Modular Internet Programming with Cells

Ran Rinat

Scott Smith

http://www.jcells.org

Motivations

- Persistent language-level network connections
  - Tightly coupled Internet protocols keep a persistent socket connection; no language-layer protocols do this
- Java/.NET are first generation Internet languages
  - Lets work on the second generation
- Modules and components have commonalities
  - Unify them
- Code architecture that mirrors deployment architecture
  - Current practice declares module interface but not network interface

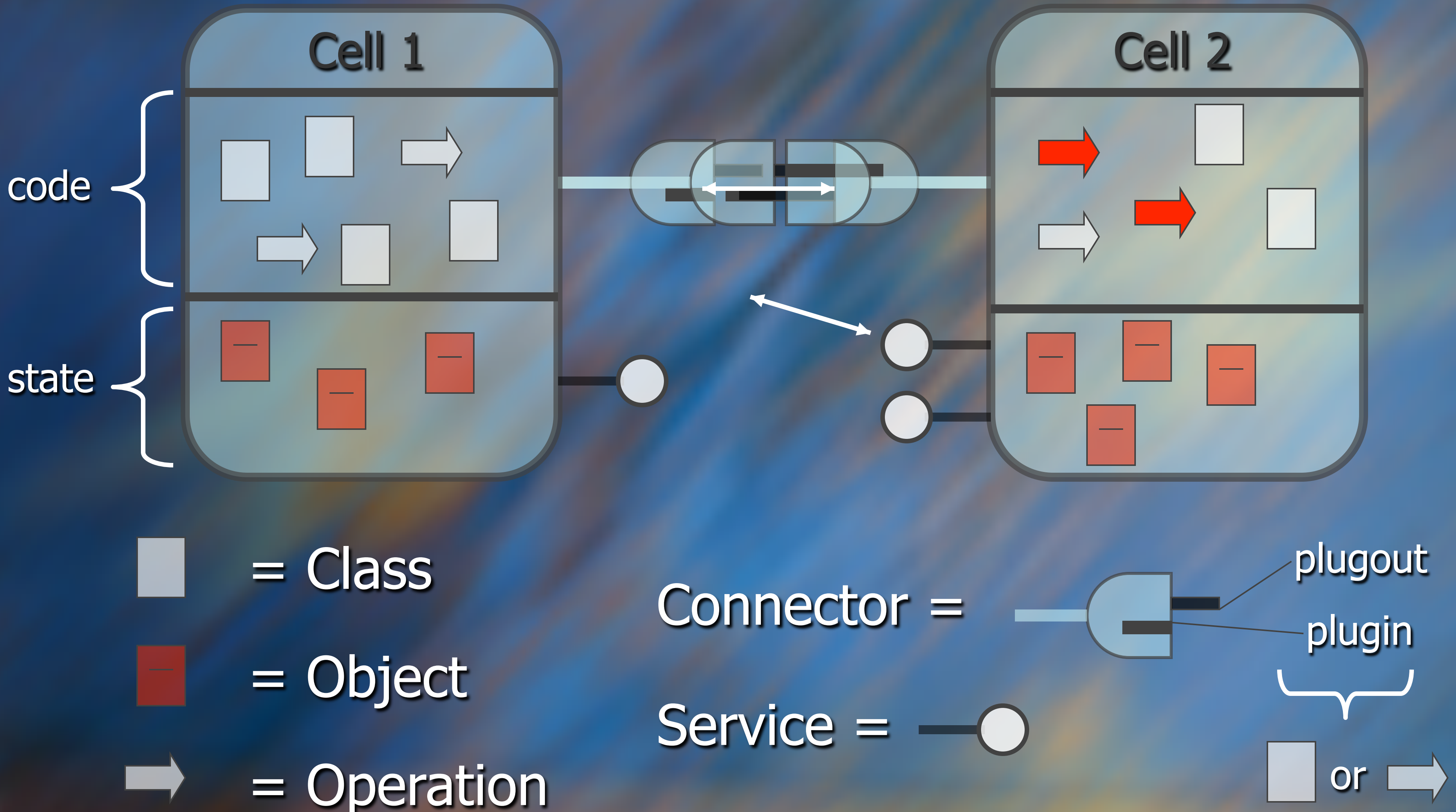
Our Proposal: *Cells*

- Deployable containers of objects and code
- Implicitly distributed
- *Connectors* for forming persistent links
  - Can be dynamically linked and unlinked
  - Can be linked locally or across the network
- Unifies notions of module and component
- May be dynamically loaded, unloaded, copied
- Serve as principals in a security architecture

Cells Unify Existing Technologies

Technology	Commonalities
Modules	Import and export, linking, namespaces
Components	Advertise services, support distribution
RMI	Invocation of remote cell services
Applets	Code shipment via cell shipment
Serialization	Cells serialize with their serialized objects
Mobile Objects	Cells move as Code+object packages
Object prototype	Cells are prototyped, cloned

Basic Cell Elements



□ = Class

🟥 = Object

⇨ = Operation

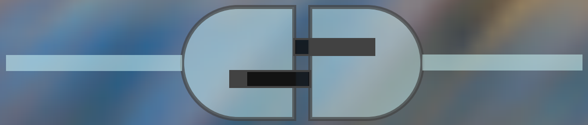
Connector =

Service =

The CVM (Cell Virtual Machine)

- “JVM/CLR for cells”
- Many CVMs concurrently running on the Internet
- Cells are loaded into a CVM
- Cells in different CVM’ s may communicate *transparently*, as if they were local
  - Invoke services on remote cells
  - Connect to remote cells
- CVM controlled by a distinguished President Cell

Cell Connectors

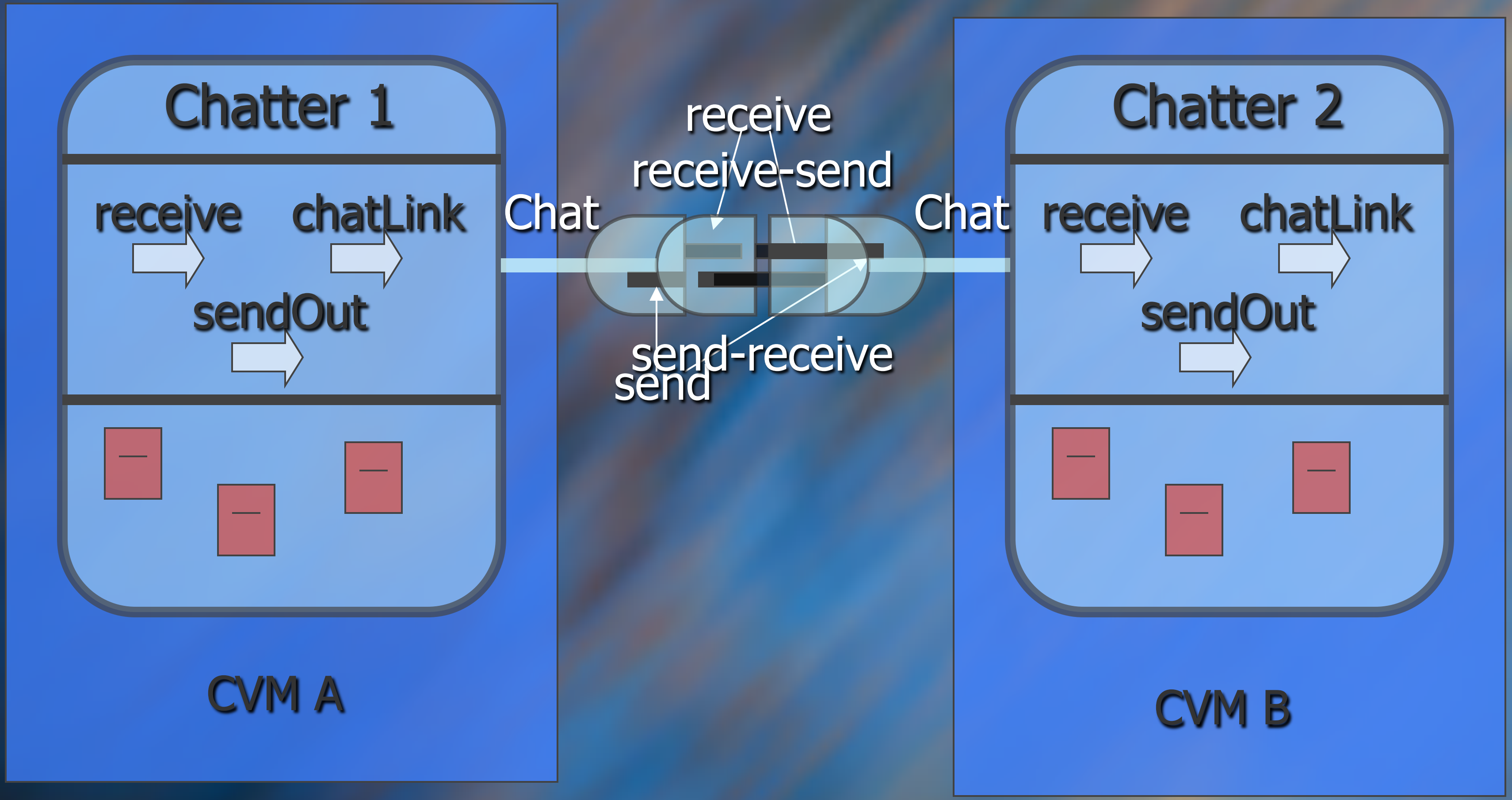


- Cells upon first loading have no connections
- Can connect and disconnect dynamically
- Multiple connections on a single connector possible when it is unambiguous

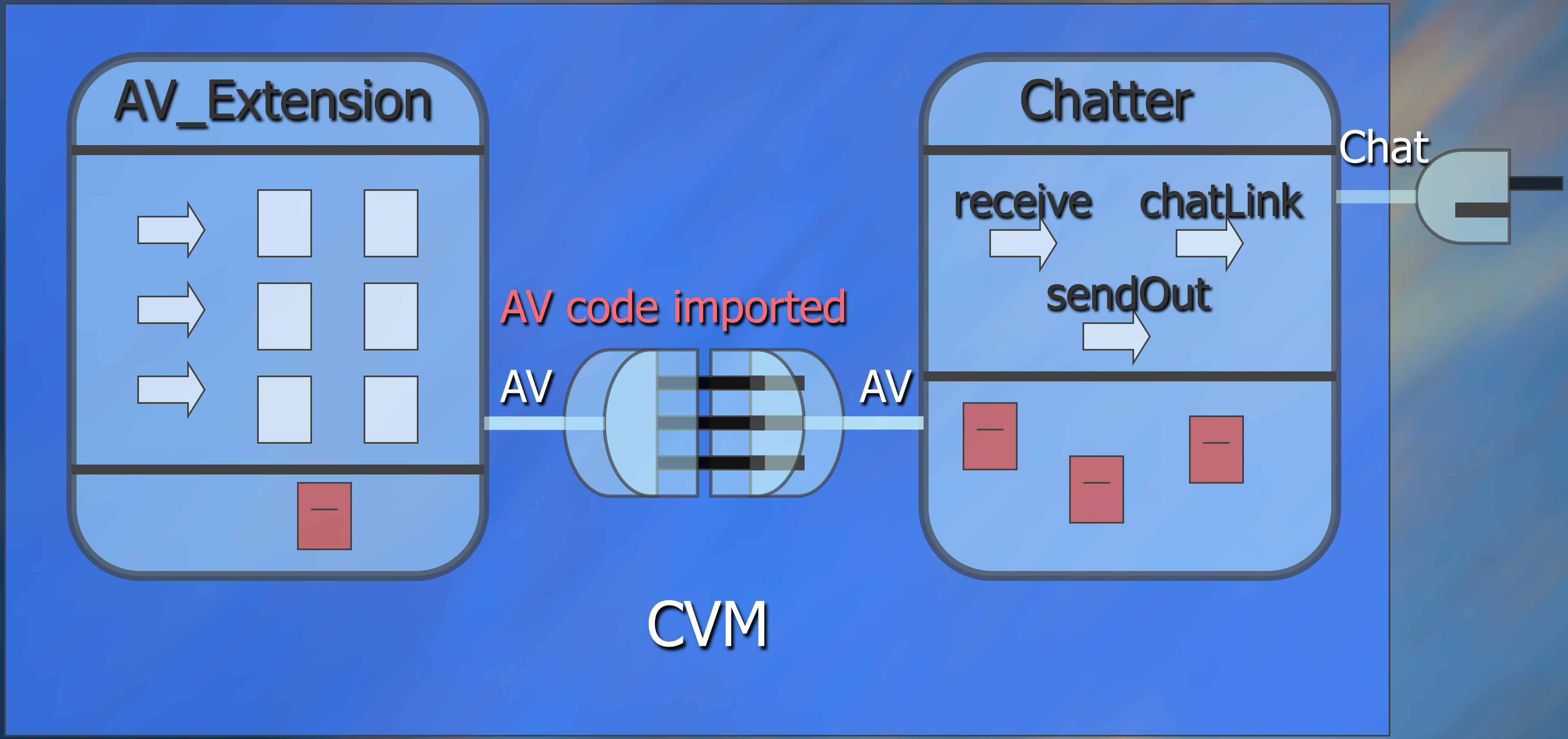
Cell connectors serve multiple purposes

1. Code import, *a la* packages/modules
   - ➔ Cell-module additionally has state associated with it
   - ➔ In this model all module linking is at *run-time*
2. Code plugin for dynamic extensibility
3. Persistent (network) data connections

Chatter Example



Chatter with AV_Extension



JCells

- New cell-based programming language
- 90% the same as Java in syntax and semantics
- Java concepts replaced: RMI, ClassLoader, CLASSPATH, applet, package, security arch., …

Implemented by compilation to Java

- CVM (Cell Virtual Machine) implemented by JVM
- Basic features now implemented
- Full implementation in progress

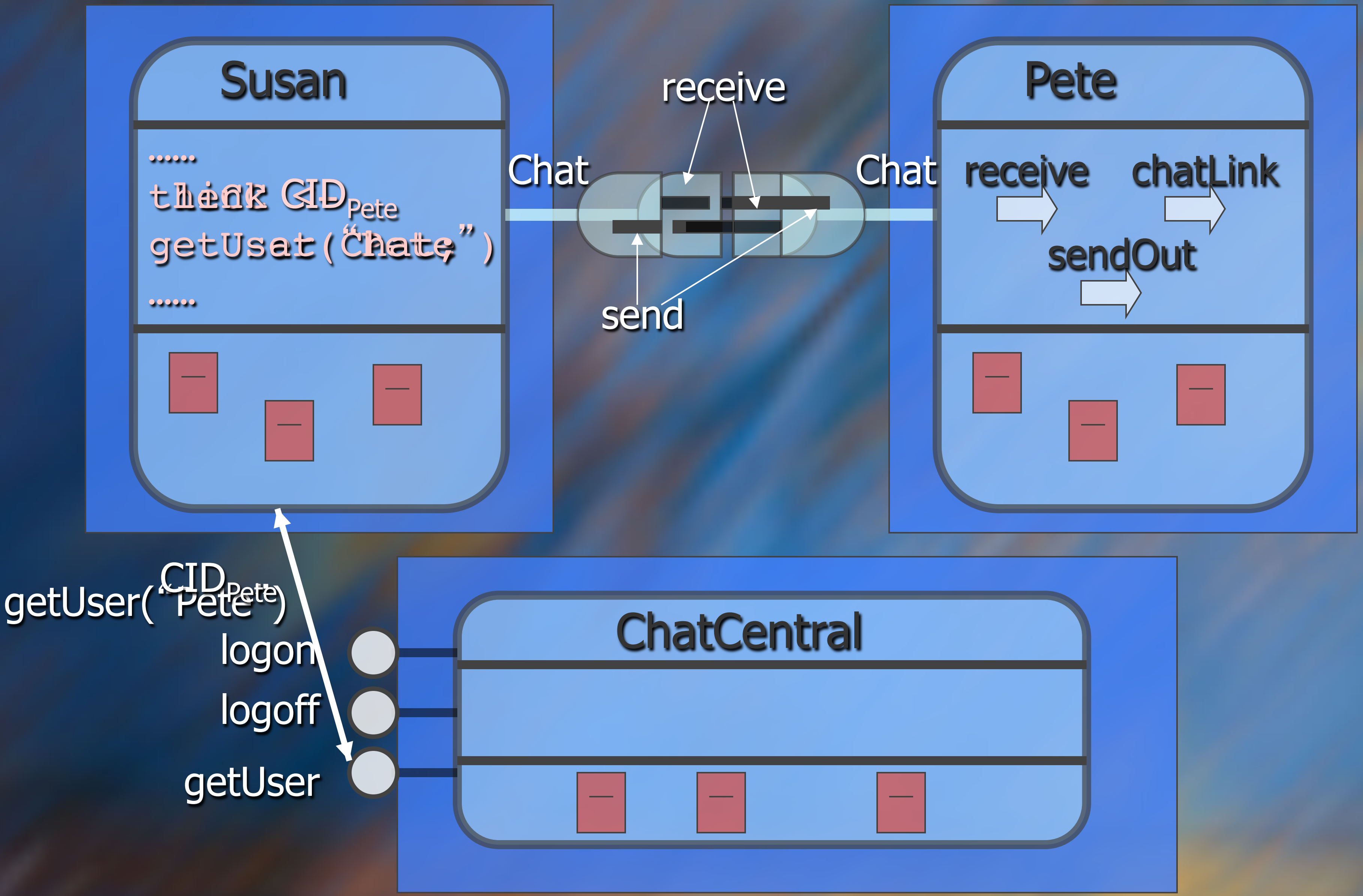
JCells Chatter Code Fragment

```
cell Chatter
{ … // Type declarations, etc
 connector Chat { plugins  { send … }
                  plugouts { receive … } };
 void linkToChatter(cell Chatter other) {
  … link other at Chat
       [receive -> send, send <- receive]; …
 void unlinkFromChatter()
  … unlink at Chat; …
 void sendMessage(string m) { … send(m); … }
}
```

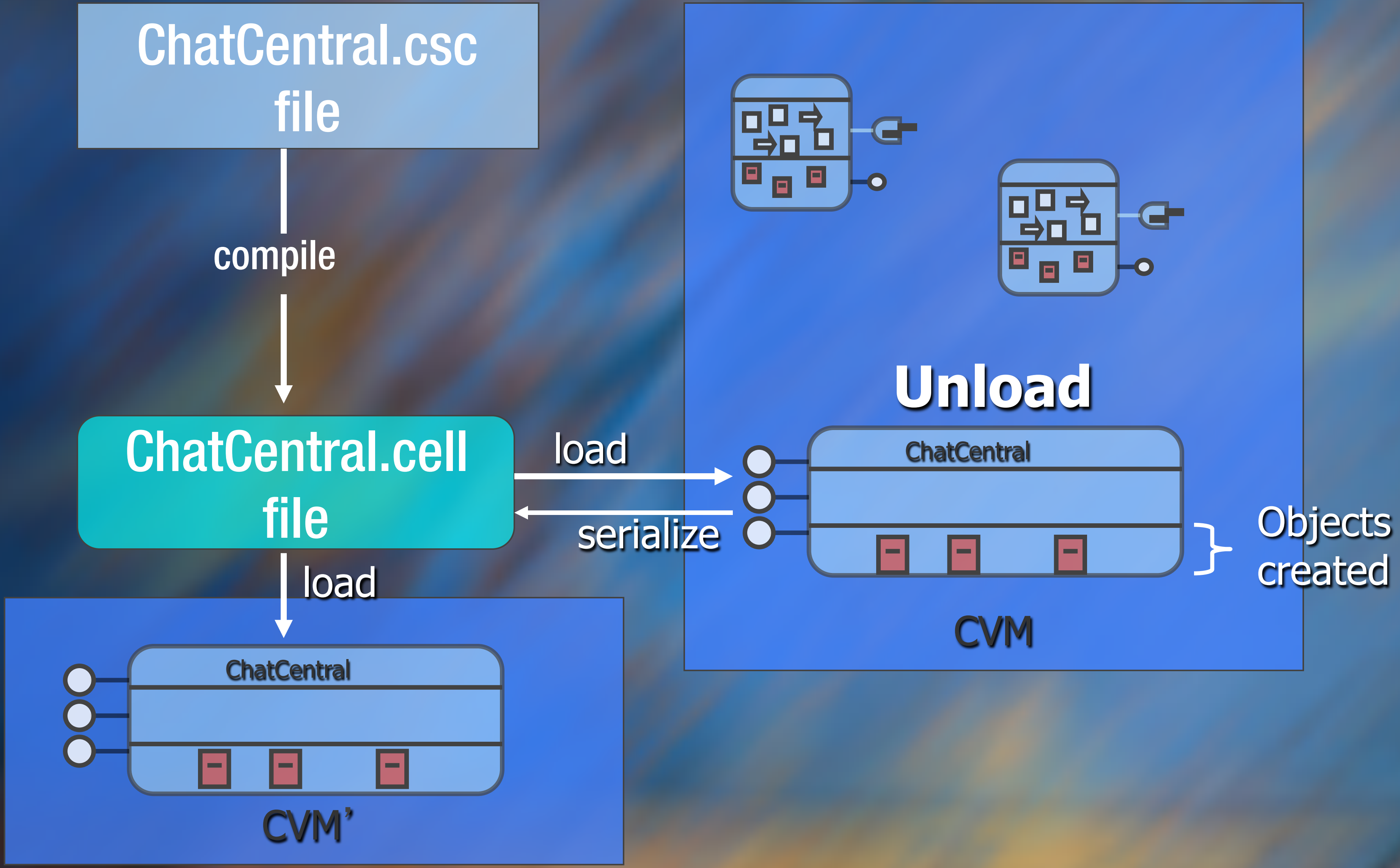
Cell identifiers (CID’ s)

- CID is a Universal (string) name for a cell
  - With a CID alone you can address a cell that could be anywhere
- Cells transparently addressable by CID after moving
  - Implemented similar to snail mail forwarding
- No two cells anywhere can share a CID

Universality of CID’s



Cell File States



ChatCentral.csc file

compile

ChatCentral.cell file

load

serialize

load

Unload

ChatCentral

Objects created

CVM

ChatCentral

CVM’

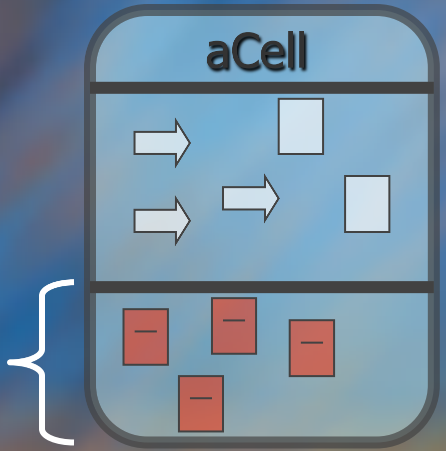
Cell (Re-)deployment

- Cell source code in .csc files
- Cells can be in two states
  1. Cell active in a CVM, with fixed identity CID
  2. Serialized cell in .cell files, with (or without) CID
- .csc files compile to .cell files
  - → These .cell’s are anonymous (no CID)
  - → They own no objects
- Loading and CID’s
  - → Anonymous .cell’s get a CID upon loading

Cells and their objects

- Every object in a CVM is owned by a cell
- Default policy

  "you own the objects your code creates"
- Cells serialize with their objects
- Modulated object references survive cell movement



Copying and Moving Cells

- Serializing a cell
  - Its classes, its objects and CID serialized
  - ".cell file" produced
- This .cell file can then be loaded into another CVM
- Move is serialize-unload-(transfer .cell file)-load

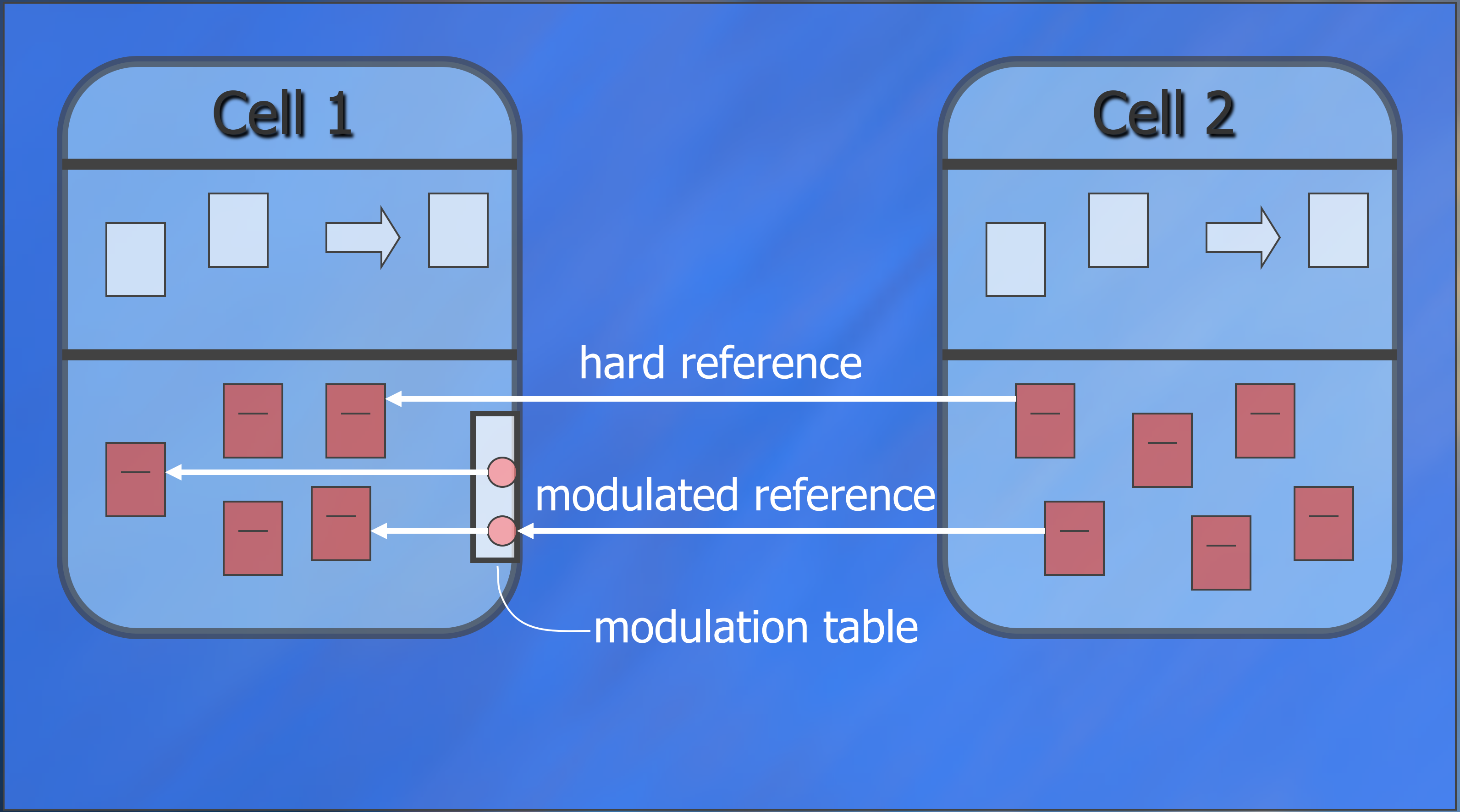
Distribution

- Transparency of distribution
  - Differs from RMI where parameters *implicitly* copied if object is remote
- Not all services/connectors support distributed use
  - Parameters must all be passed by copy (or modulated reference - forthcoming)
  - Classes cannot be plugged in across the network
- Cell movement across the network is supported

Object References and Parameters

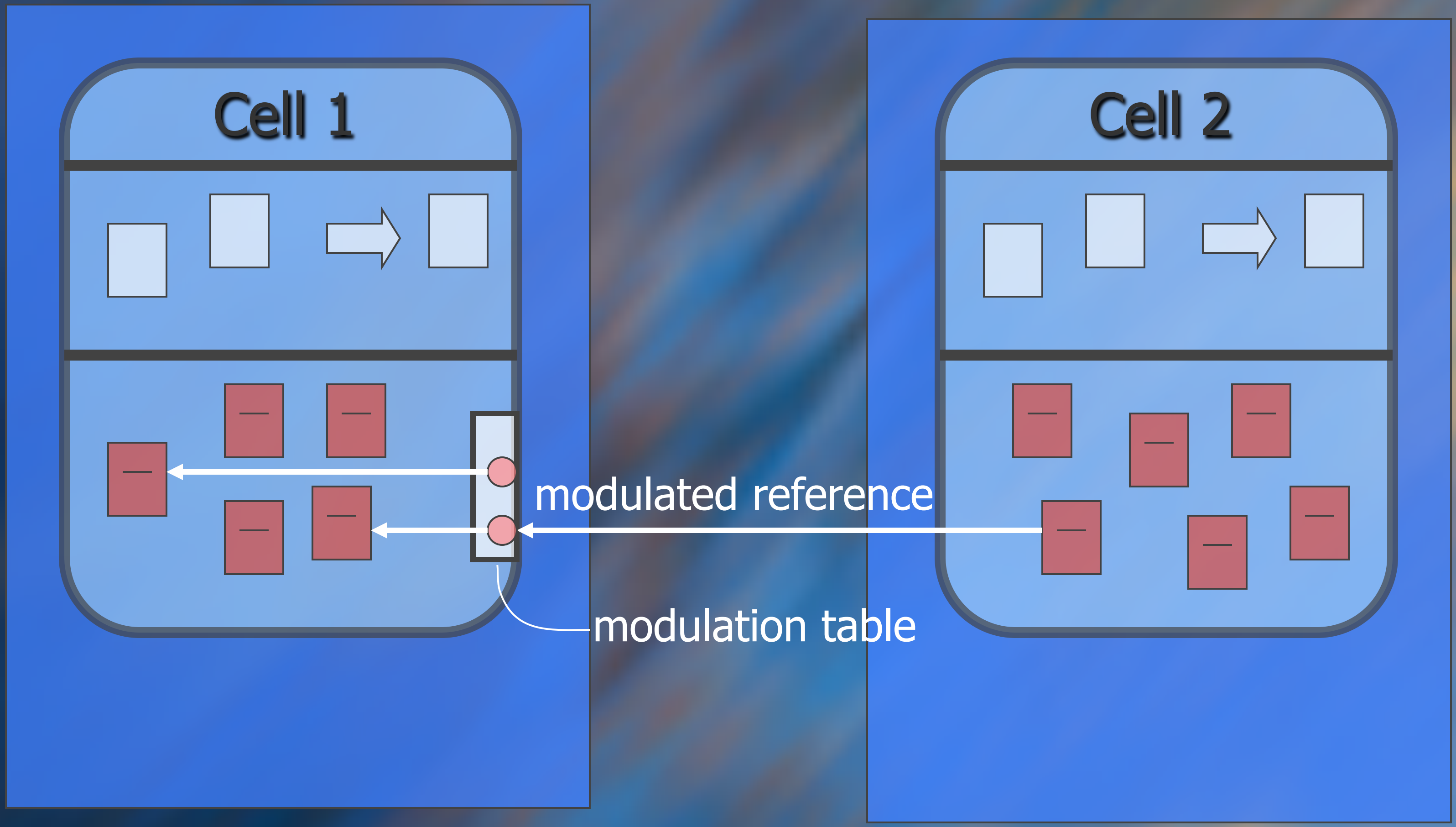
- *Hard* references
  - Your standard object reference
  - Local (intra-CVM) only; but inter-cell allowed
- *Modulated* references
  - Used for more tightly-coupled interactions between cells
  - Both intra-CVM and inter-CVM (implemented via a proxy)
  - Can be dynamically revoked (e.g. revoke at disconnect time)
- Parameter passing
  - Intra-CVM, no restrictions
  - Inter-CVM, cannot pass hard references
  - Explicit `copy` parameter syntax for inter-CVM case

Modulated vs hard references



modulated reference invalidated

Inter-CVM modulation



No inter-CVM hard references

Cell Types

- Strongly typed
  - No dynamic checks except cast
- Cell references have cell types

```
cell Chatter myChatter;
```

- Cell types in Java spirit except structural subtyping on cells for more universality
  - Connector can have unused plugouts

New Cell Security Architecture

[FCS, Copenhagen, July 2002]

- Each cell is a principal with a public/private key
- Access control decisions can be *cell-based*
  - *“I only will connect on my privChat connector with Joe or Sue”*
- Uses SDSI/SPKI Internet standard, RFC2693
  - Groups, authorization certificates, revocation, delegation
- Cells can declare they will not share objects
- Additional capability layer
  - without an initial capability to a cell, can’t even try connection

Thorny Issues Galore

- If superclass code makes an object, who owns it, super or subclass’ cell? (super’ s)
- When a cell is serialized, it could have hard references to objects it doesn’ t own (null them)
- When a plugged-in class is unplugged, what happens to live objects of that class?
    - (They become *zombies* – unusable)
- What if cell is unloaded when another cell is plugging in one of its classes (disallow unload)

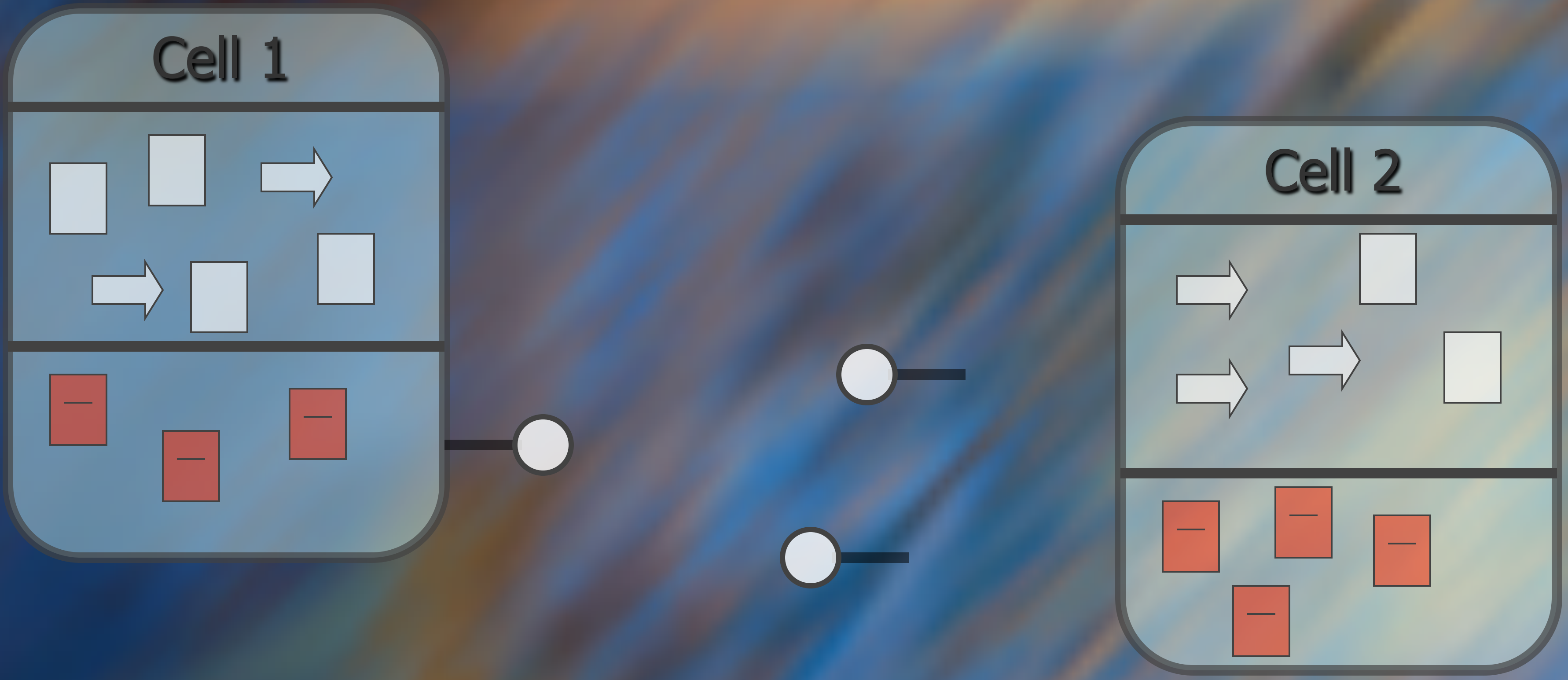
Related Work

- Technologies partly incorporated
  - Java
  - Modules: Modula-3, Units/Jiazzi, . . .
  - Components: Corba, COM, . . .
  - Prototype-based languages: Self, . . .
- JavaSeal: passive seal = .cell; seals own objects; …
- J-Kernel
- XML/SOAP/UDDI/WSDL School

jcells.org

Cells address Internet needs

Internet Need	*Cell Solution*
Code-level interaction	Link via connectors
Call-level interaction	Service invocation
Components move around	Cells can be copied/moved
Cross-network interaction	Supported by cells
Cross-component class inheritance (e.g., applets)	Supported, between locally linked cells
Different political entities	Cell-level security, degree of cell isolation controllable
Political situation volatile	Unlink supported, affects modulated references

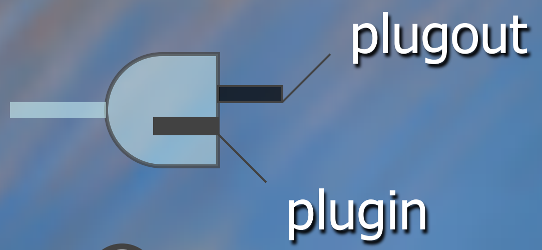


□ = Class

🟥 = Object

➡ = Operation

Connector:



Service: