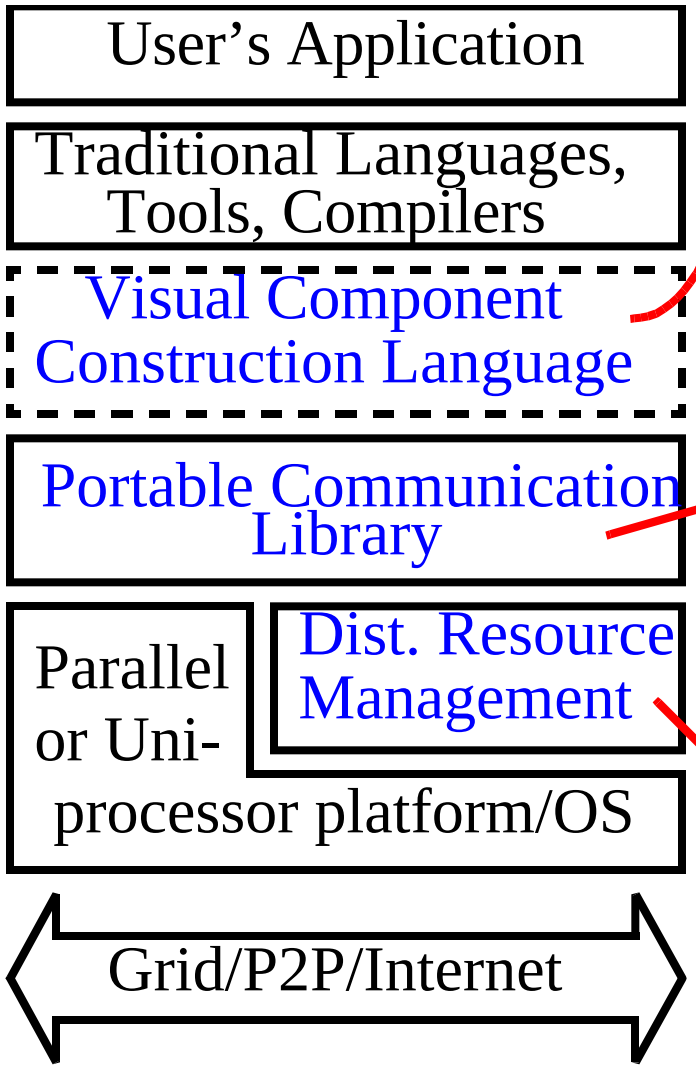
`acqrl acqwl rlsrl rlswl wl2rl`

`irecv ibsend irecvx iacqrl iacqwl`

Process and thread control

`enlist init myinfo hdlr prior`

Elepar's Three Layers



Software Cabling (SC): Visual OO component coordination methodology for building & analyzing portable, distributable apps from modules implemented in traditional langs

Cooperative Data Sharing (CDS): Efficiently supports messaging (push), blackboard/shared (pull), & hybrid styles on variety of architectures

People, Instruments, Computers, and Archives (PICA): Rules and protocols for finding, bargaining for, and scheduling distributed, independently-controlled resources of all kinds

SC: Software Cabling

CDS lays the groundwork for creating apps as collections of small, portable, sequential threads which can be stateless and atomic.

SC is a very-high-level graphical component composition language which leverages CDS communication style. It contains the necessary constructs to make real-world programming possible: e.g.

- **Modular (OO), template-based program construction**
- **Adaptable to the language of the programmer's choice (e.g. Java, Fortran, C, C++)**
- **Supports arrays and data parallelism**

Based on a formal computational model (F-Nets), providing leverage for program verification and powerful debugging techiques & replay

SC: Modules

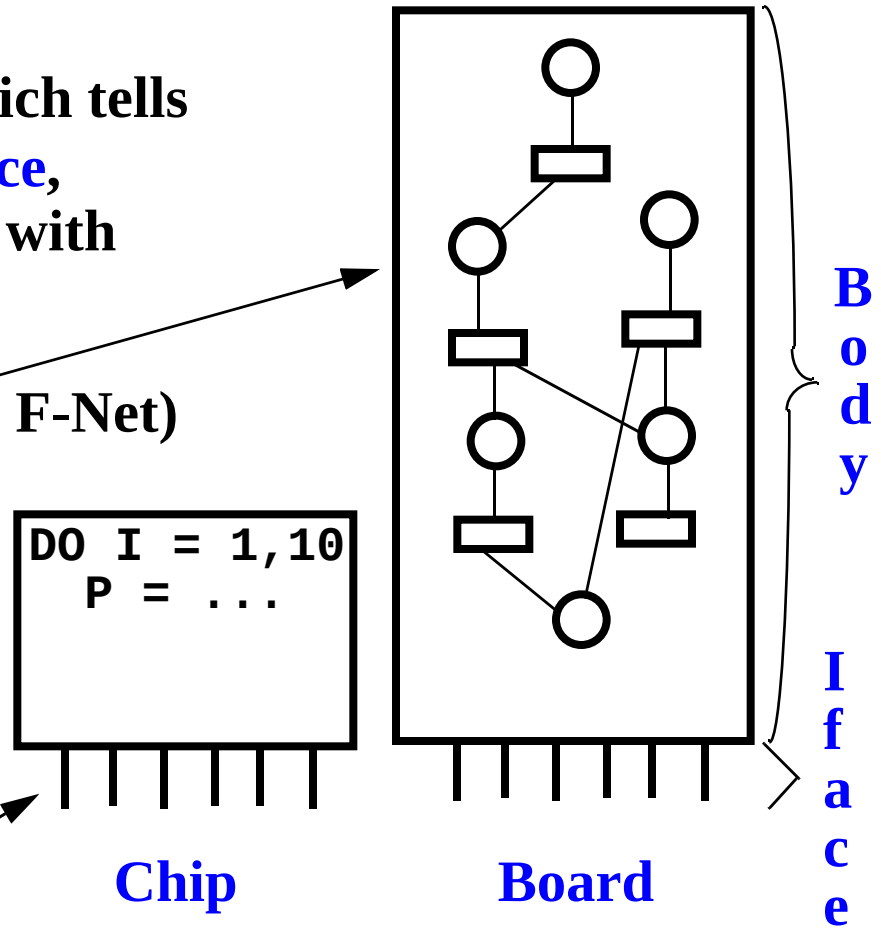
Two kinds of modules:

Both consist of a body, which tells what to do, and an interface, which allows it to interact with its environment

Board body is graphical (like F-Net)

Chip body is written in your favorite language (e.g. Fortran)

Interface consists of pins which carry data and signals.



SC: Chip Body

A chip's body is a subprogram or function, and the pins in the chip's interface are its arguments.

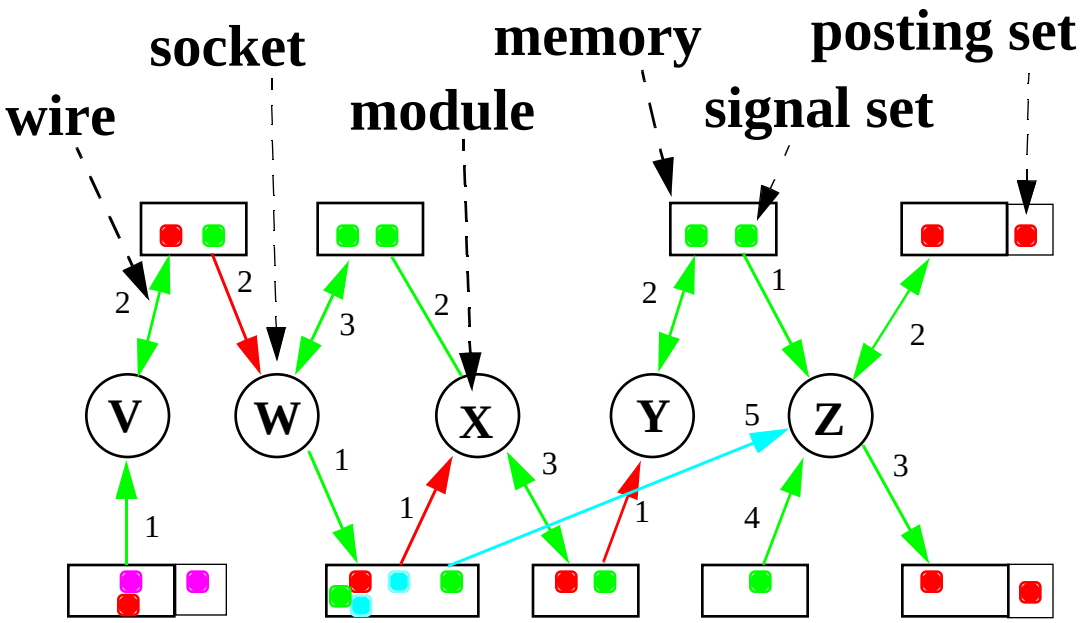
To change the color of a memory, the chip posts a signal to the pin, using a special statement of roughly the form:

```
post signame to pinname
```

This is the only special statement in your code, and it does not block or otherwise change the local behavior of the code*

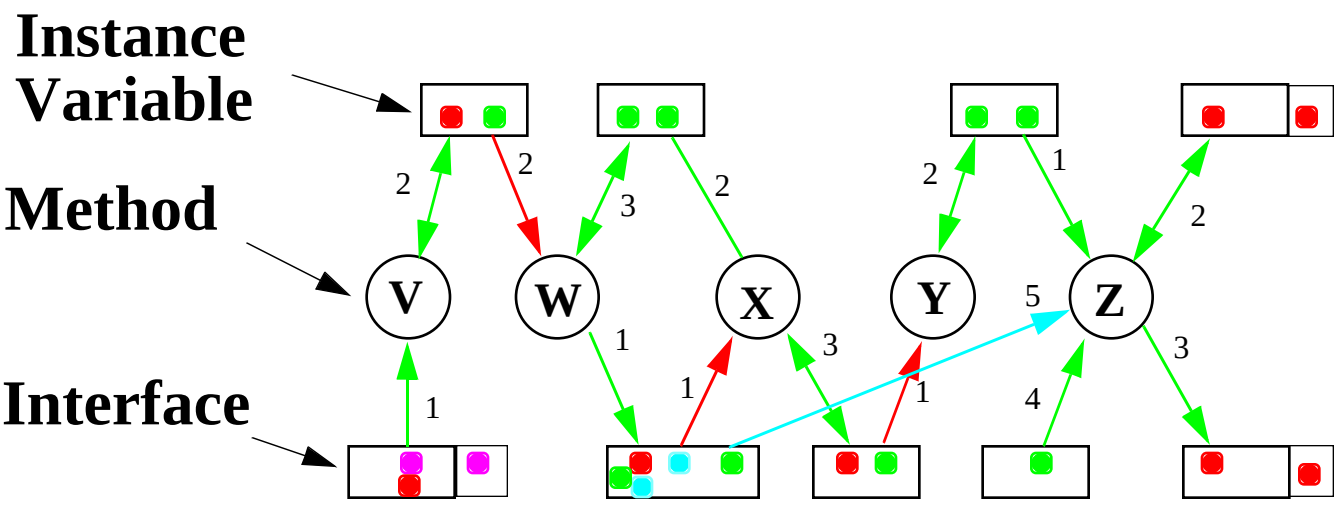
***except f or optionally making the pin ar gument inaccessible**

SC: Board Body



SC: Boards are Objects (if you like)

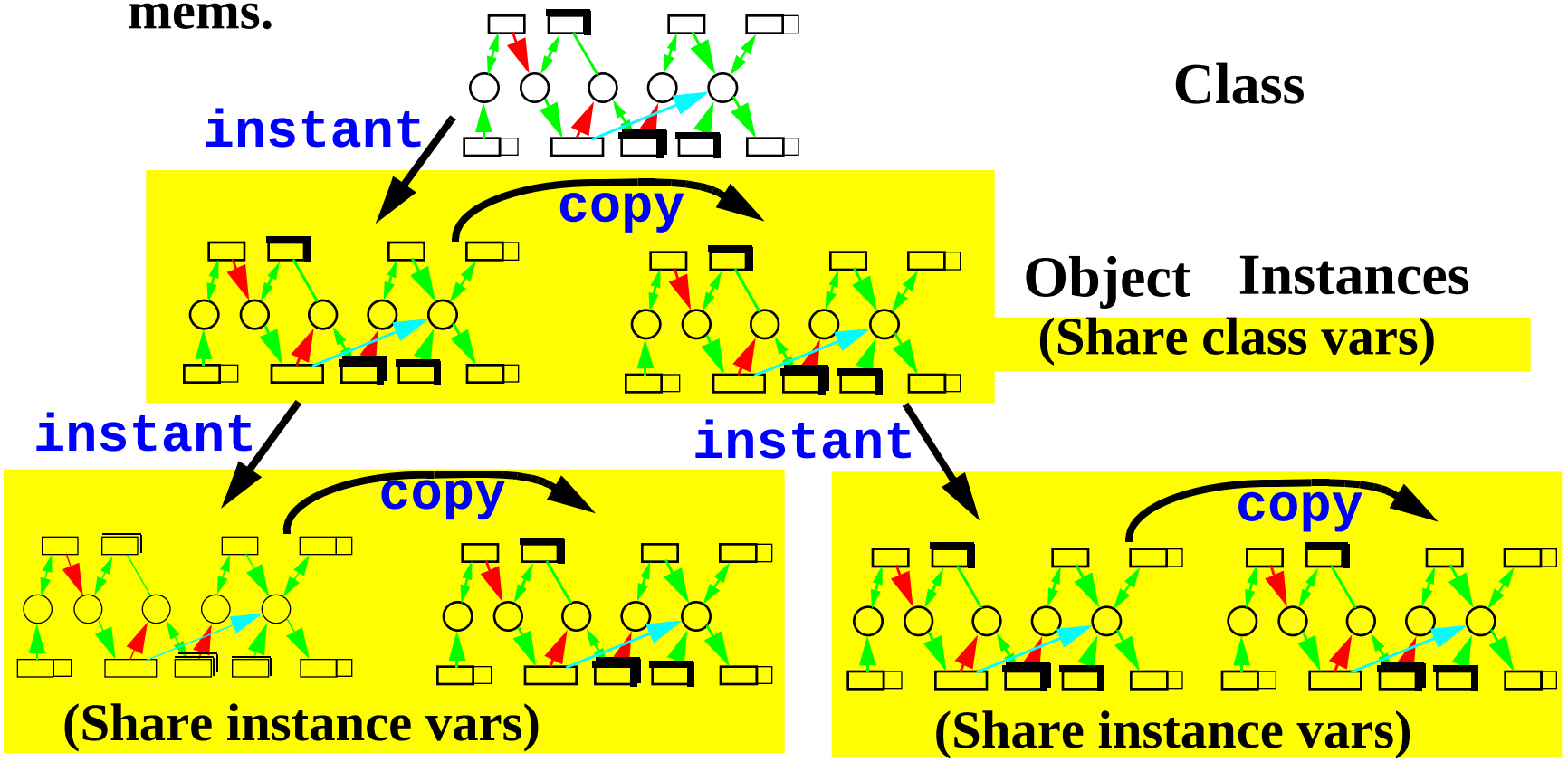
Can be seen as methods (modules in sock ets) which can be in vok ed by "messages" (signals) thr ough a well-defi ned interface to modify and r etur n encapsulated data .



Classes ar e about using a single object fr om differ en call sites? i.e. what about "classes"?

SC: Getting Classes

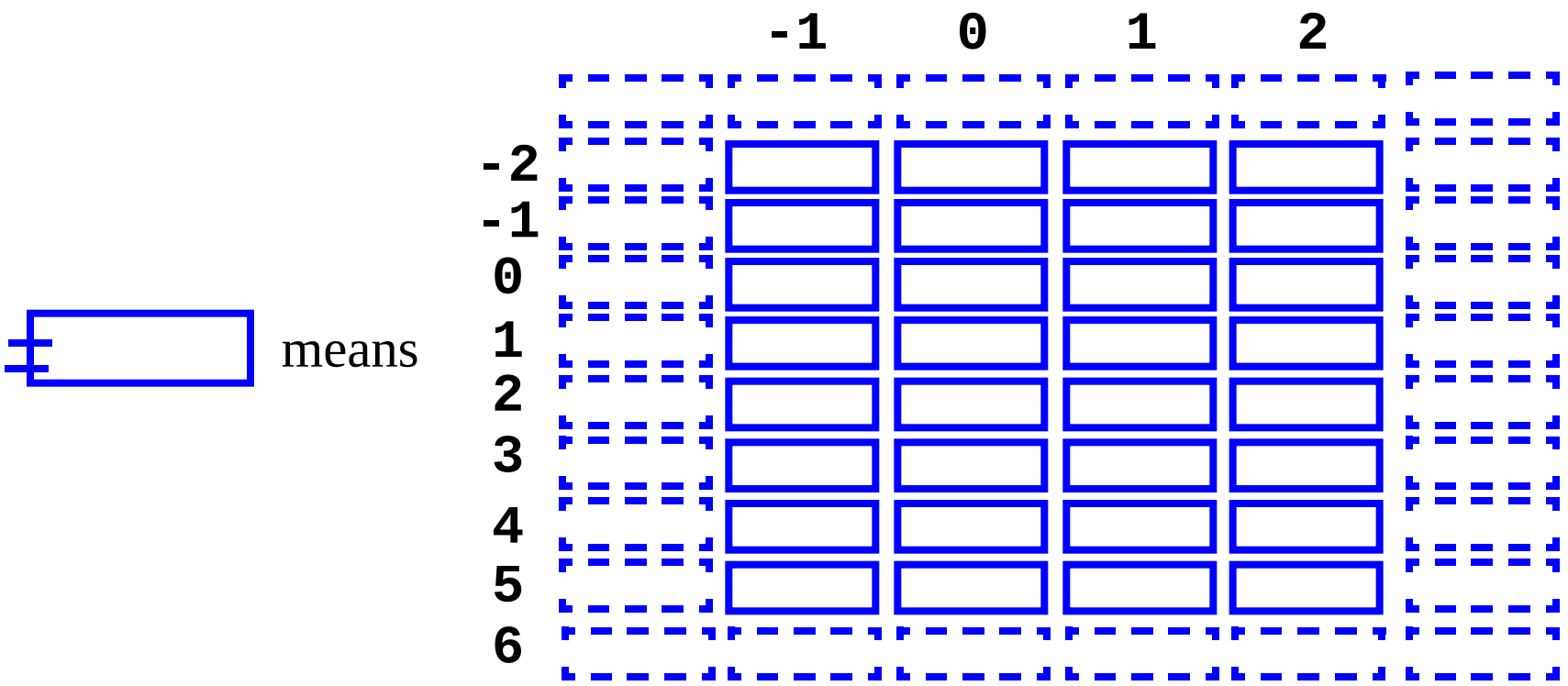
Two operations: `instant` pre-creates some mems on a board, and `copy` clones a board, sharing pre-created mems.



SC: Arrays

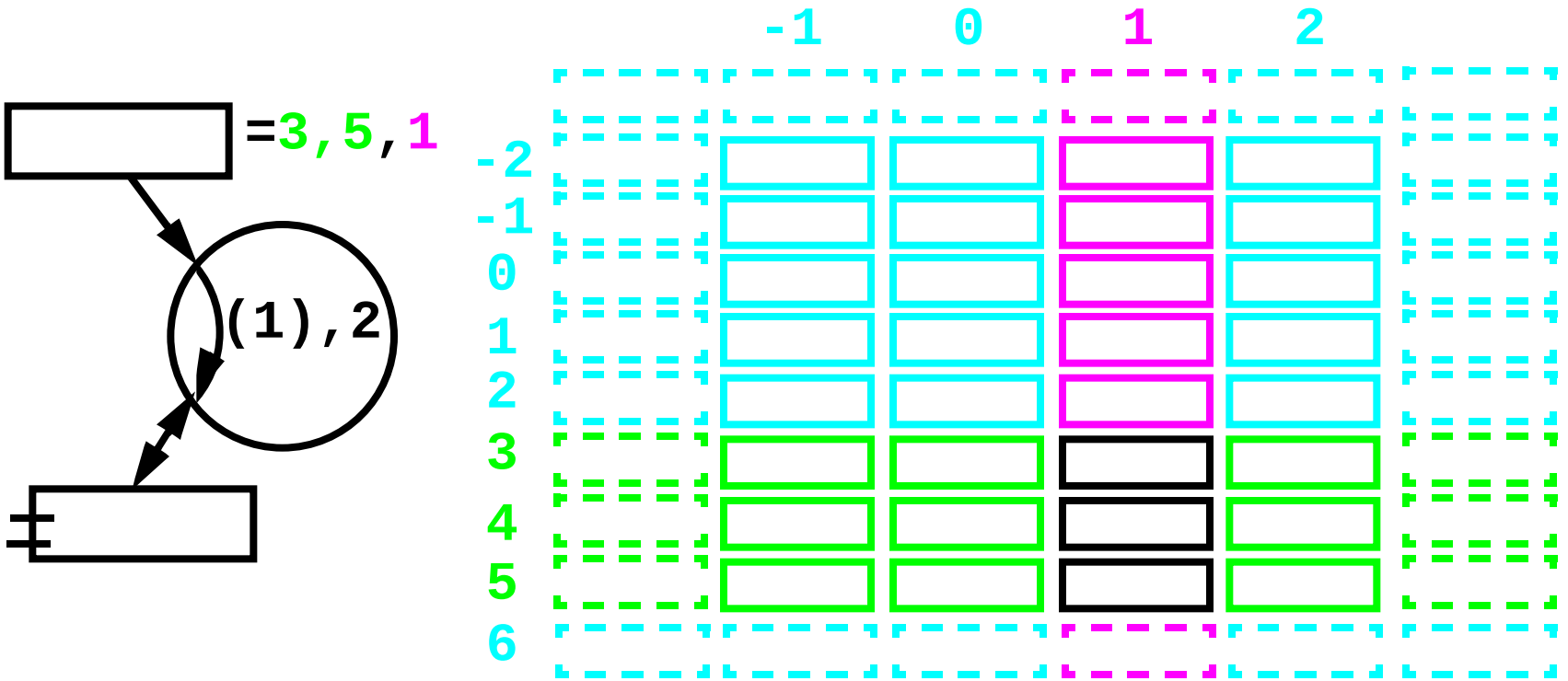
Putting hash marks in the left of a memory rectangle means that it represents an array of memories (# of dims = # marks)

Arrays extend infinitely in every direction.



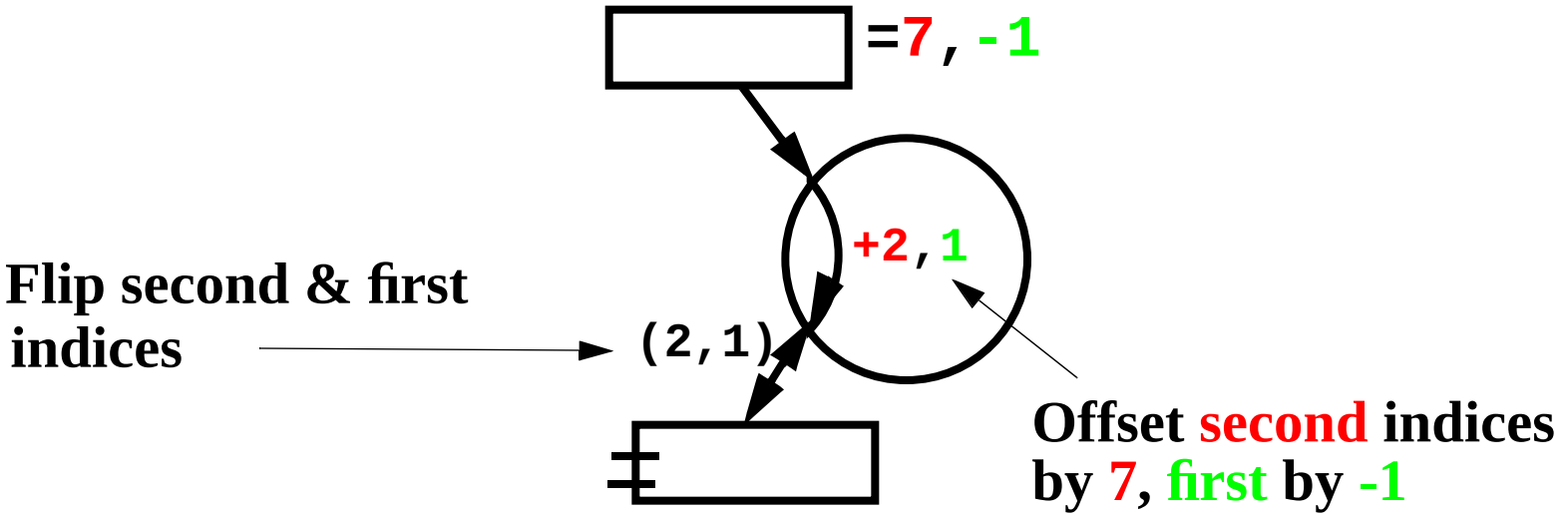
SC: Arrays (cont'd)

A selection, like subscripting, permits a socket to restrict access to an index or index range in one or more dimensions.



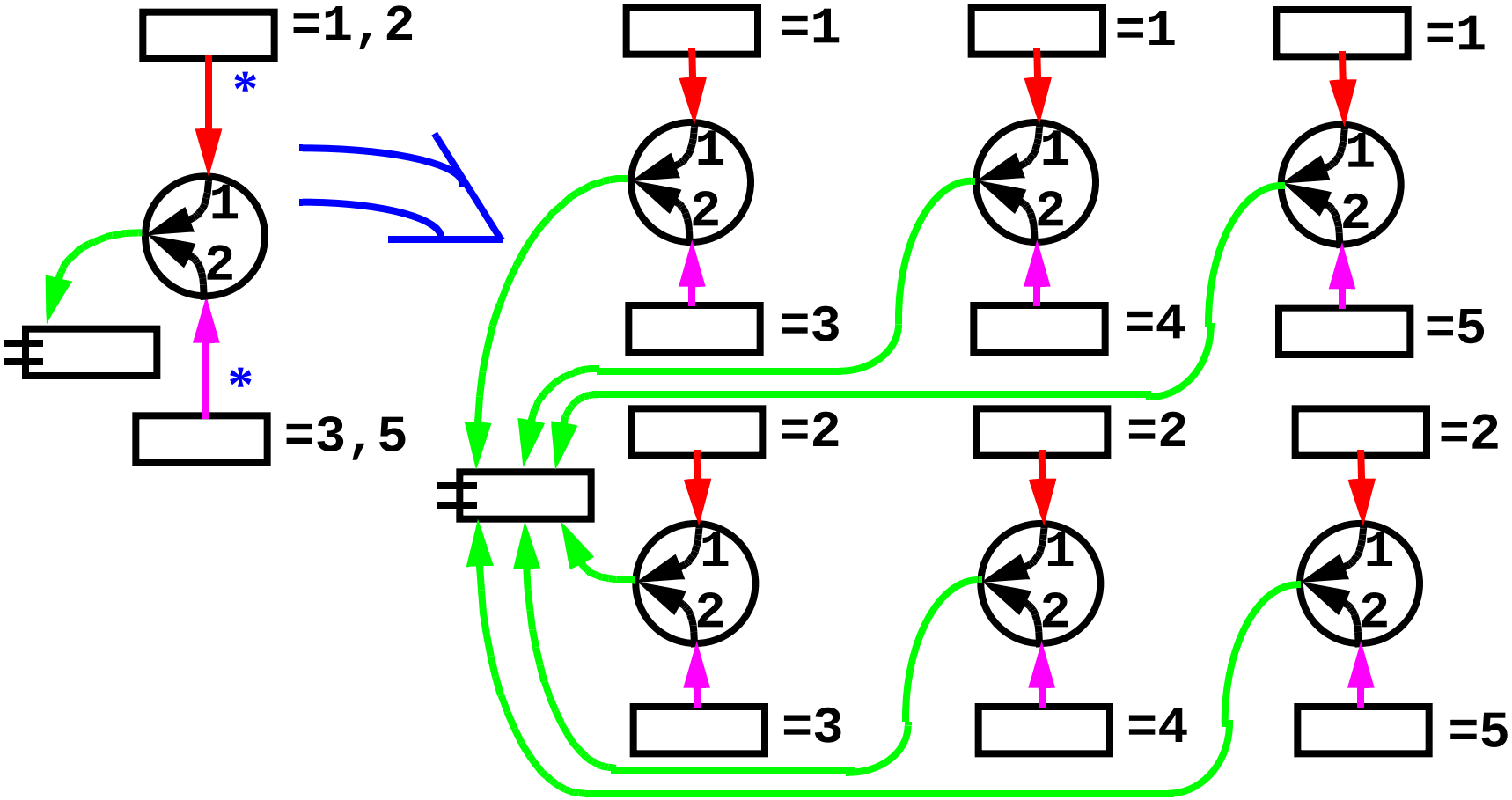
SC: Arrays (cont'd)

Translation and permutation operate on entire array, to offset array indices and alter the order of indices (effectively transposing the array)

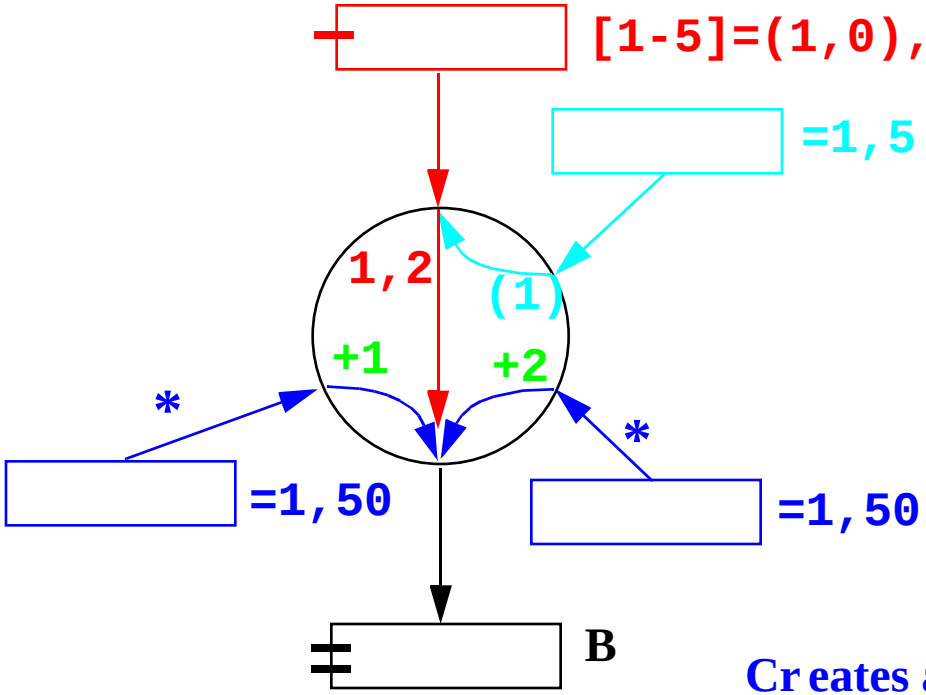


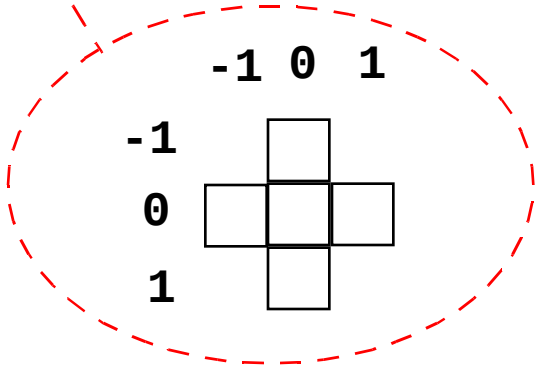
SC: Data Parallelism

For data parallelism, sockets (and therefore modules) are replicated in SC using the DupAll and DupAny



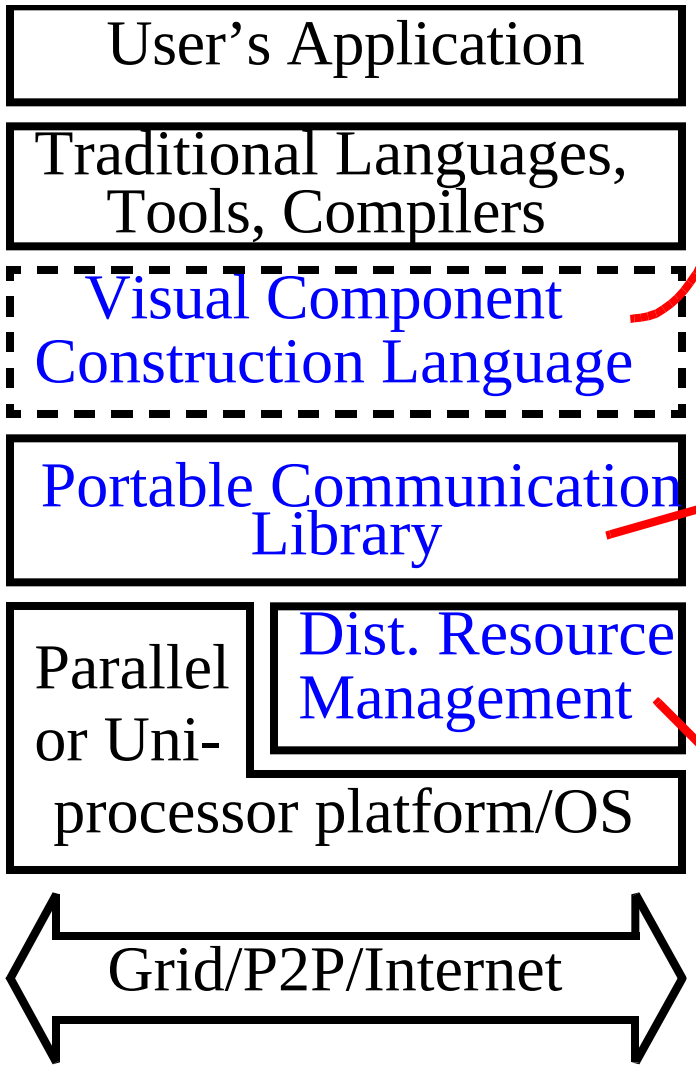
SC: Array Example w/the works





Cr eates a 50 by 50 array of sock ets, offsets B by a differ ent amount in X and Y f or each, and accesses a cr oss-shaped stencil fr om B center ed at 0,0

Elepar's Three Layers



Software Cabling (SC): Visual OO component coordination methodology for building & analyzing portable, distributable apps from modules implemented in traditional langs

Cooperative Data Sharing (CDS): Efficiently supports messaging (push), blackboard/shared (pull), & hybrid styles on variety of architectures

People, Instruments, Computers, and Archives (PICA): Rules and protocols for finding, bargaining for, and scheduling distributed, independently-controlled resources of all kinds

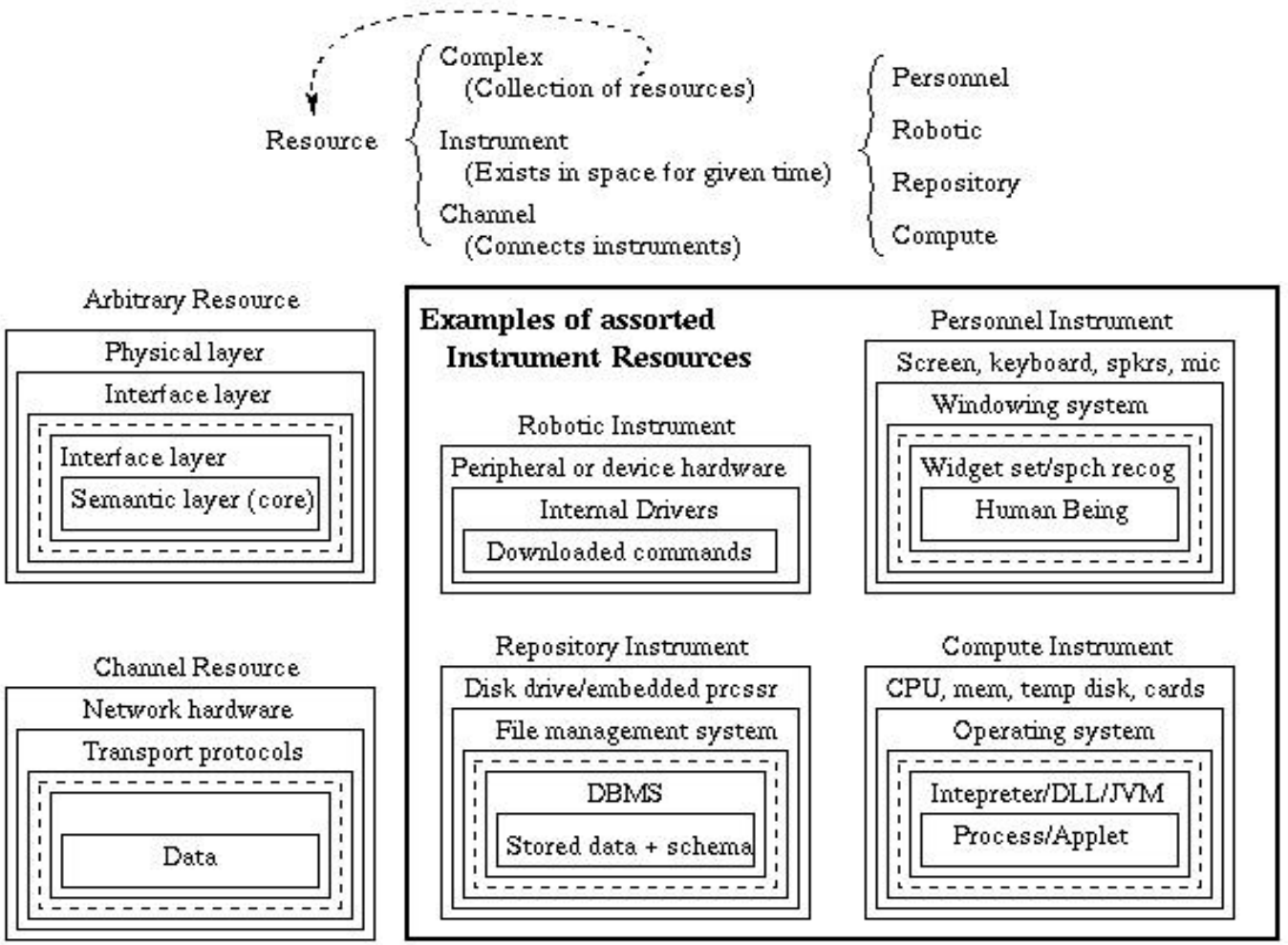
PICA: People, Instruments, Comp's, Archives

Protocols and guidelines for distributed resource discovery, bidding, (co)scheduling and reservation, and usage/relinquishment

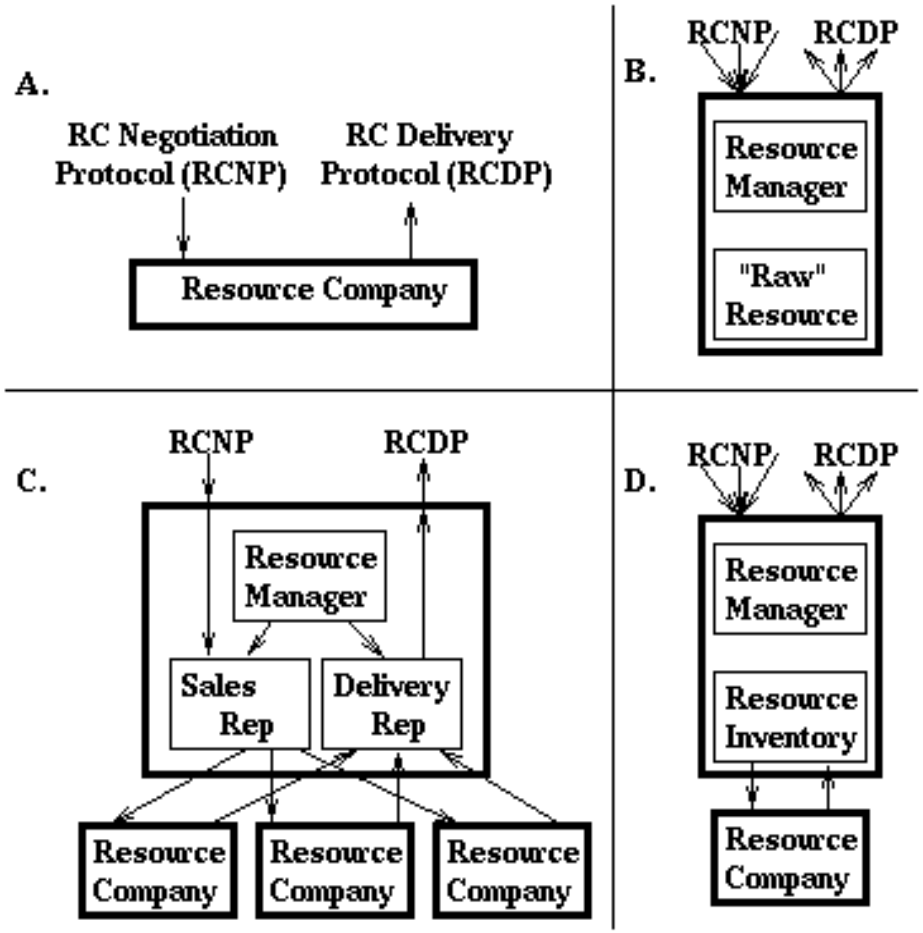
Four principle components:

- Resources: Standard way to specify complex resources
- Resource Companies: Standard way to request, bid for, and/or provide resources
- Resource Keys (i.e. capabilities): How resources are passed from place to place
- Resource Supply Chains: Fan-in/Fan-out of complex resources and payments between suppliers and customers

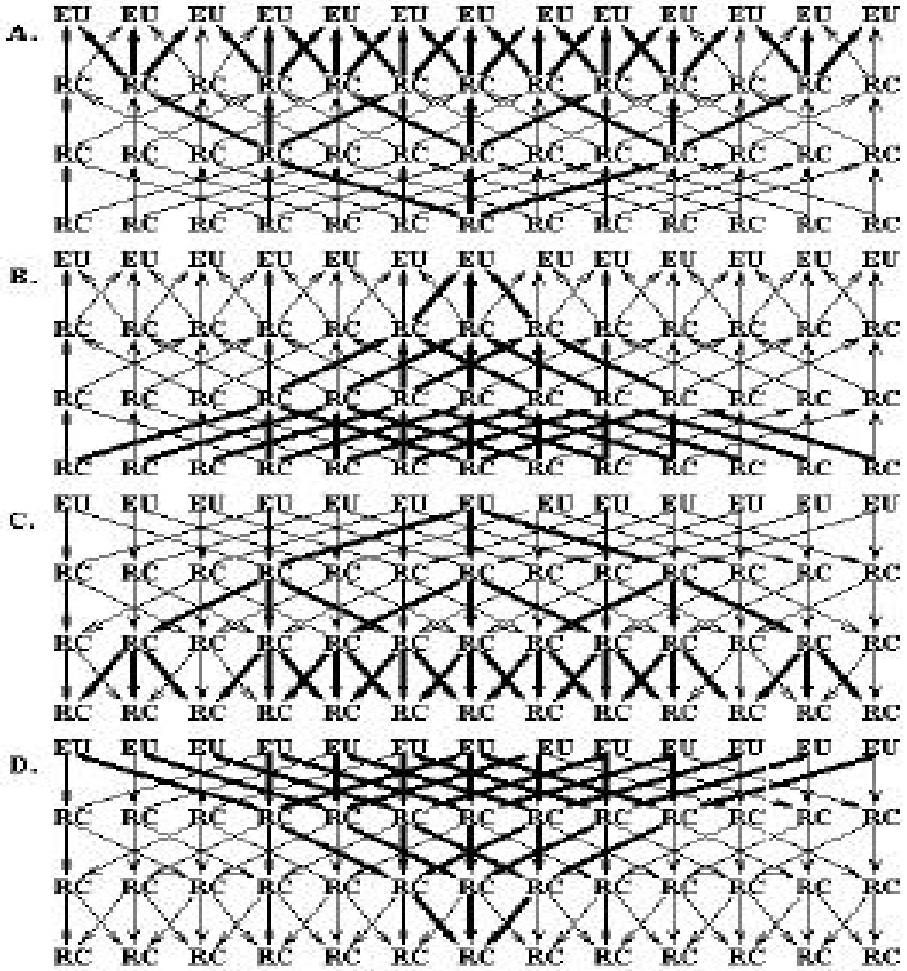
PICA: Resources



PICA: Resource Companies



PICA: Supply Chain



Overall Summary

True P2P Computing is a BIG problem.

Nothing is wrong with picking the low-hanging fruit, but it's not clear that one can get to the ultimate goal via little steps.

High-level languages are here for a reason: To simplify programming by abstracting the interface, providing portability, amortizing the programming investment. P2P doesn't change that.

Characterizing any program based on the architecture it runs on, whether P2P or anything else, is often a mistake unless one is intentionally harnessing unique characteristics of that architecture