Croquet - Hedgehog

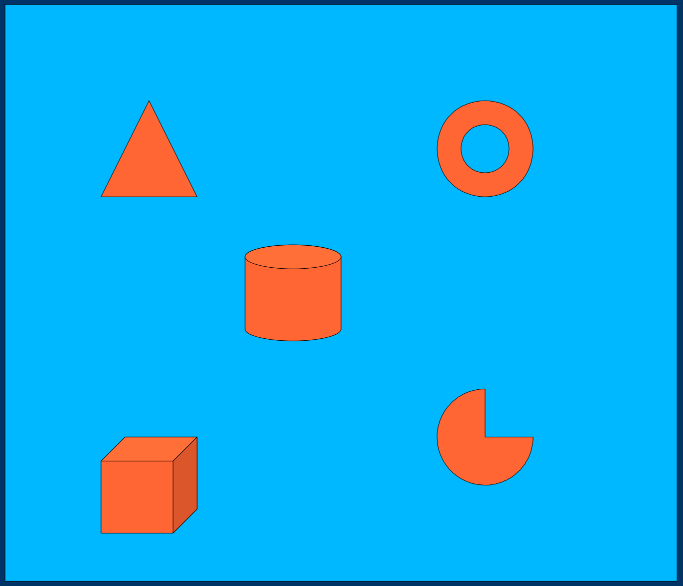
David A. Smith
Andreas Raab
David P. Reed
Alan Kay

Hedgehog Architecture

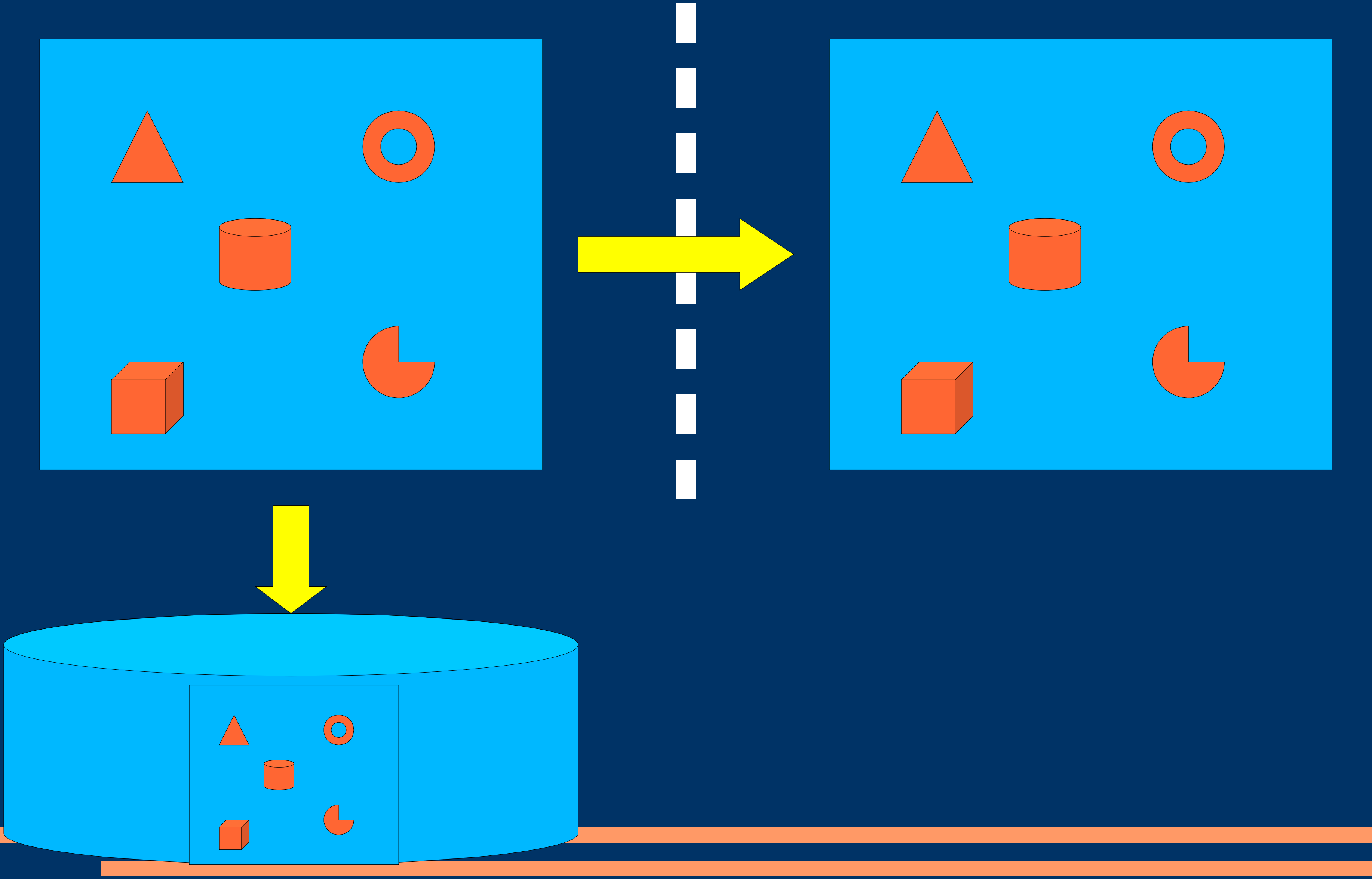
- Islands
- Replicated Islands
- Routers/Controllers and Genuine Time Based Replication
- Joining, participating
- Capability Facets
- Overlay Portals
- Ghost objects
- Reference objects
- Graphics and Transform Support
- Tweak

Islands

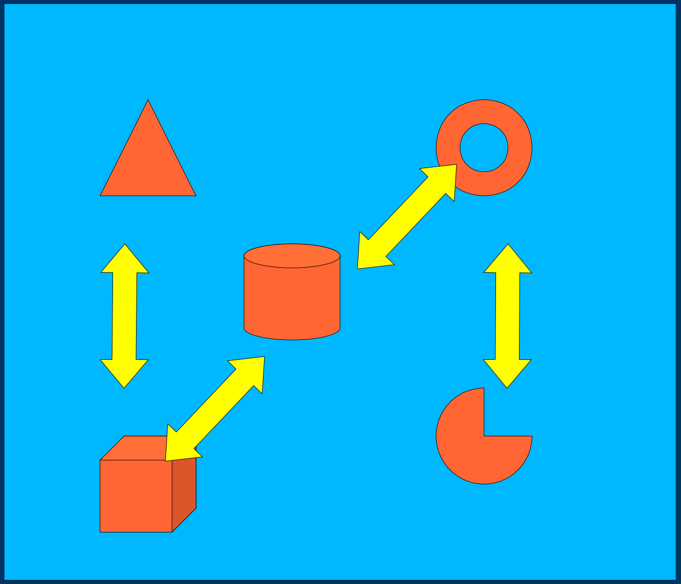
Islands are “safe” containers



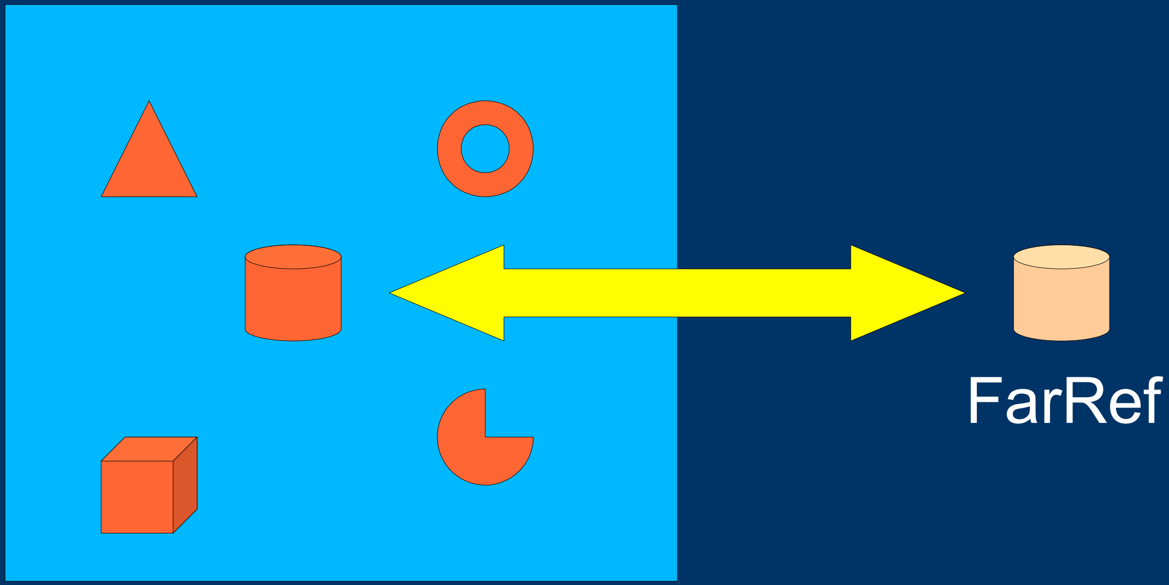
They can be easily saved and duplicated



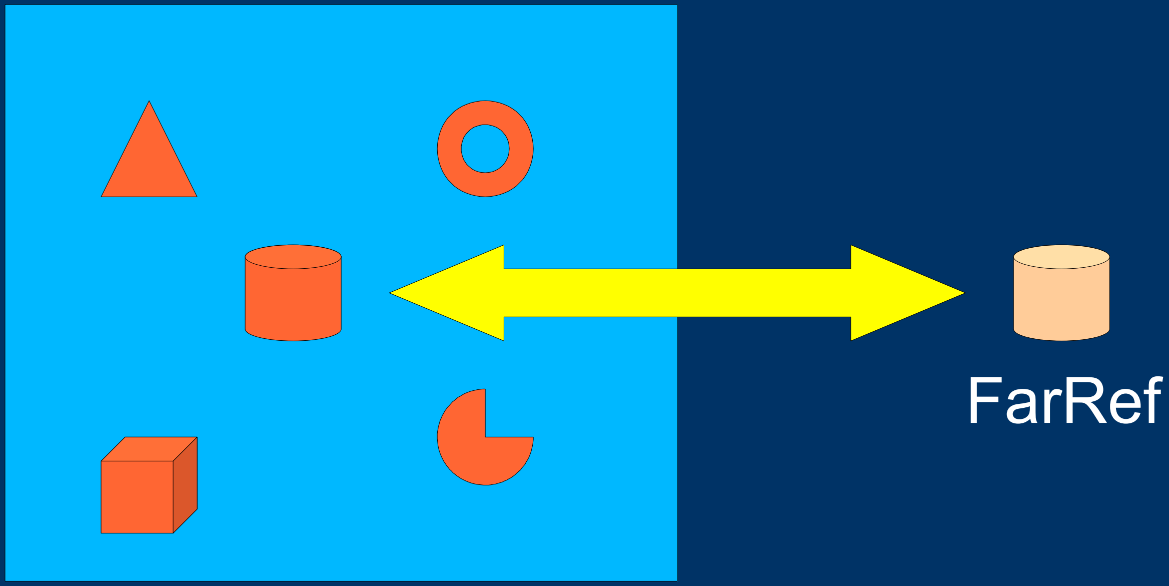
Objects can send messages to other objects in islands normally



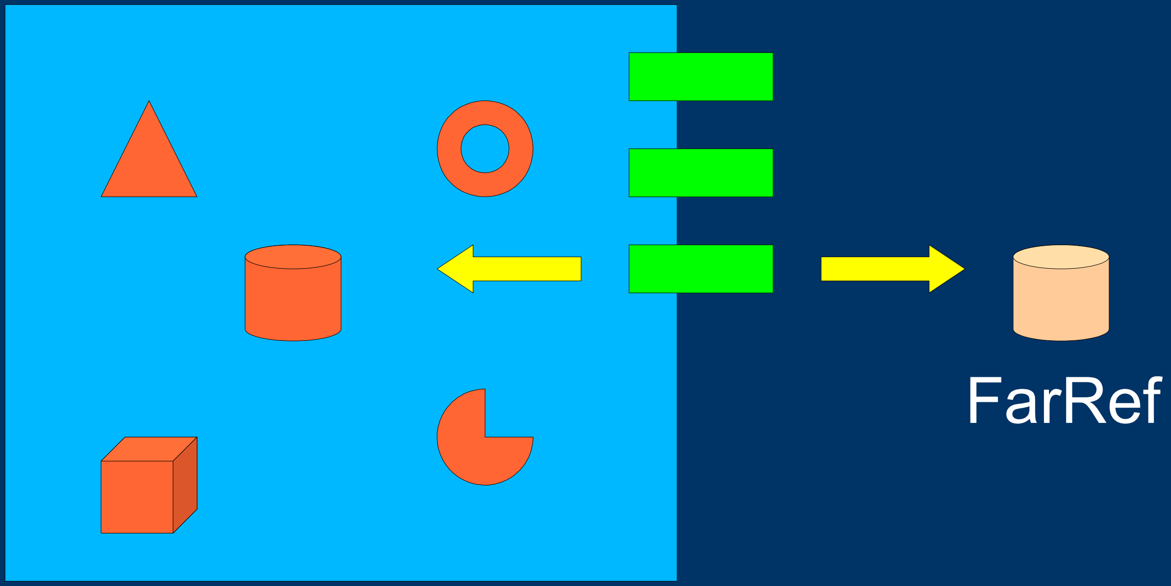
Objects in islands can only be accessed by reference externally



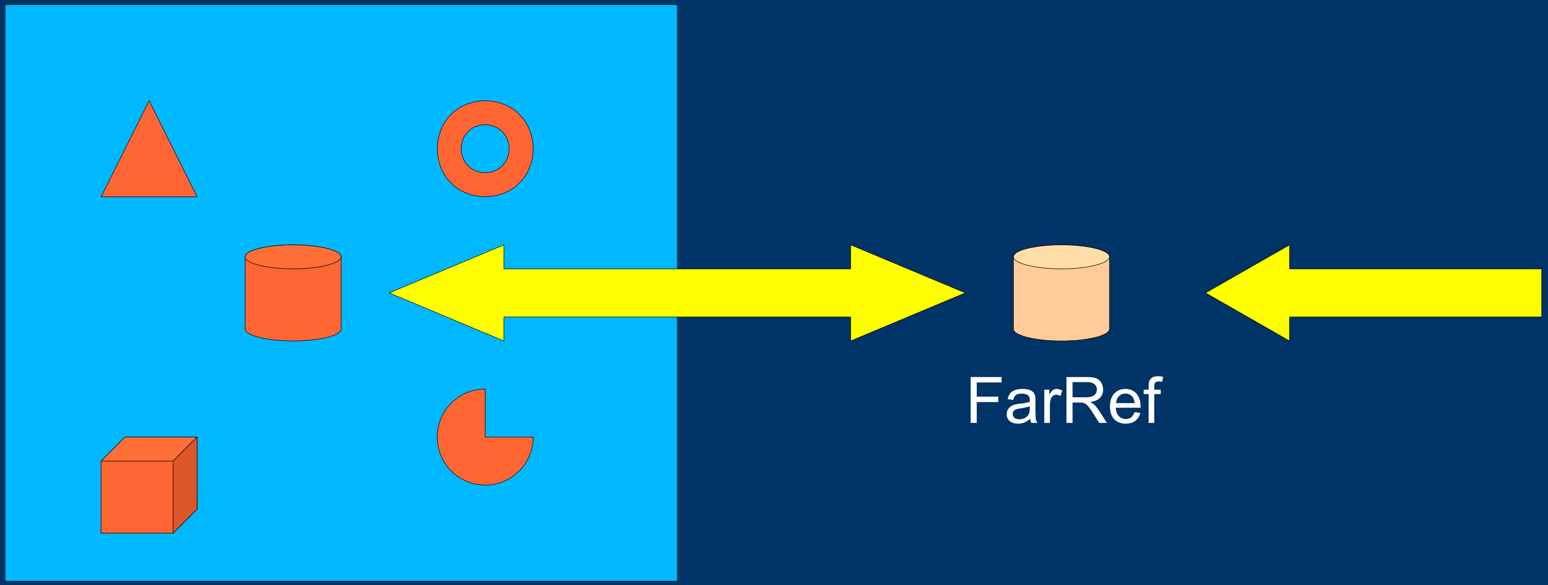
FarRefs are created whenever an object is externally accessed



The Island maintains a list of named objects that can be accessed externally

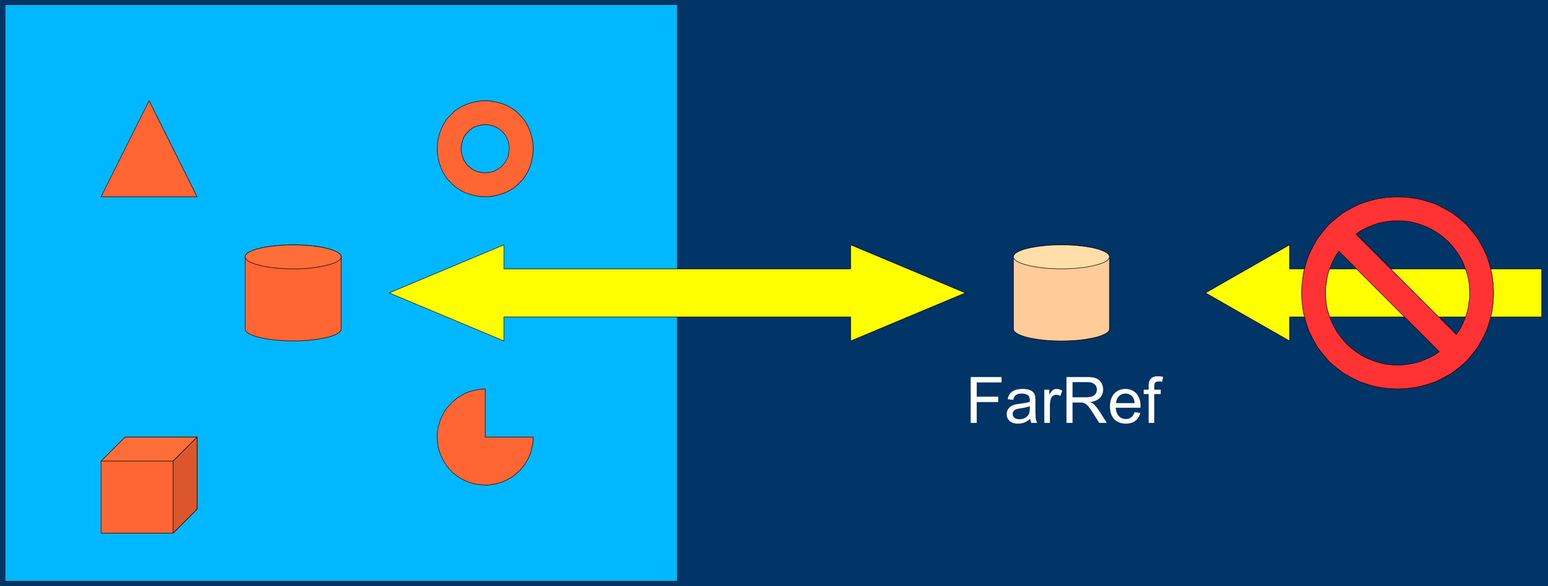


Messages are sent indirectly via the FarRef



```
farRef send:[obj | obj sendMessage.].
```

BUT: You should never have to do this.

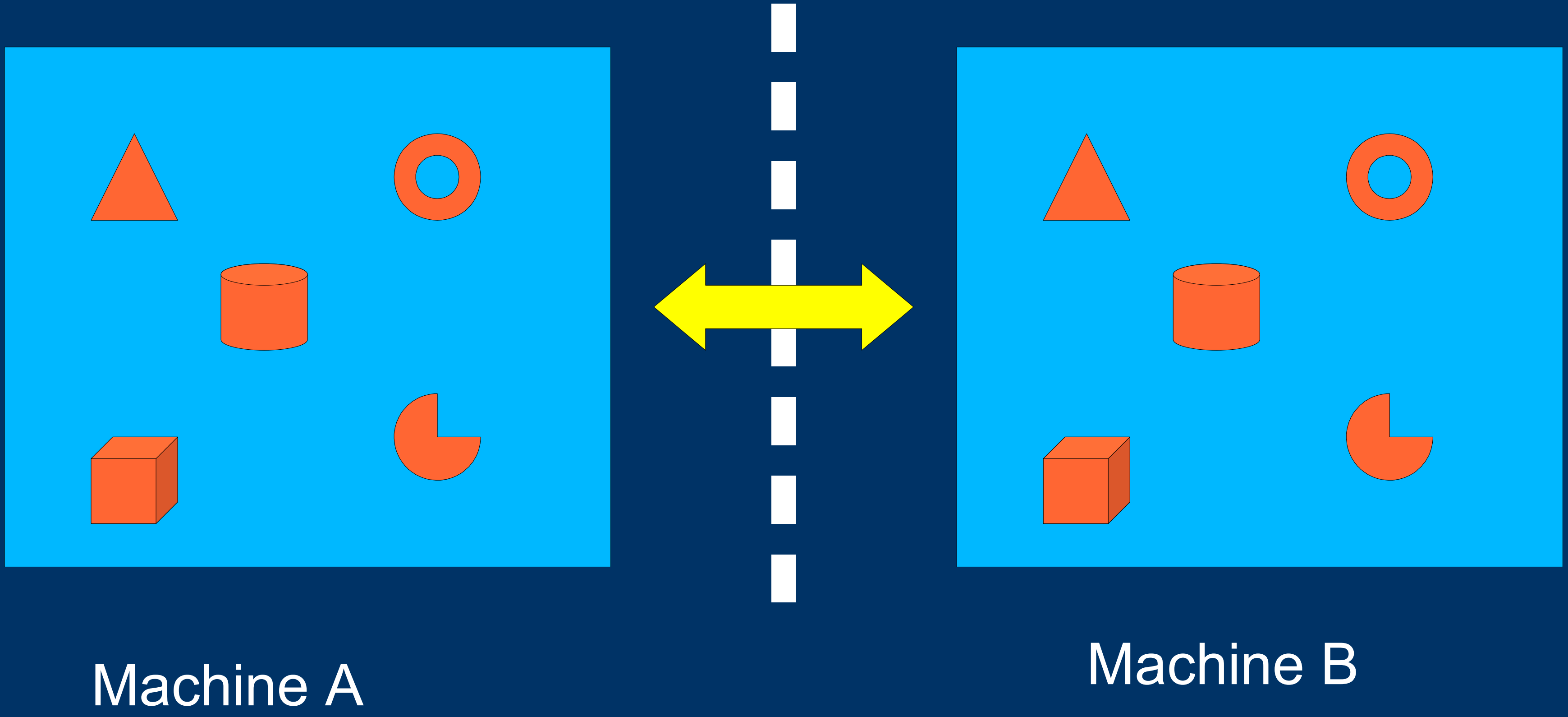


```
farRef send:[obj | obj sendMessage.].
```

Replicated Islands

Meta is Dead

Replicated Islands



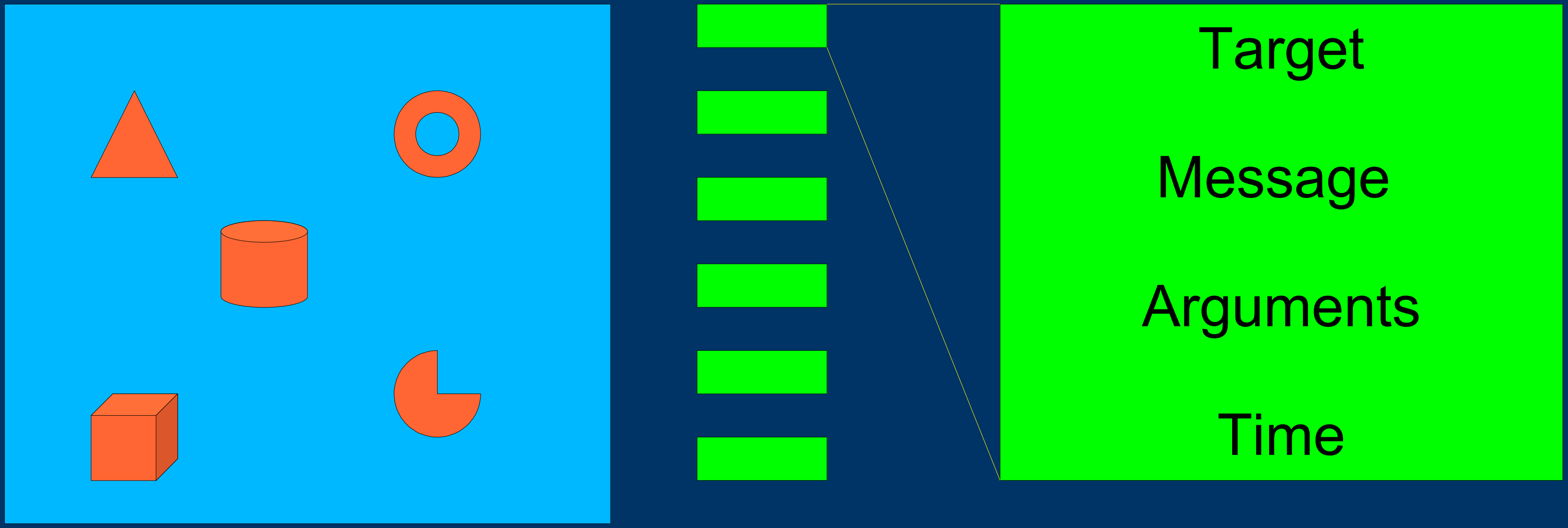
Replicated Islands

- Deterministically Equivalent
- Islands replicated via checkpoint mechanism
- Internal Future messages implicitly replicated
- External Future messages explicitly replicated
- External non-replicated messages VERY bad
- New objects: Routers and Controllers
- Meta no longer required, because Island contents and all external messages are guaranteed to be properly replicated.

Timing is Everything!

- External messages must be executed in the same order and at the same time in all replicated islands.
- Internal messages are executed deterministically, as long as island structure remains identical – we have identical results.
- But how?

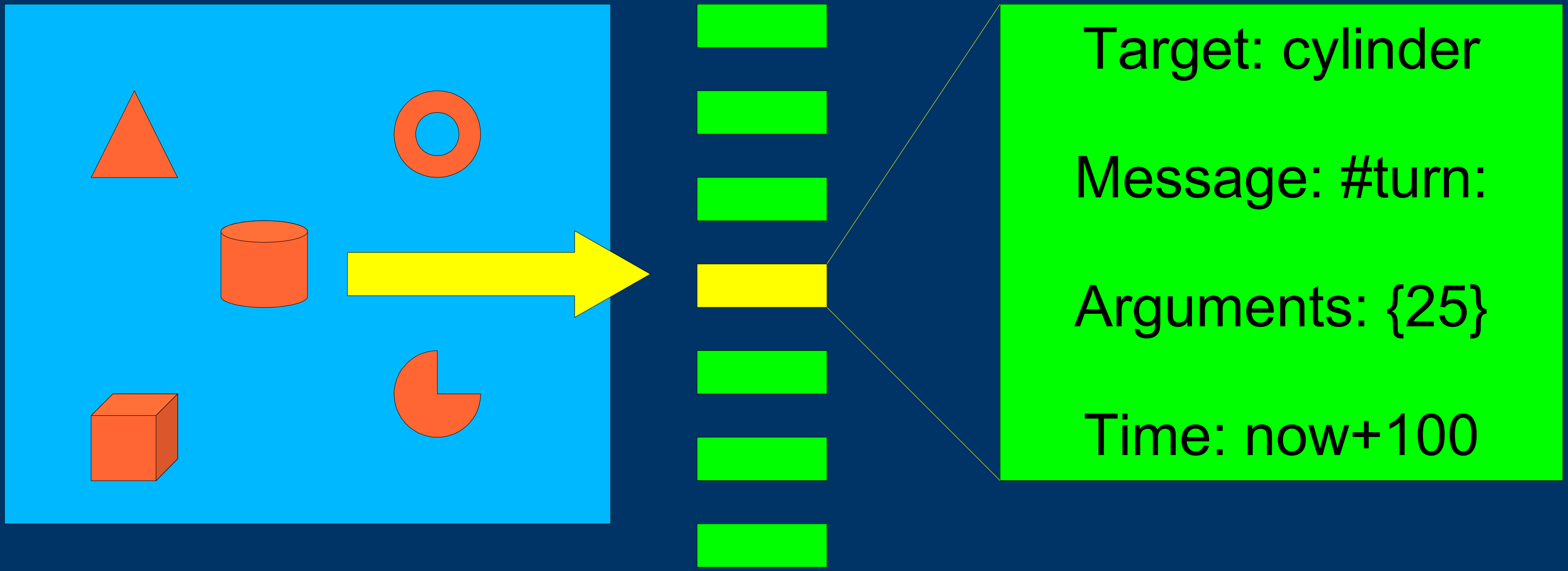
Island's View of Time is defined only by message order!



Machine A

Message Queue Sorted by Time

New message inserted with #future:



Machine A

Message Queue Sorted by Time

(self future:100) turn: 25.

An example (recursion in time):

```
ACylinder>>#aMessage: arg

" this is a typical pattern for performing
redundant tasks, such as animations "

self doSomethingWith: arg.
arg> 0 ifTrue:[
        (self  future:100)aMessage:arg-1.].
```

Routers, Controllers and Genuine Time Based Replication

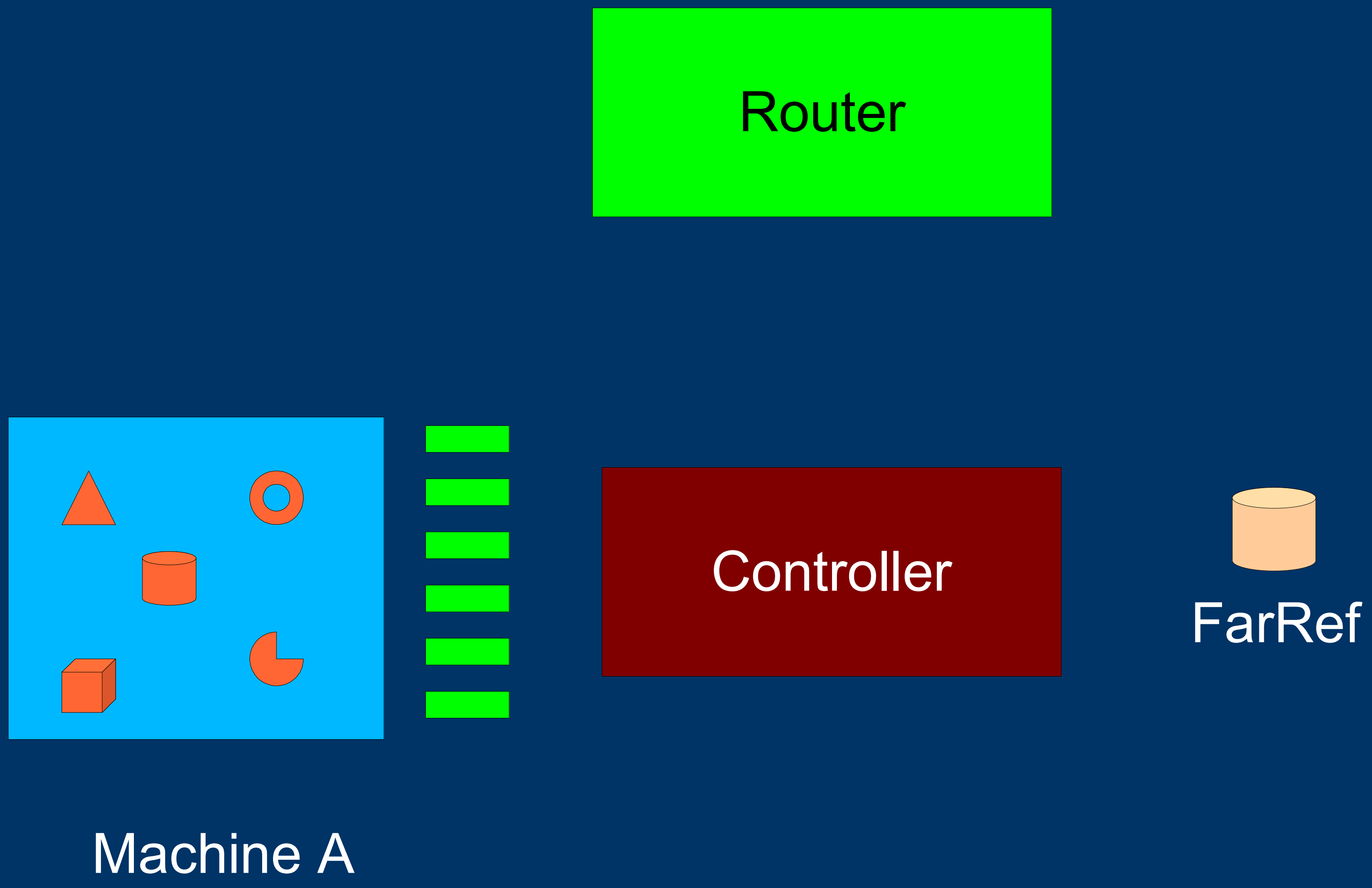
The Router

- Acts as the clock for the replicated Islands
- Determines when an external message is actually executed for all Islands
- Sends heartbeat messages to move time forward
- The Island Creator owns the Router by default
- It is possible to have a number of routers that share the ability to grant message send requests.

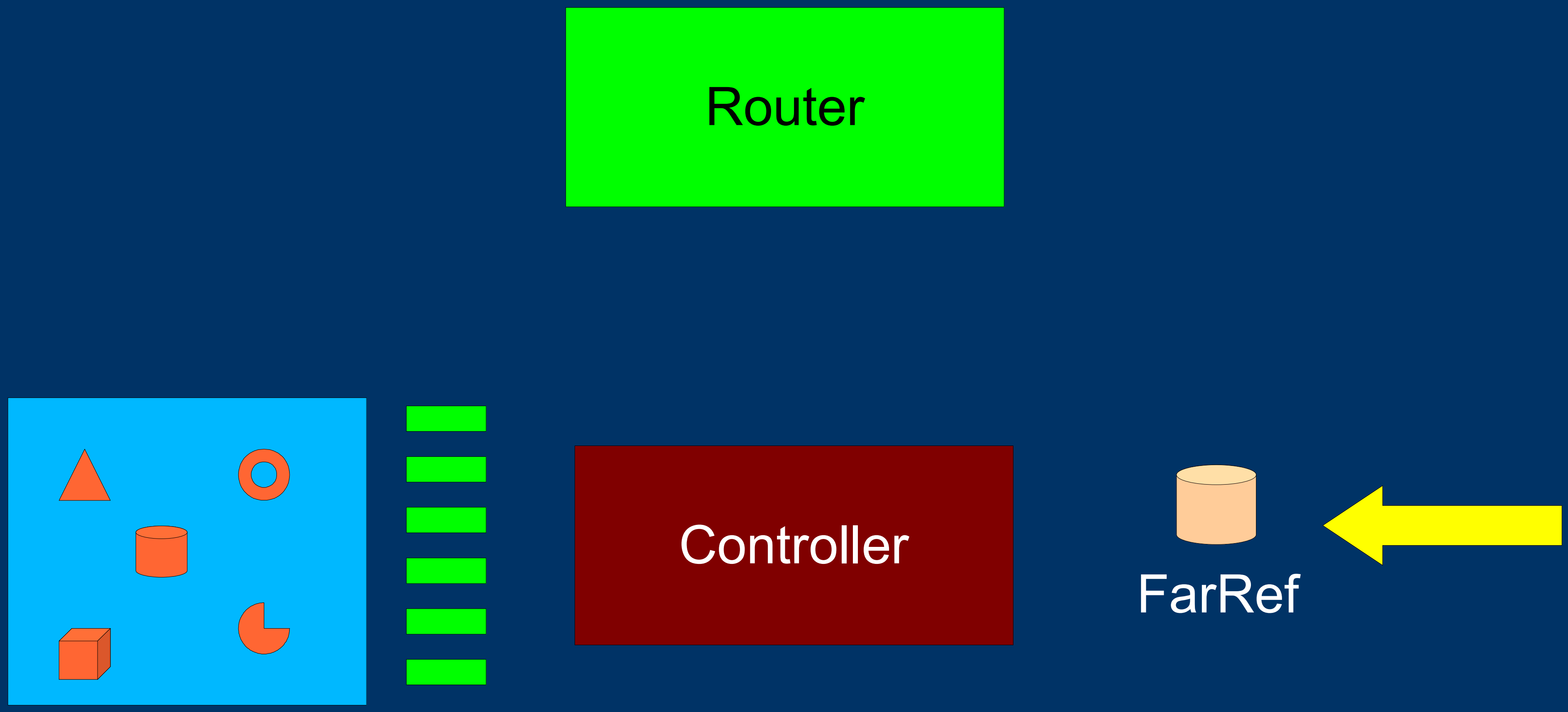
The Controller

- Manages the interface between the island and the router
- Manages the message queue
- Non-replicated part of island/controller pair
- Can exist without an island, acting as a proto-island until the real island is either created or duplicated.

The Router/Controller



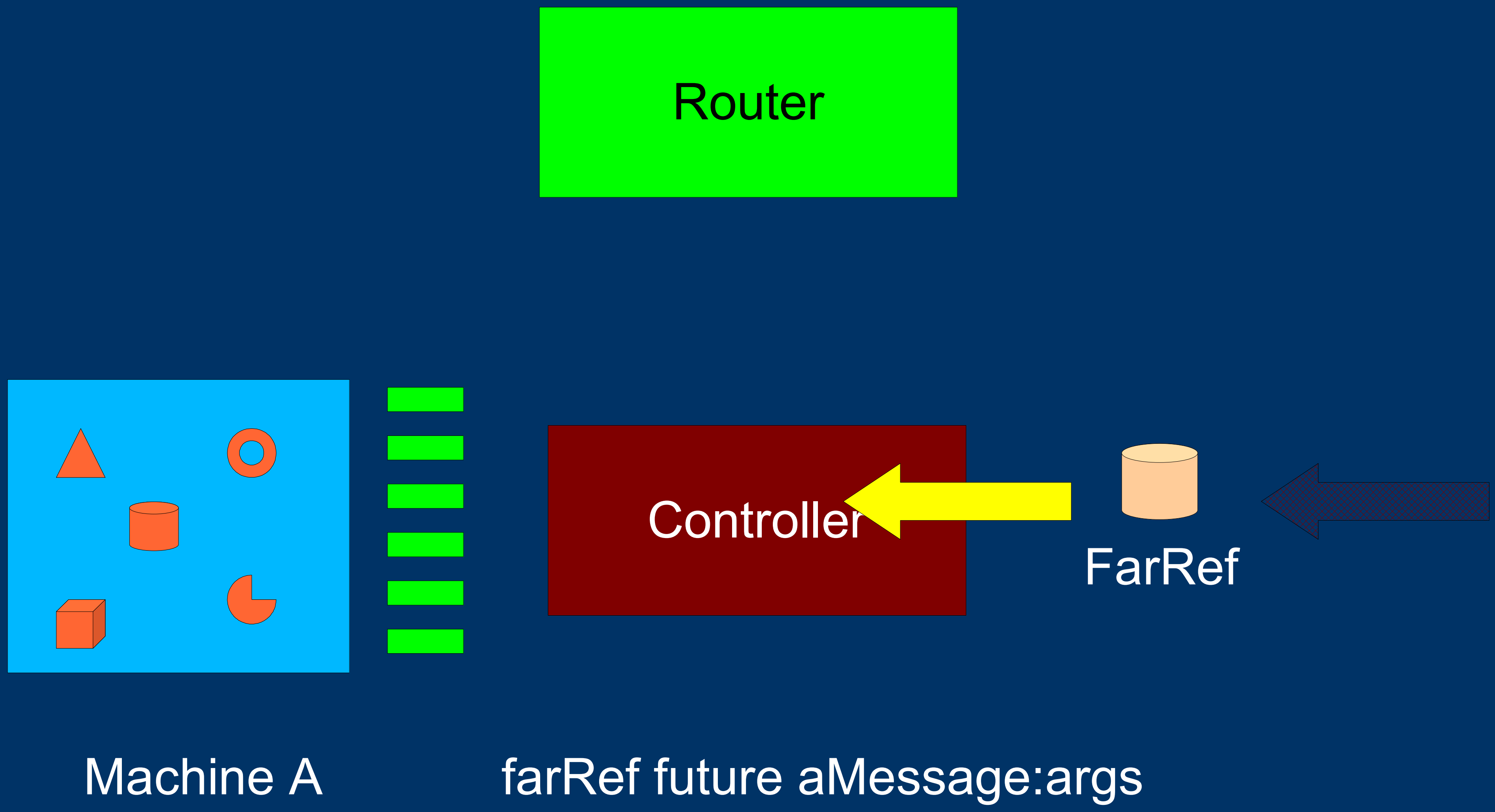
Message sent to farRef – no time is specified



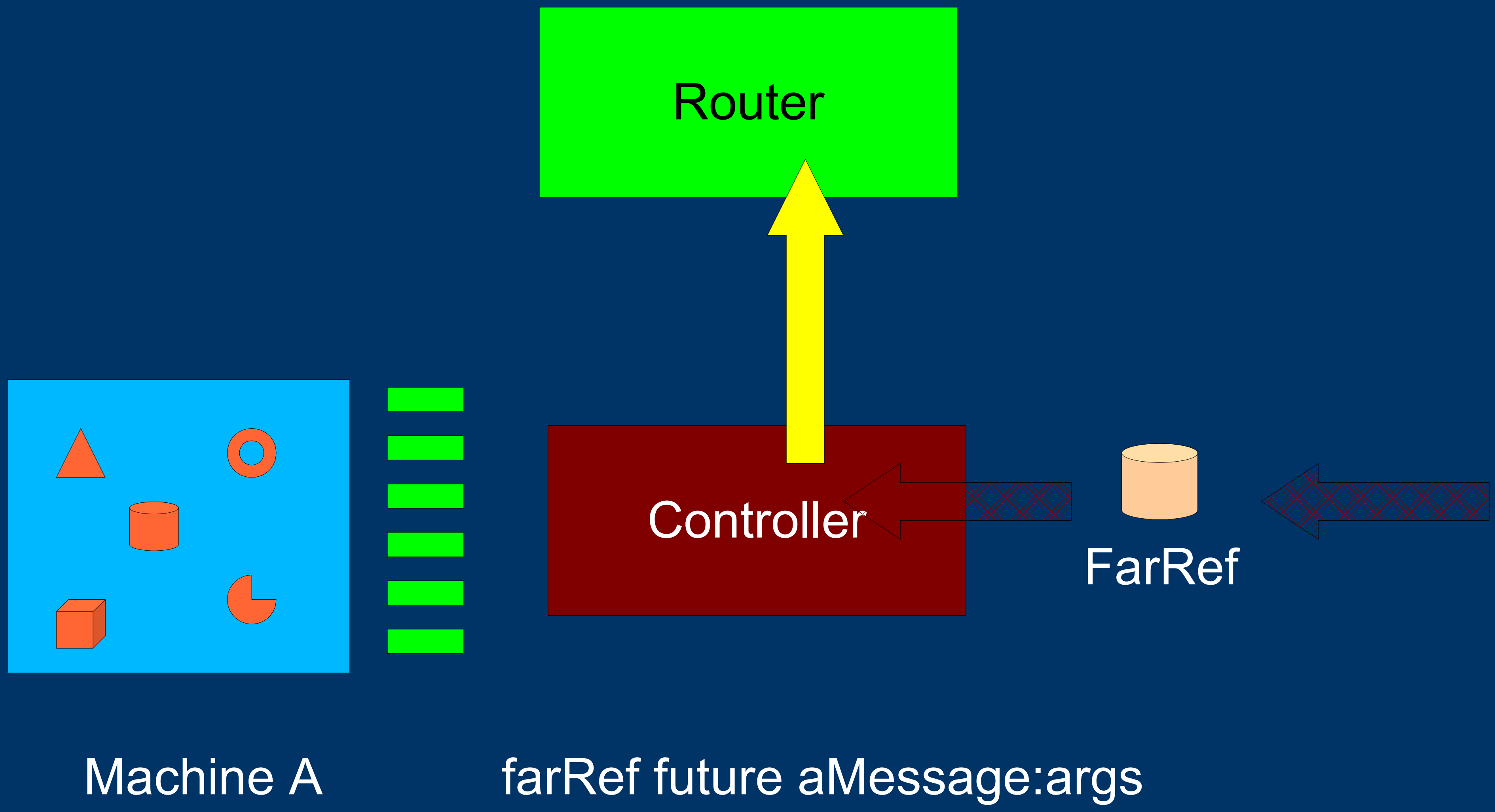
Machine A

farRef future aMessage:args

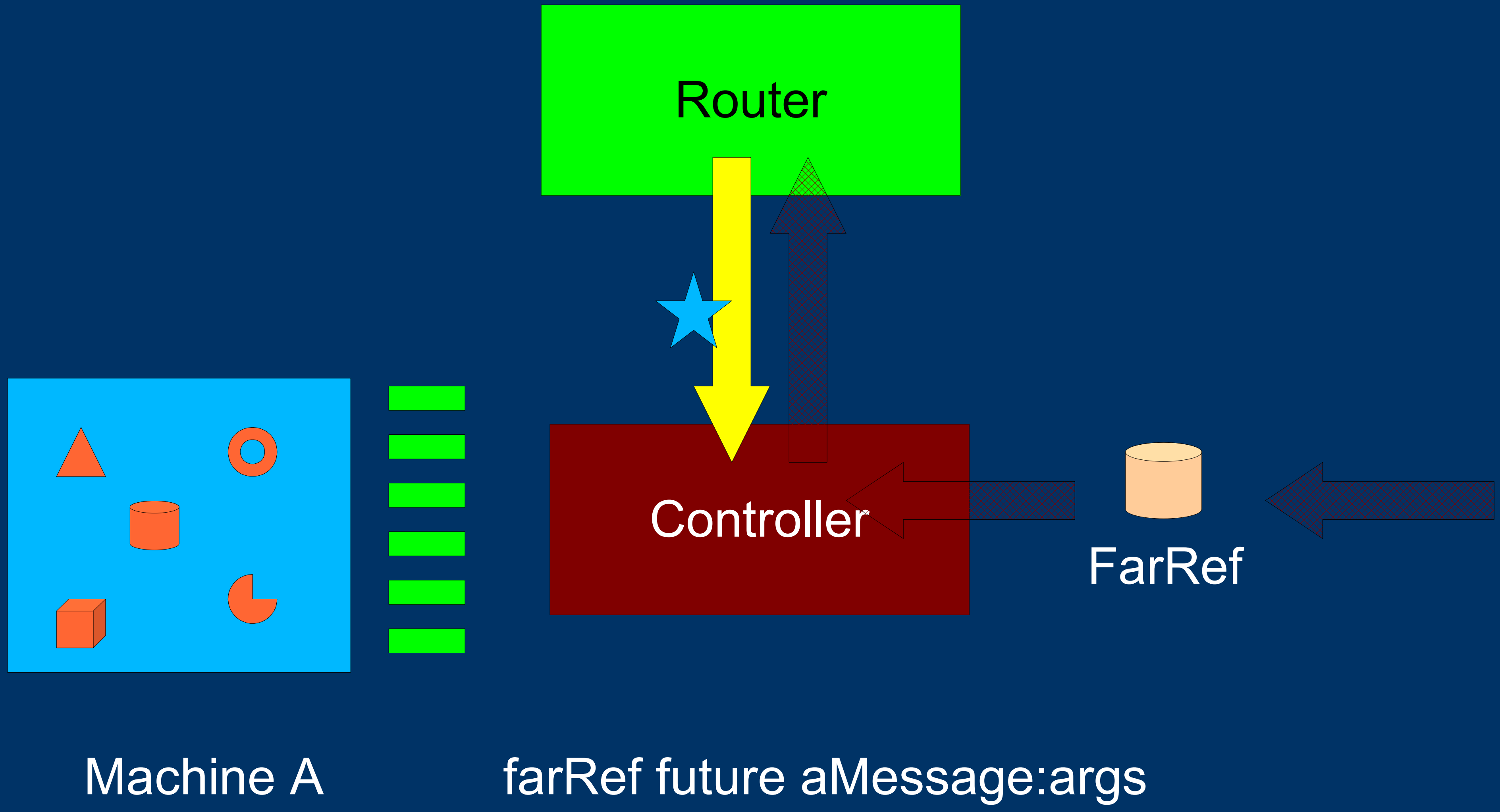
farRef forwards to controller



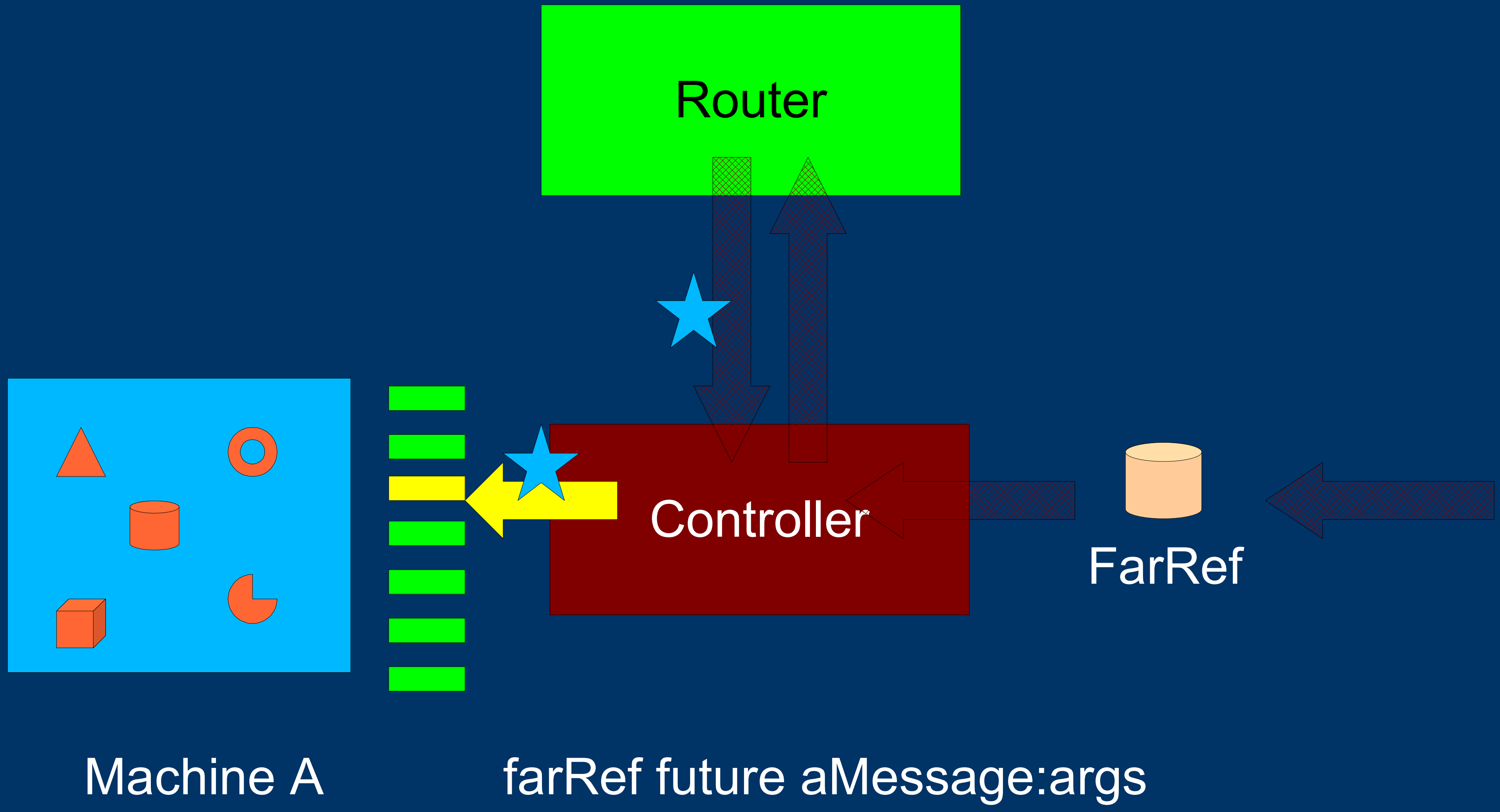
Controller forwards to Router



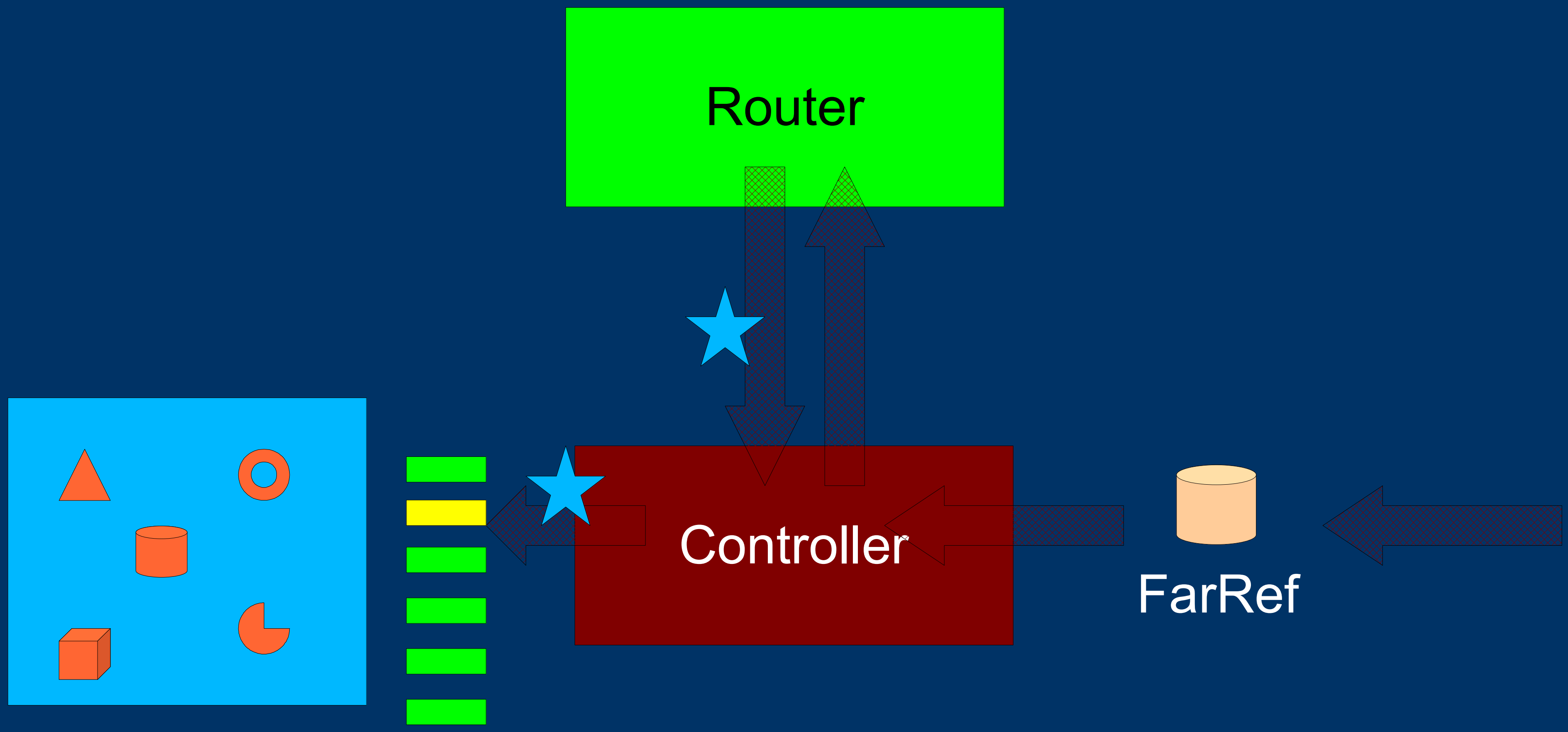
Router adds time stamp (and enumeration), forwards back to controller



Controller forwards time-stamped message to add to message queue.



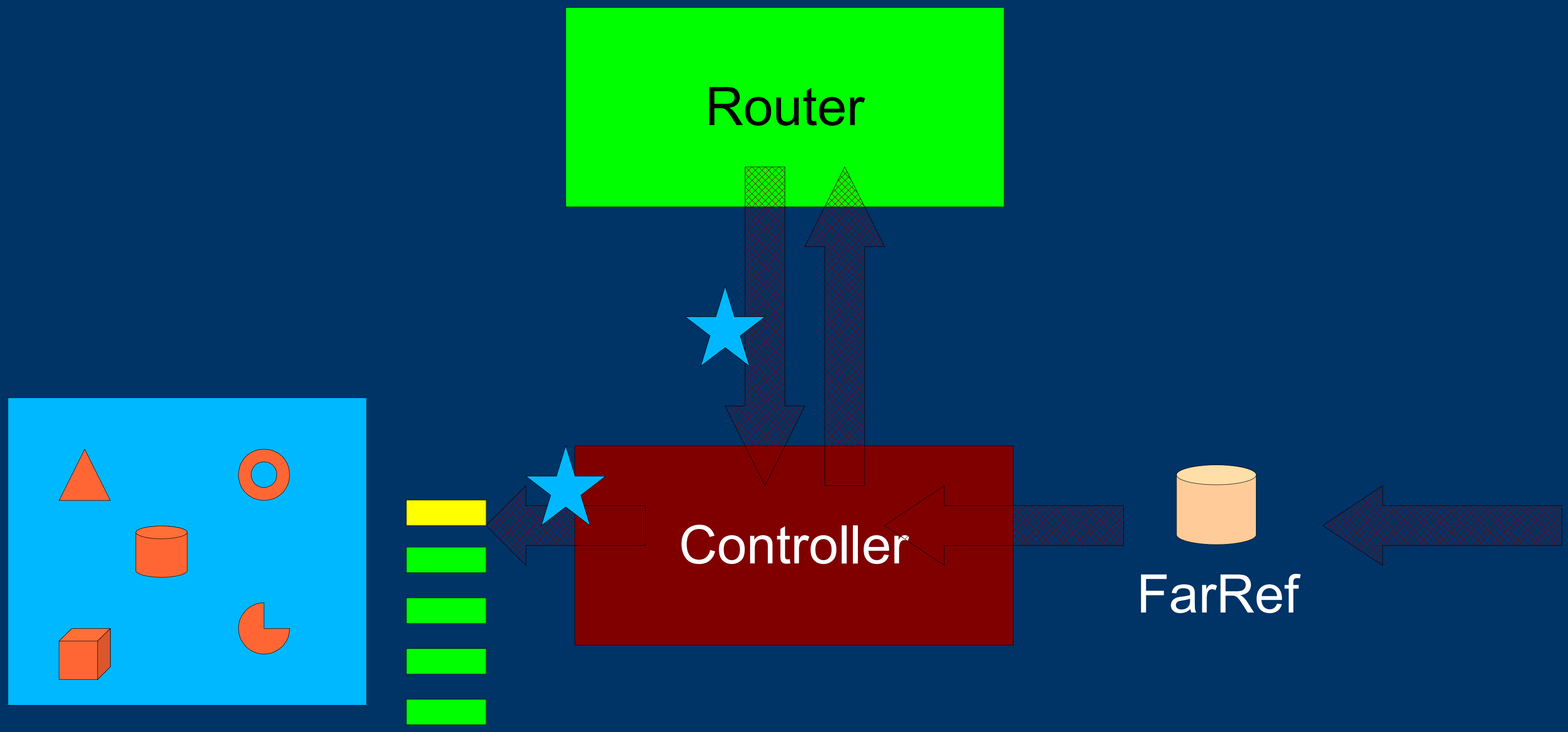
Island executes all messages up to the new external message



Machine A

farRef future aMessage:args

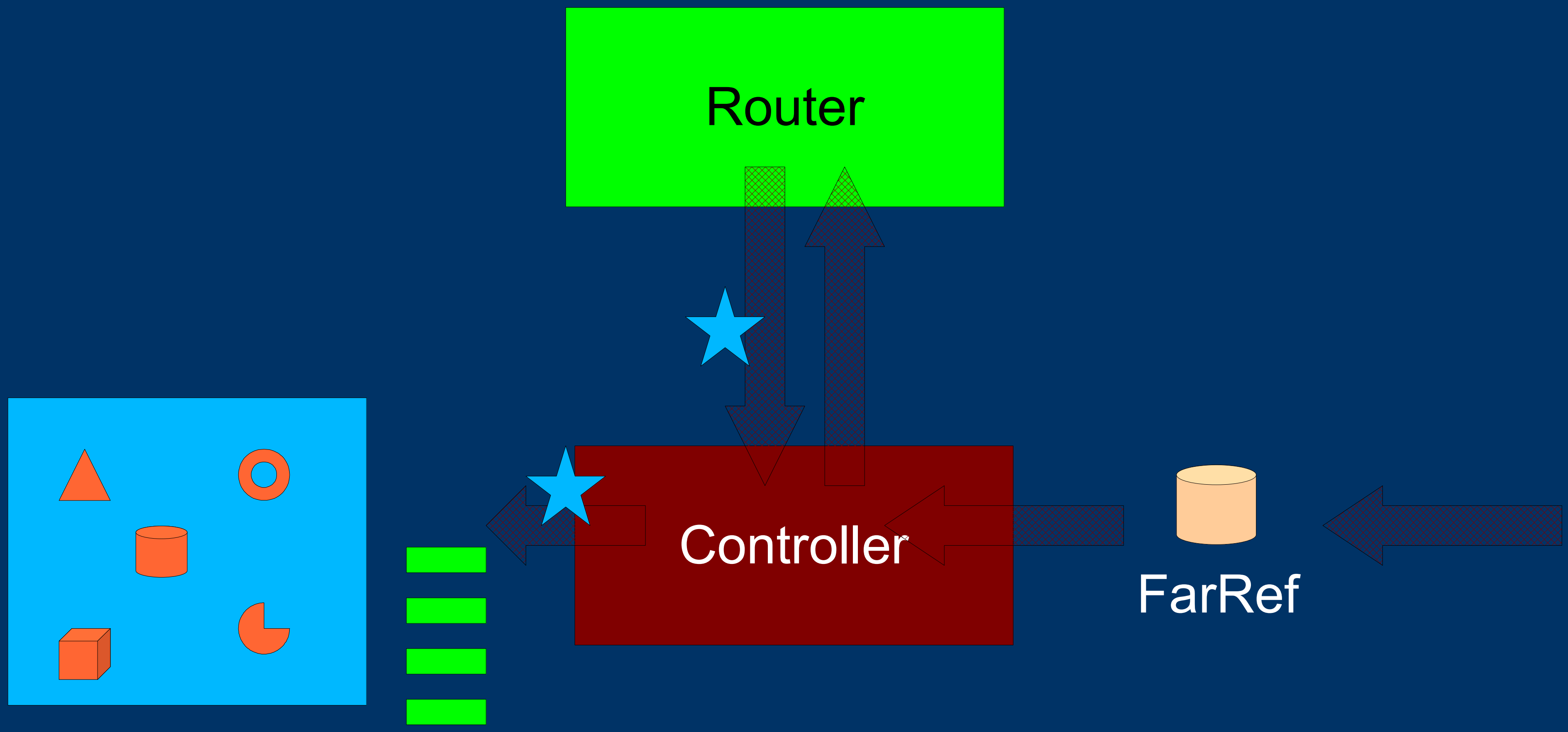
Island executes all messages up to the new external message



Machine A

farRef future aMessage:args

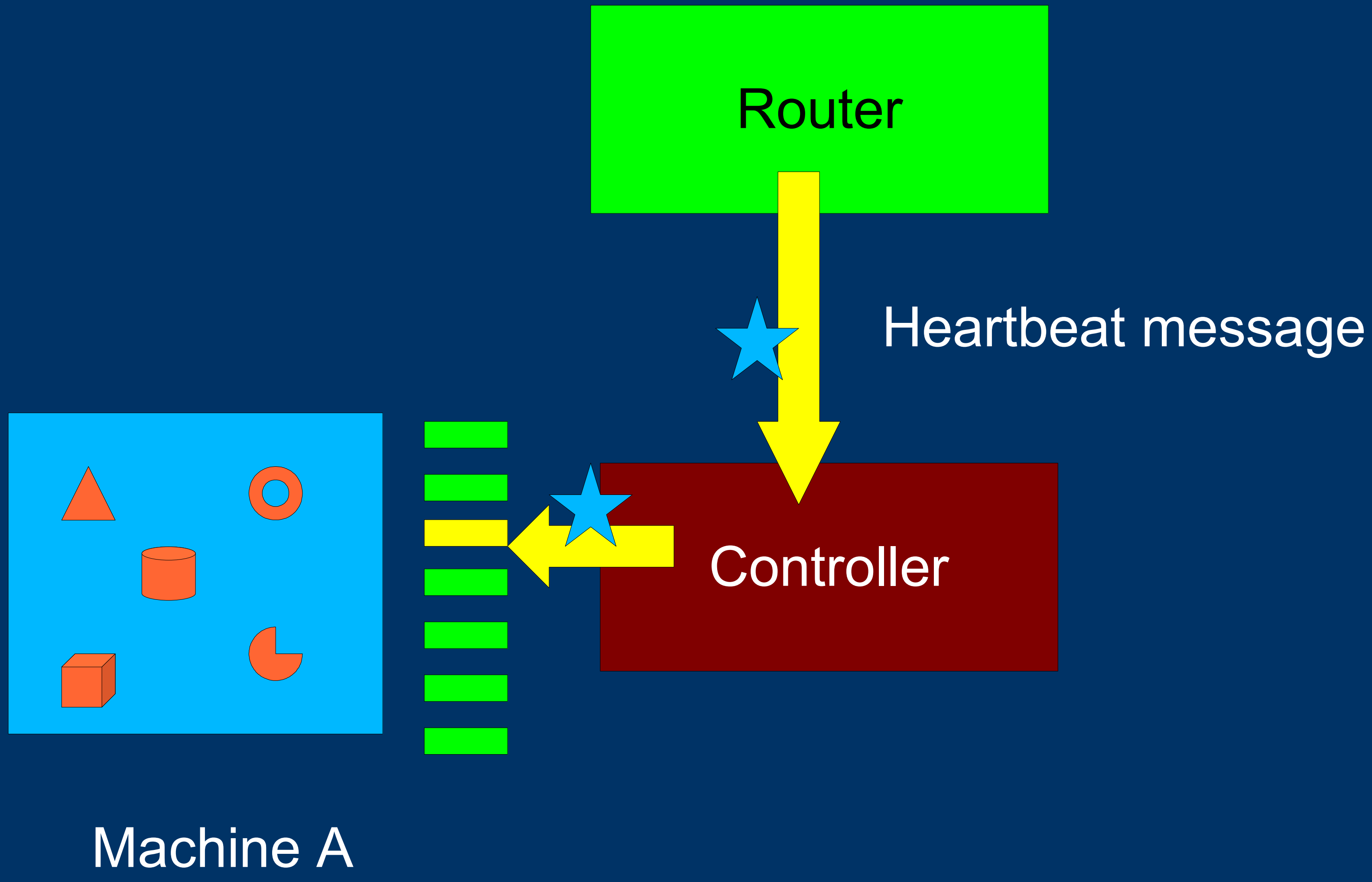
Island executes all messages up to the new external message



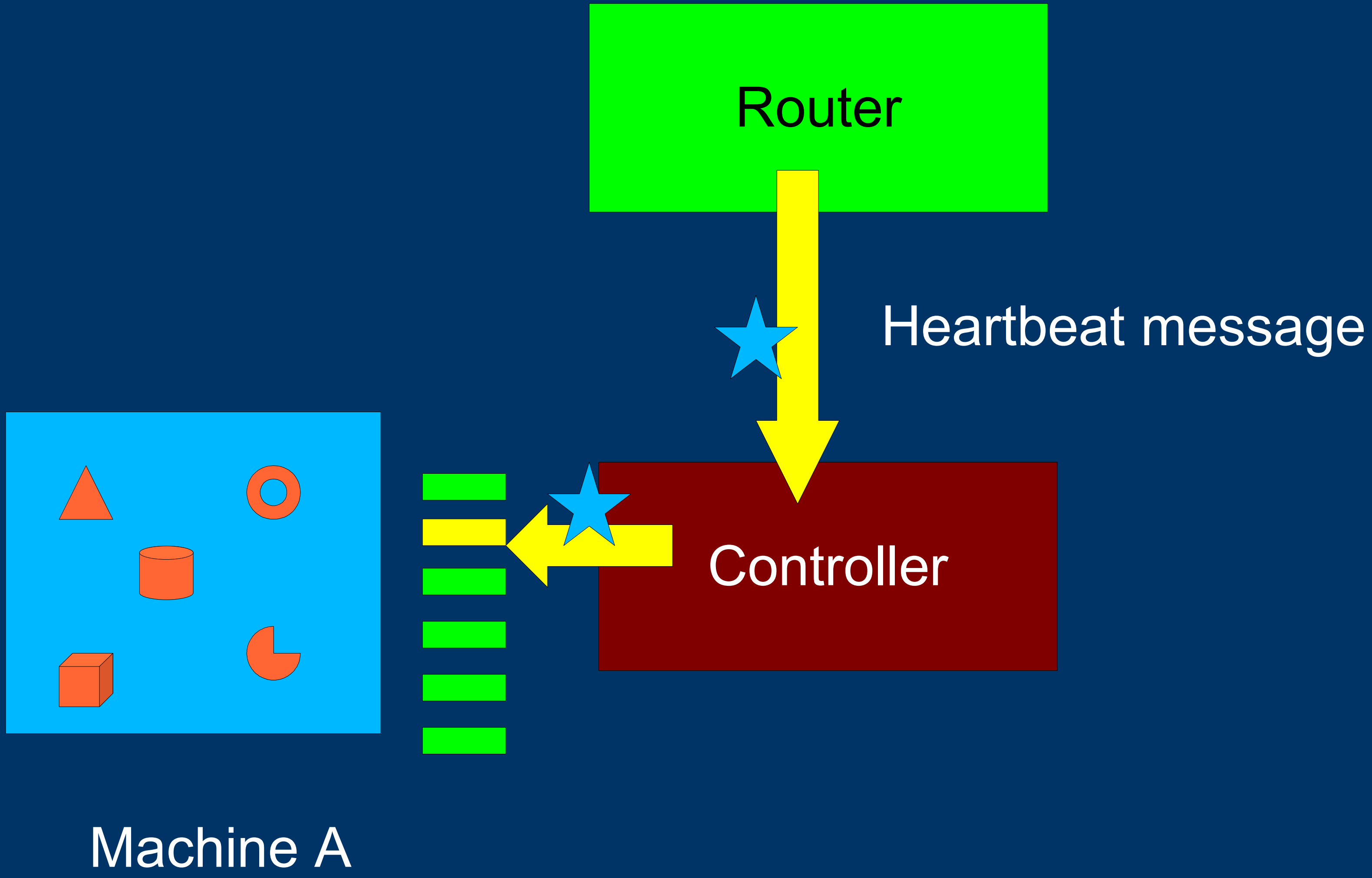
Machine A

farRef future aMessage:args

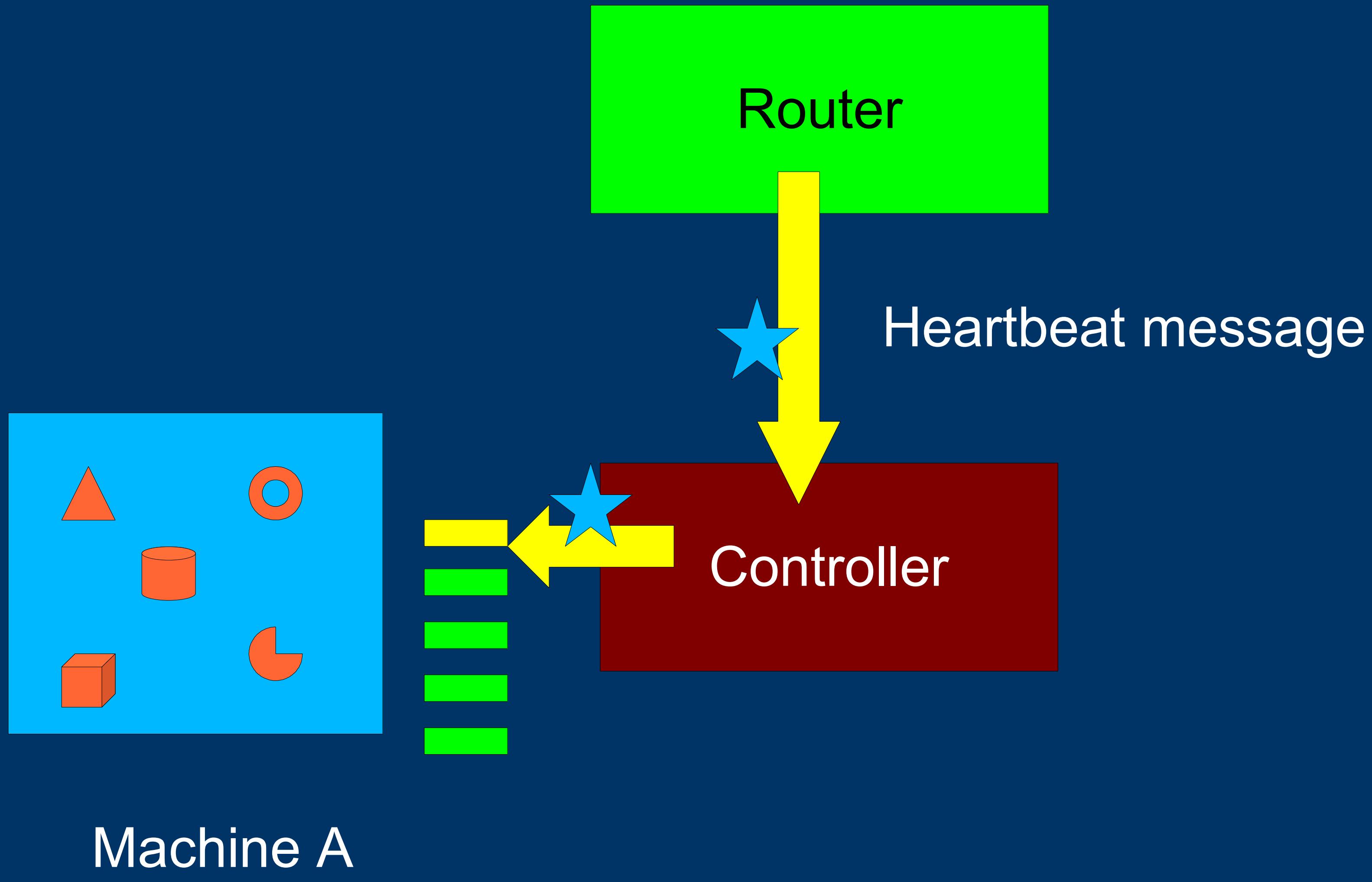
If there is no external message to move things forward, the router will manufacture one.



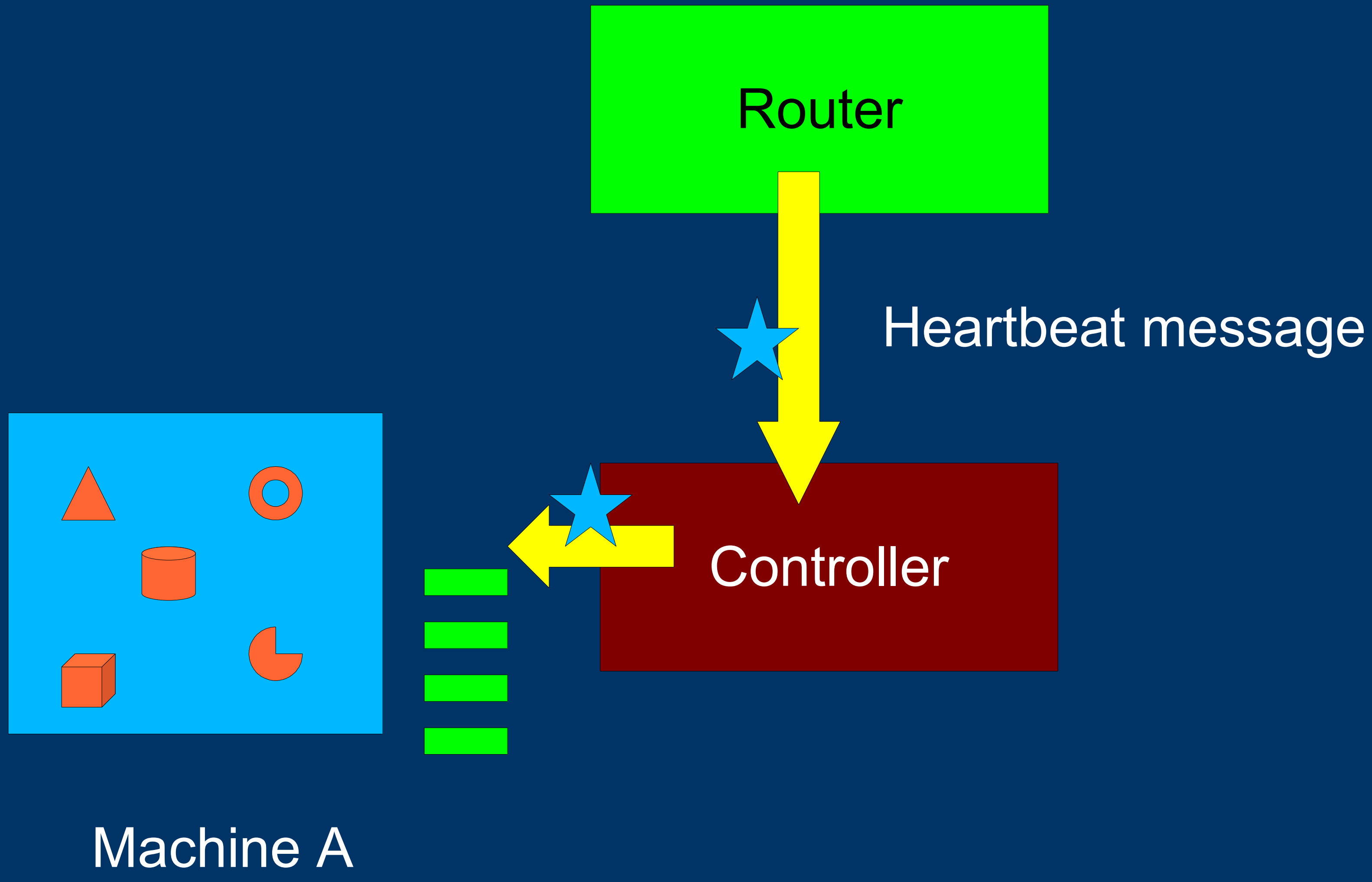
Messages are then executed by Island up to and including the heartbeat message from Router



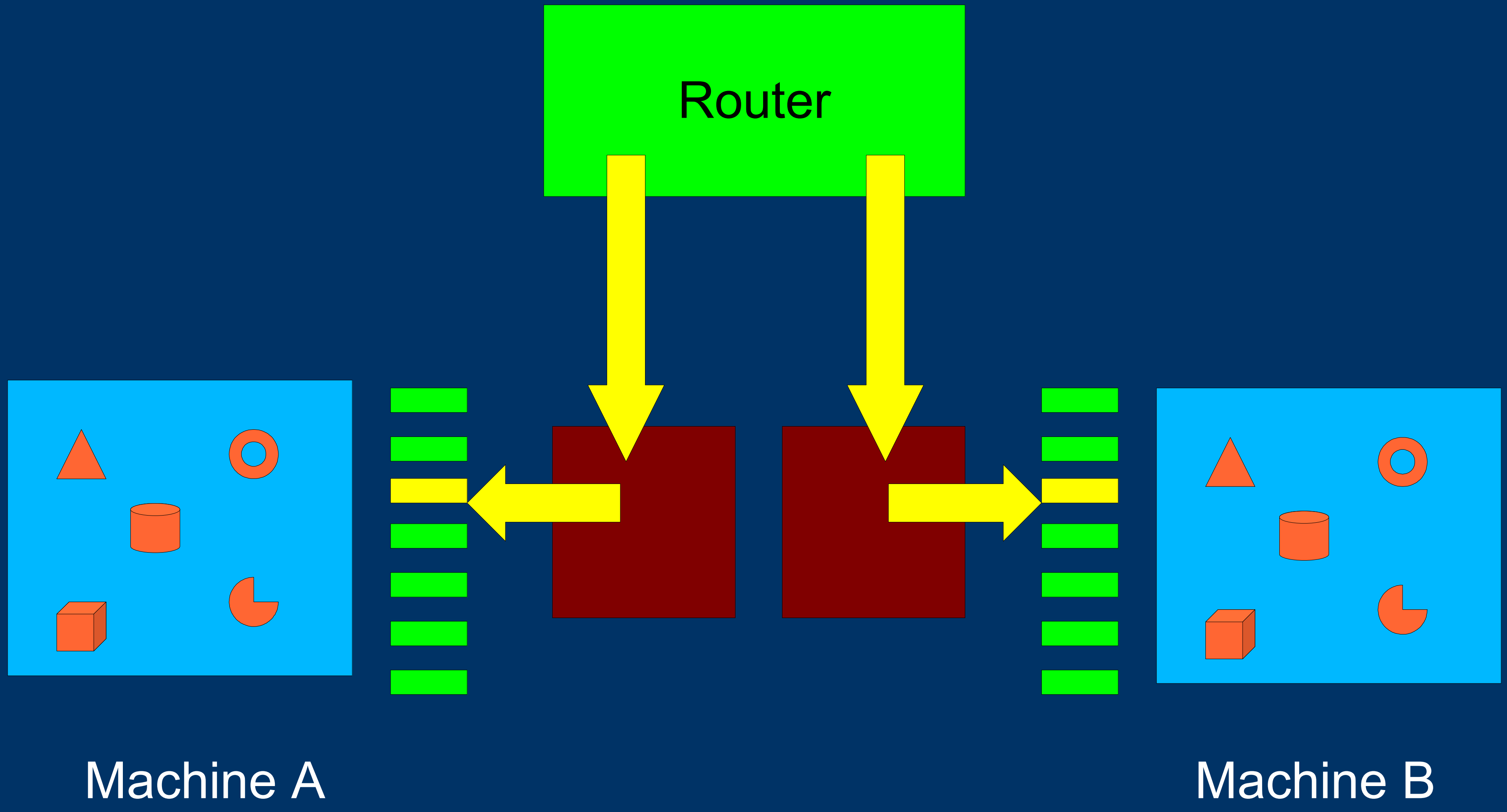
Messages are then executed by Island up to and including the heartbeat message from Router



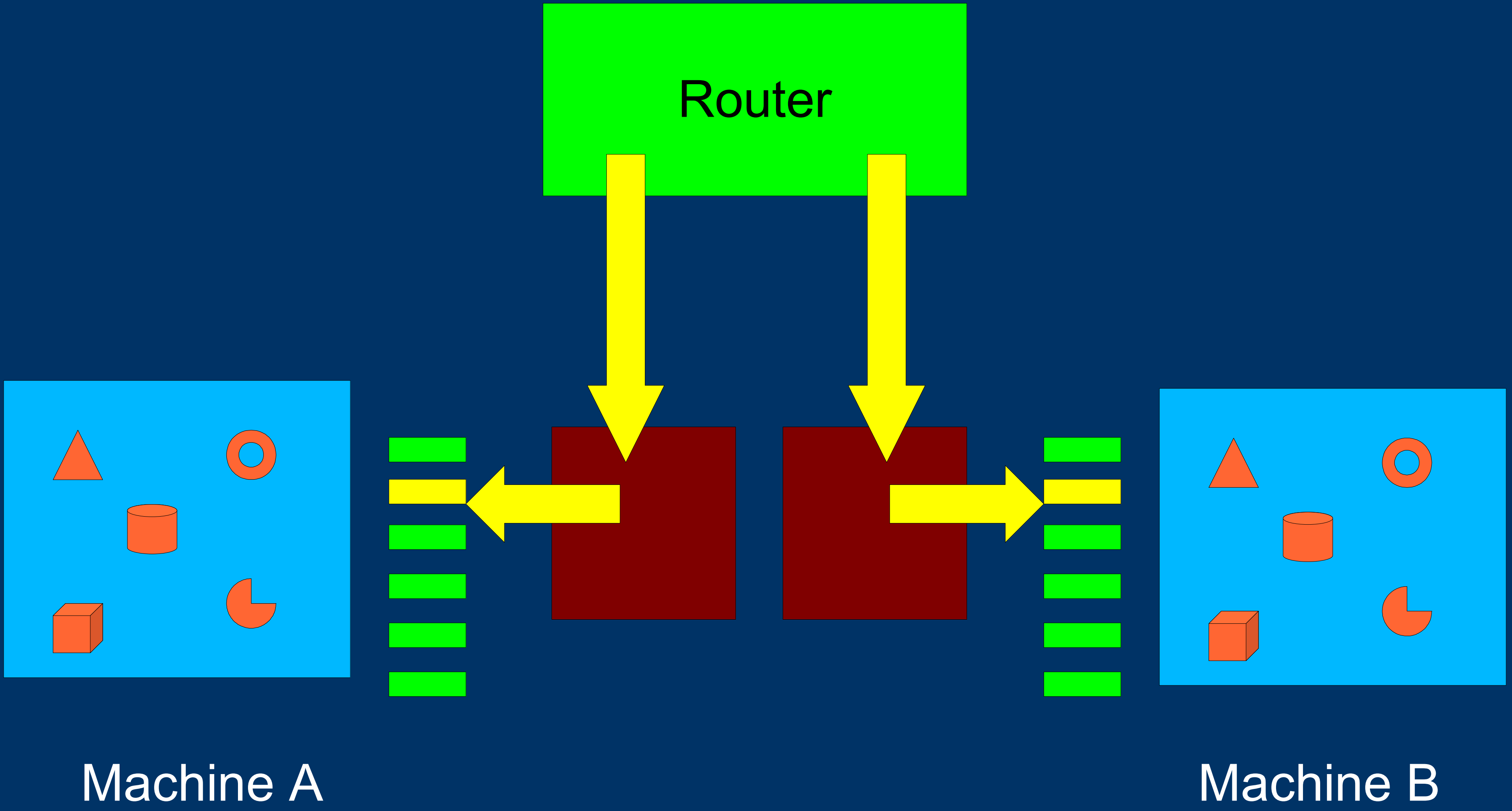
Messages are then executed by Island up to and including the heartbeat message from Router



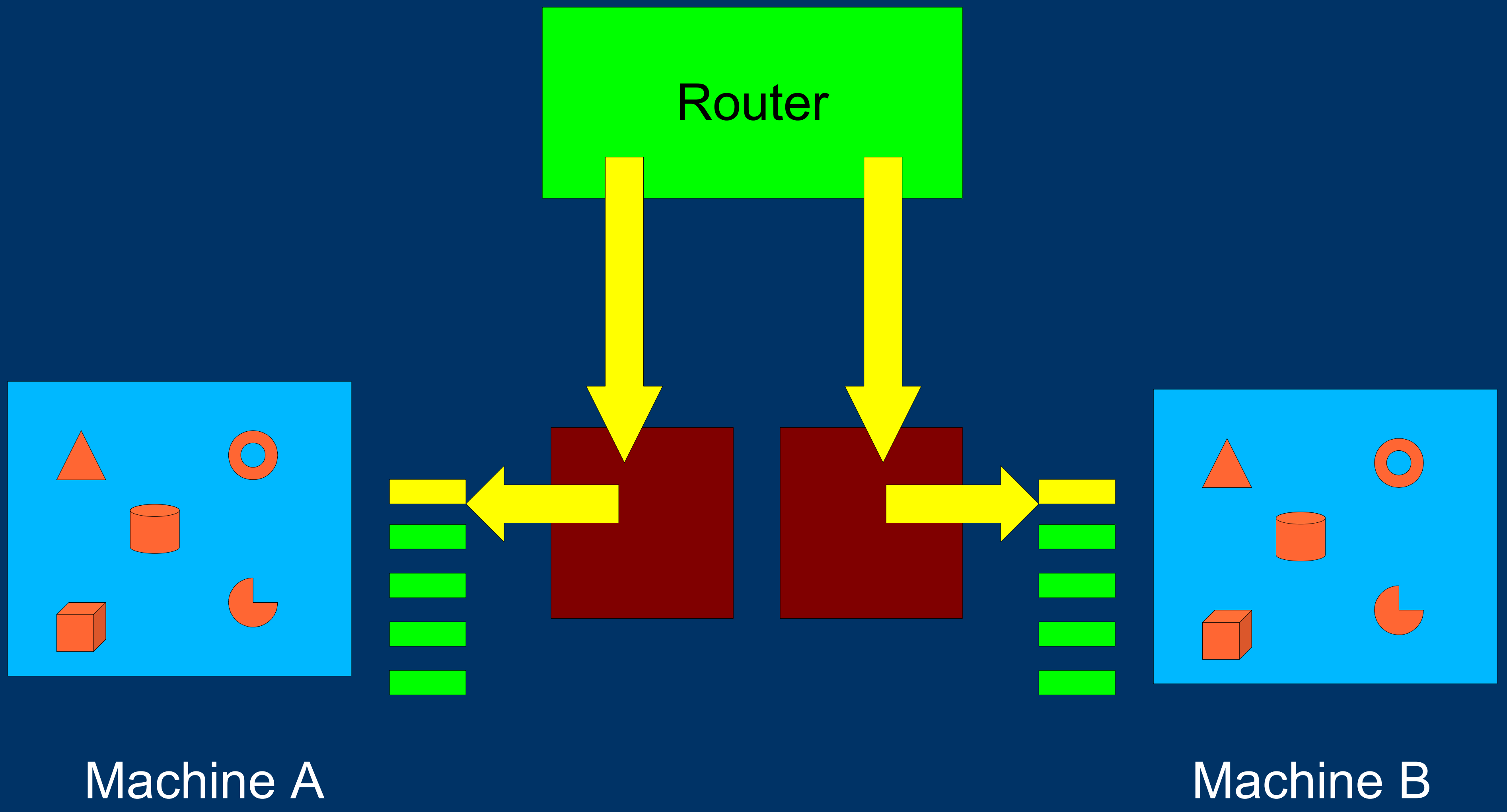
This works for any number of replicated islands.



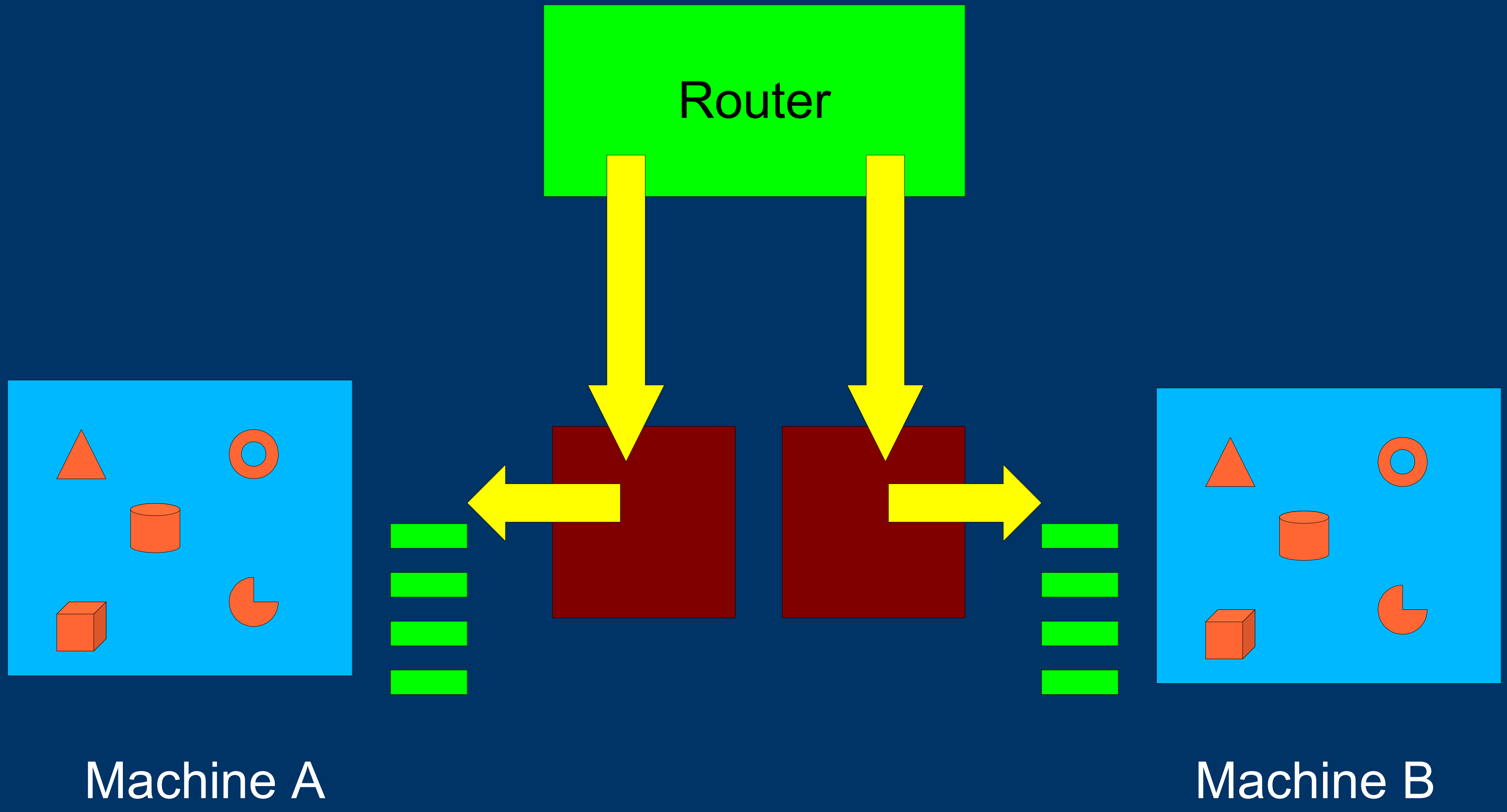
This works for any number of replicated islands.



This works for any number of replicated islands.



This works for any number of replicated islands.



Router/Controller/Island

- Does not matter where the message comes from
- Islands can not move past whatever time the Router specifies it is
- Router sends heartbeat messages to move time forward when no external messages are available to drive time forward
- Guarantees Islands execute identical messages in identical order

Router enumerates messages

- Messages from router are enumerated.
- If controller receives m1, m2, m4, controller knows that it missed m3 and request that it be re-sent.

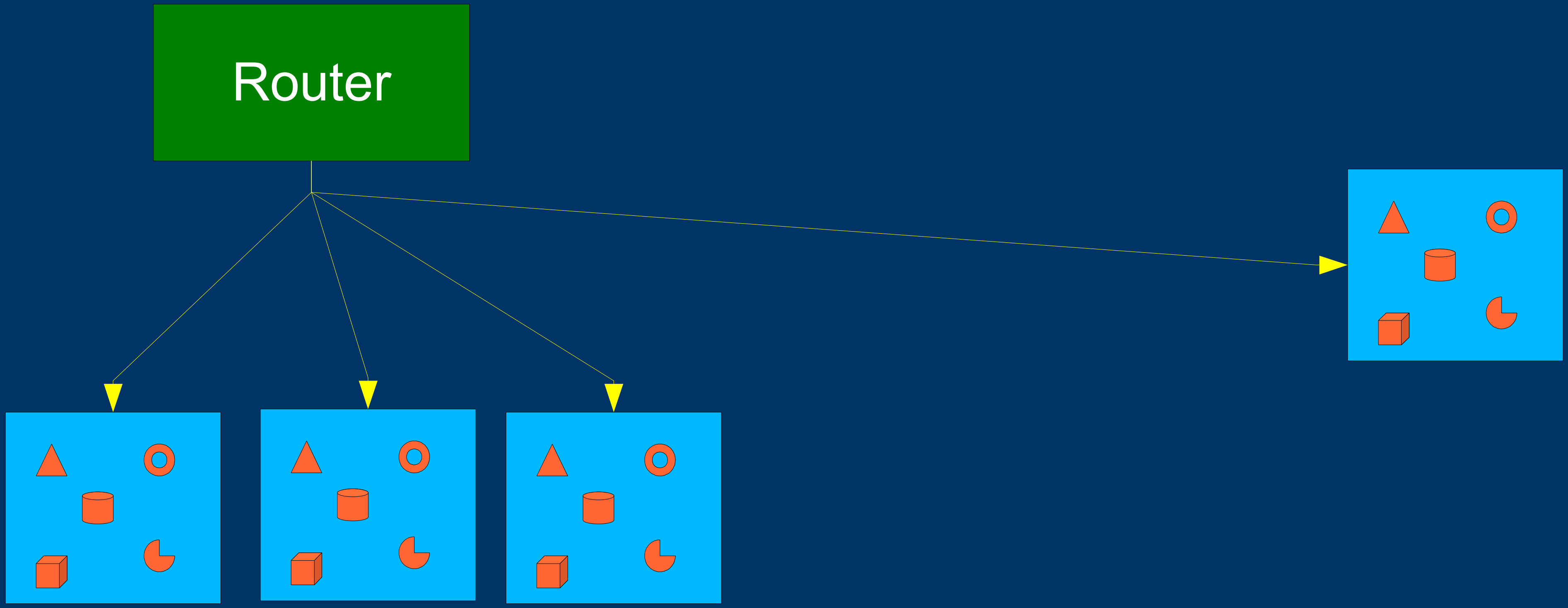
Islands view of time

- Islands only understand time in quantized terms – there is only now (when message is executed)
- – and now + x (when future message will be executed)
- Router controls execution time for all islands.
- Router needs to send heartbeat messages to ensure smooth animations.
- Heartbeat messages can be ignored by controller, with result of jerky updates.

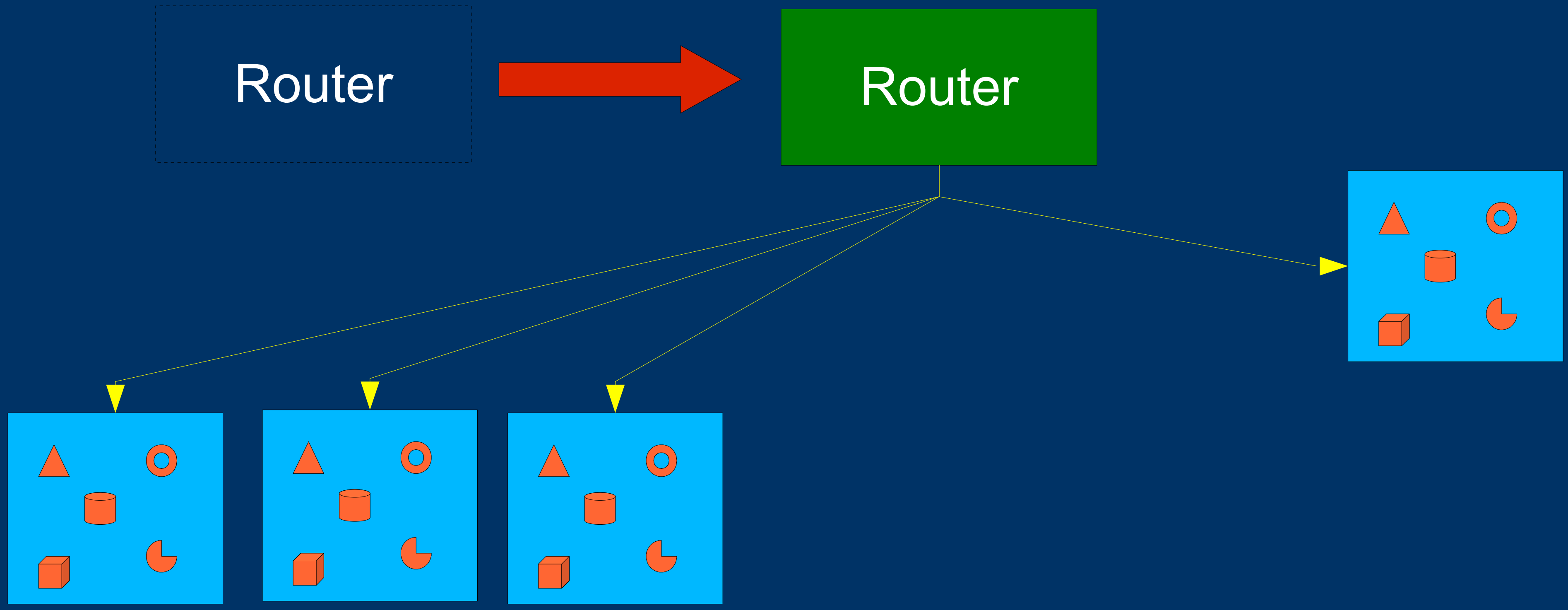
Nice side effects

- Latency does not create timing problems, just feedback problems (system acts sluggish if you have higher latency).
- Users are not punished for having a high-latency participant sharing an island (though the high latency participant has a poor experience).
- Routers can be independent of Island/controller pairs, hence can be positioned on minimal latency or centralized balanced latency servers.
- Routers can even be moved around if necessary to improve latency for specific users or groups.

Latency does not effect accuracy – only usability.

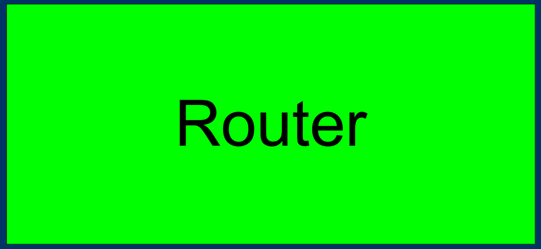


Router can be moved to more latency centric location.



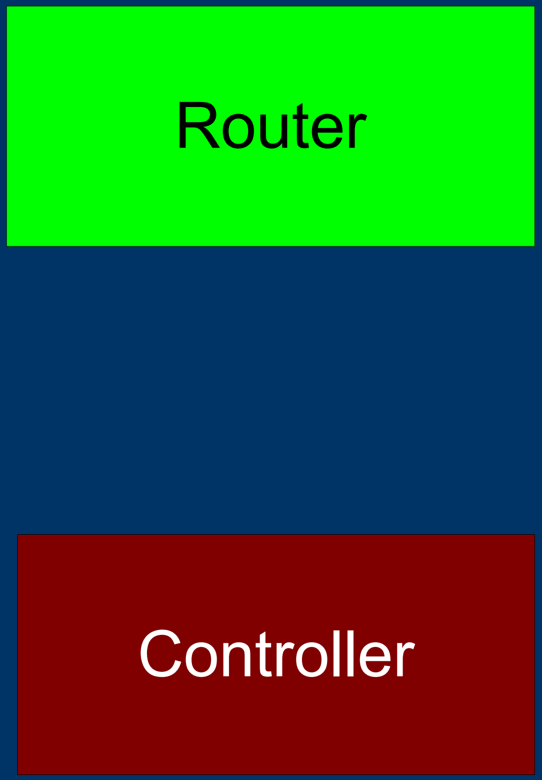
Starting and Joining

First there was the router...



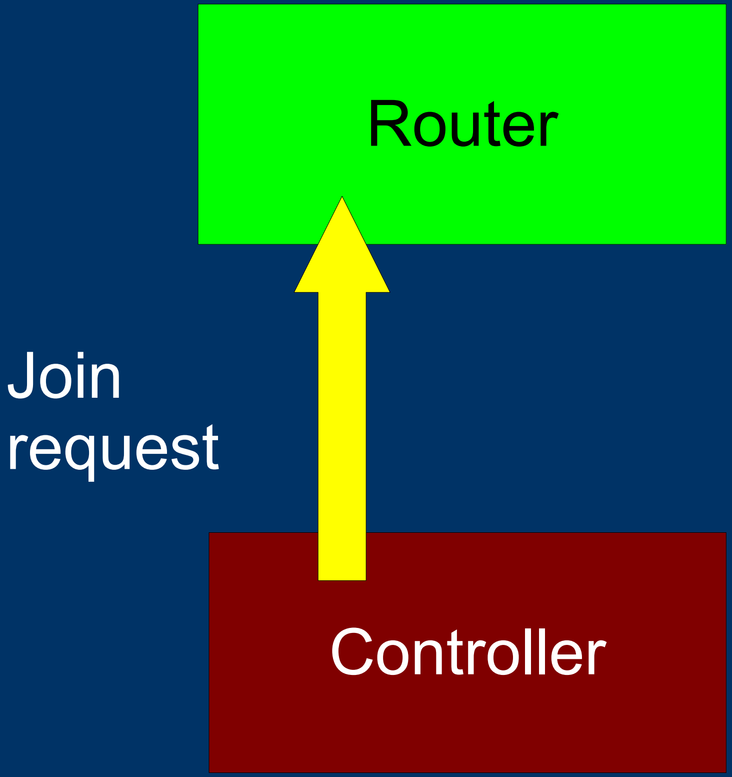
The new router can be on any machine, not just the users.

Next create a Controller



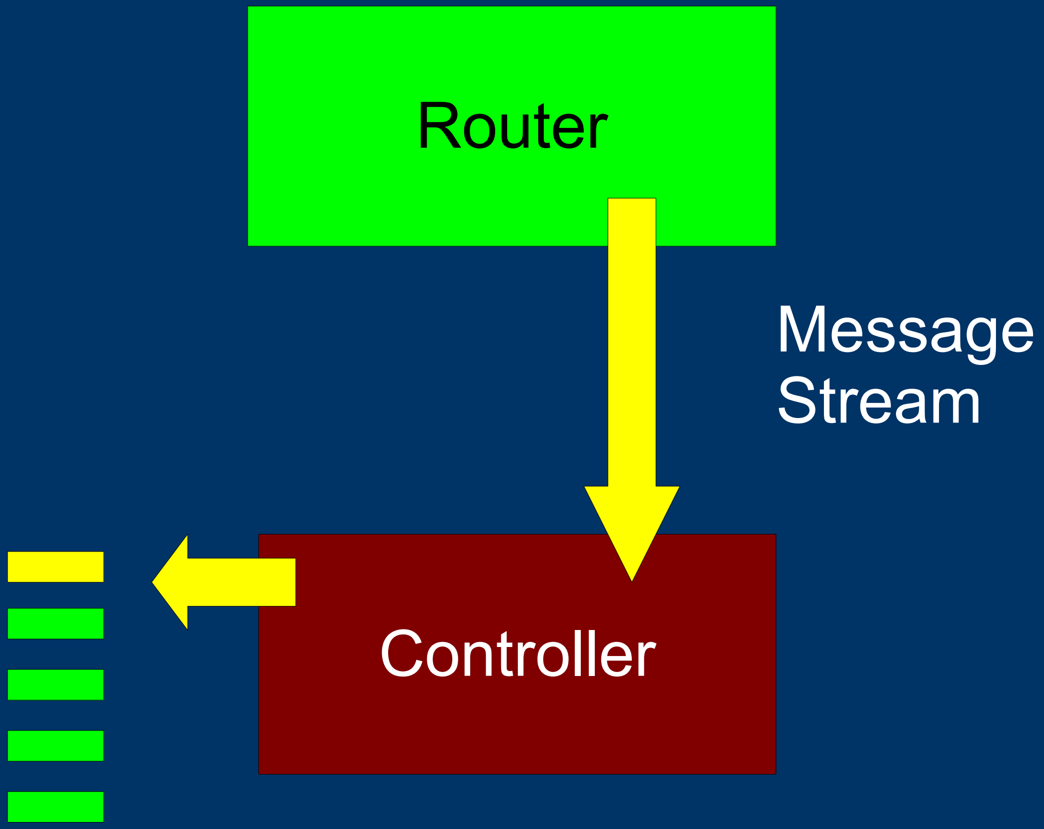
The new controller will be on the users machine. It is given the Router address and port number. Since it will be used to construct the initial island, we call it the master controller.

Request to join



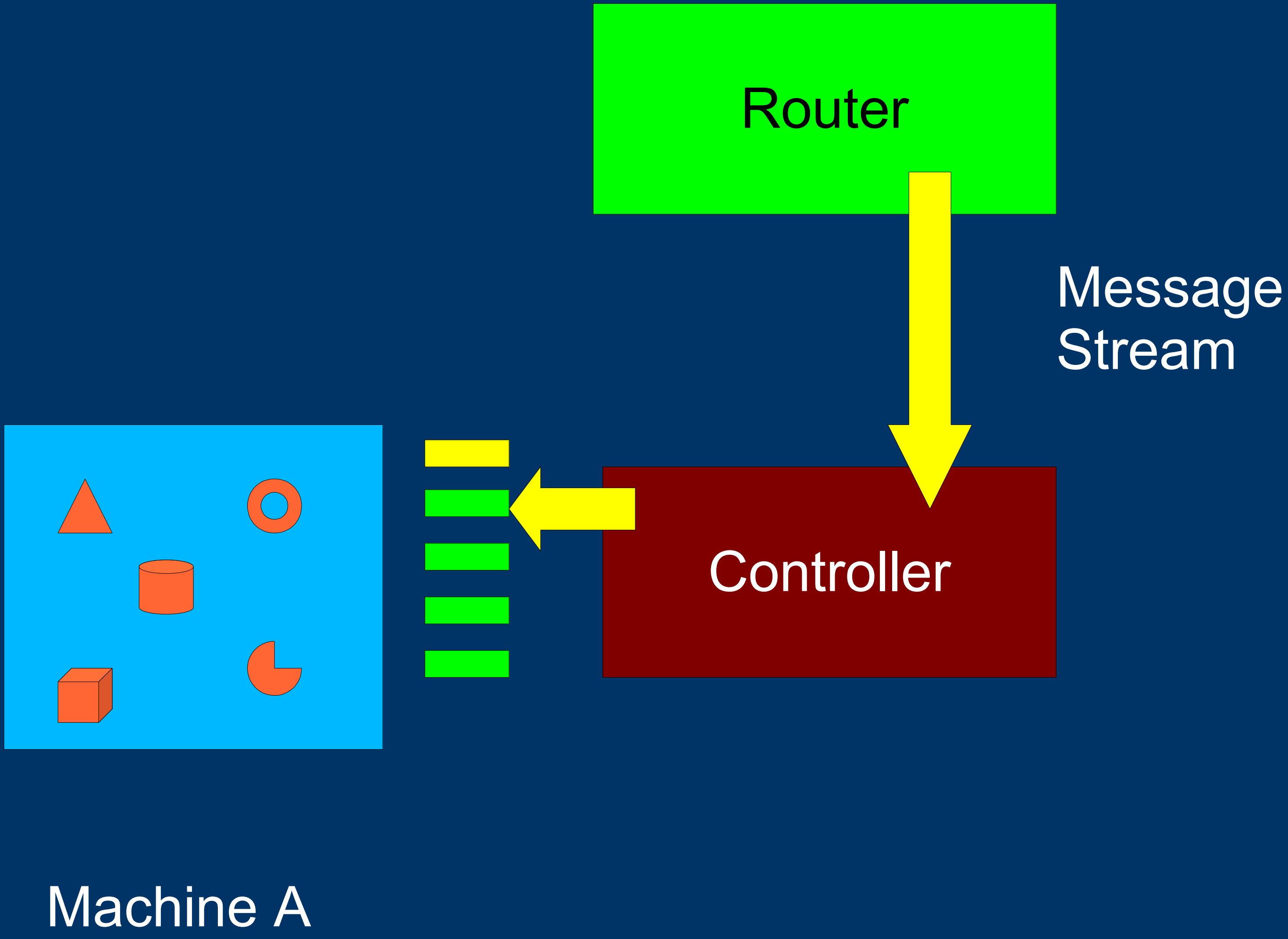
The controller sends a message to the router asking for messages. The router (if it is authorized) begins publishing its message stream to the controller.

Controller joins Router message stream

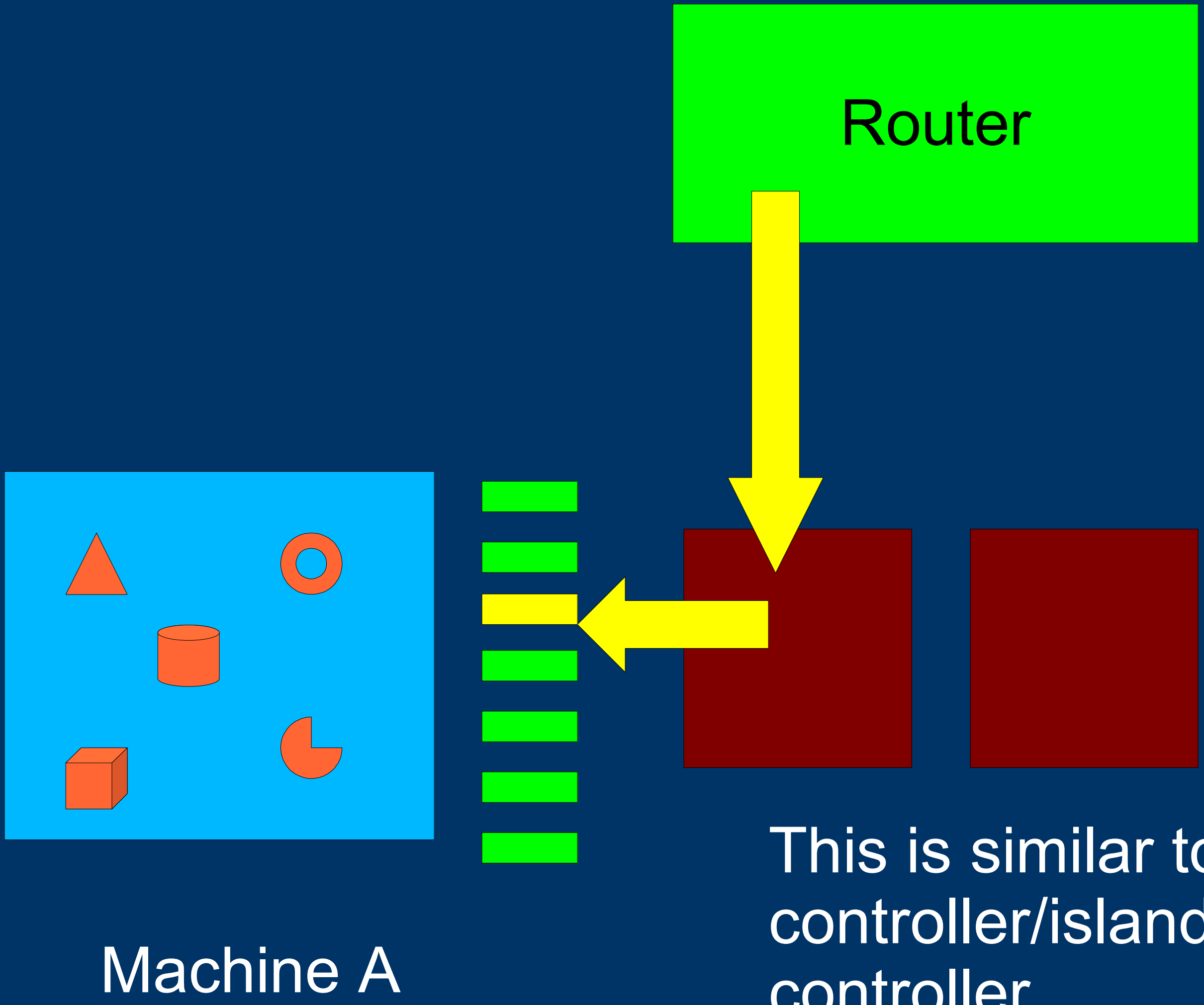


Once the join is accepted, the router sends all replicated messages and heartbeats to the controller. The controller saves these into a message queue.

Controller constructs new Island

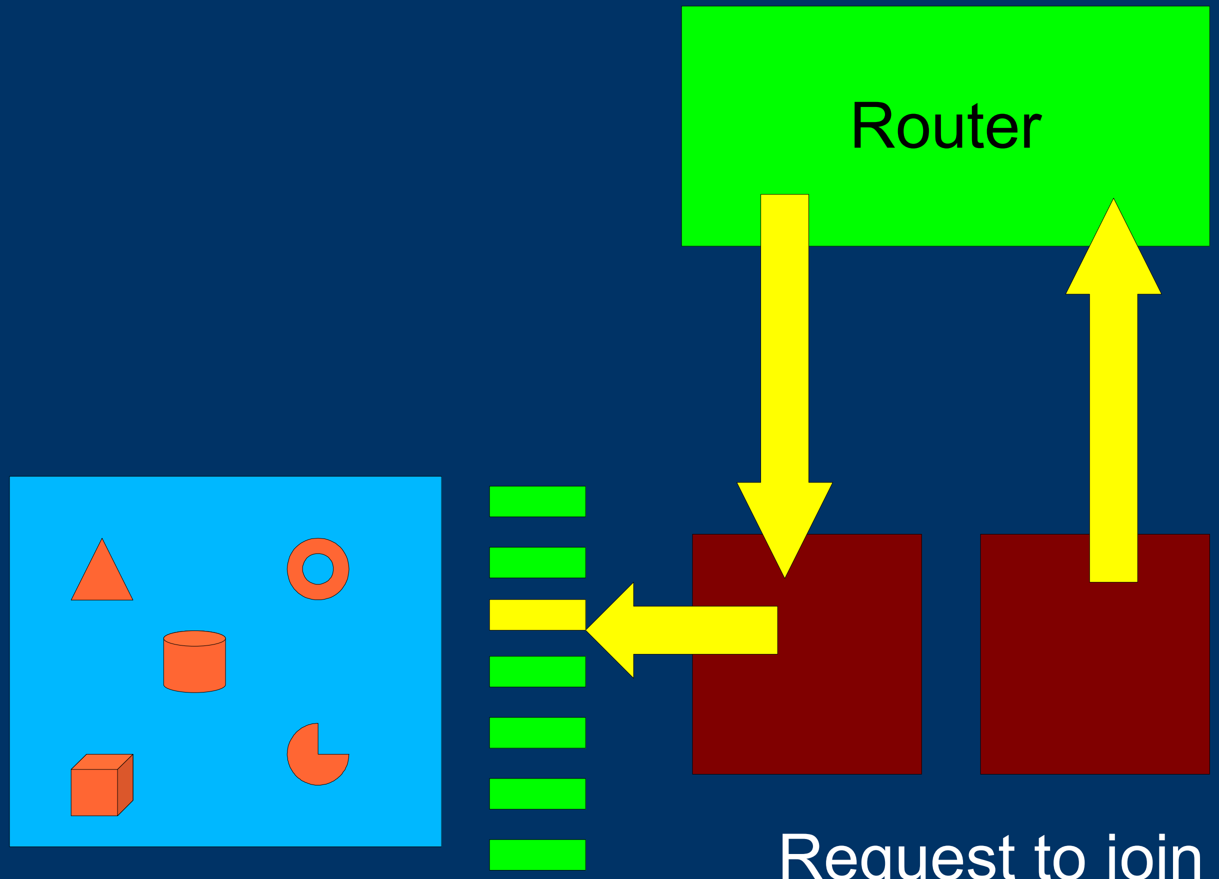


Adding a new user – construct controller



This is similar to constructing the initial controller/island pair. First, create the controller.

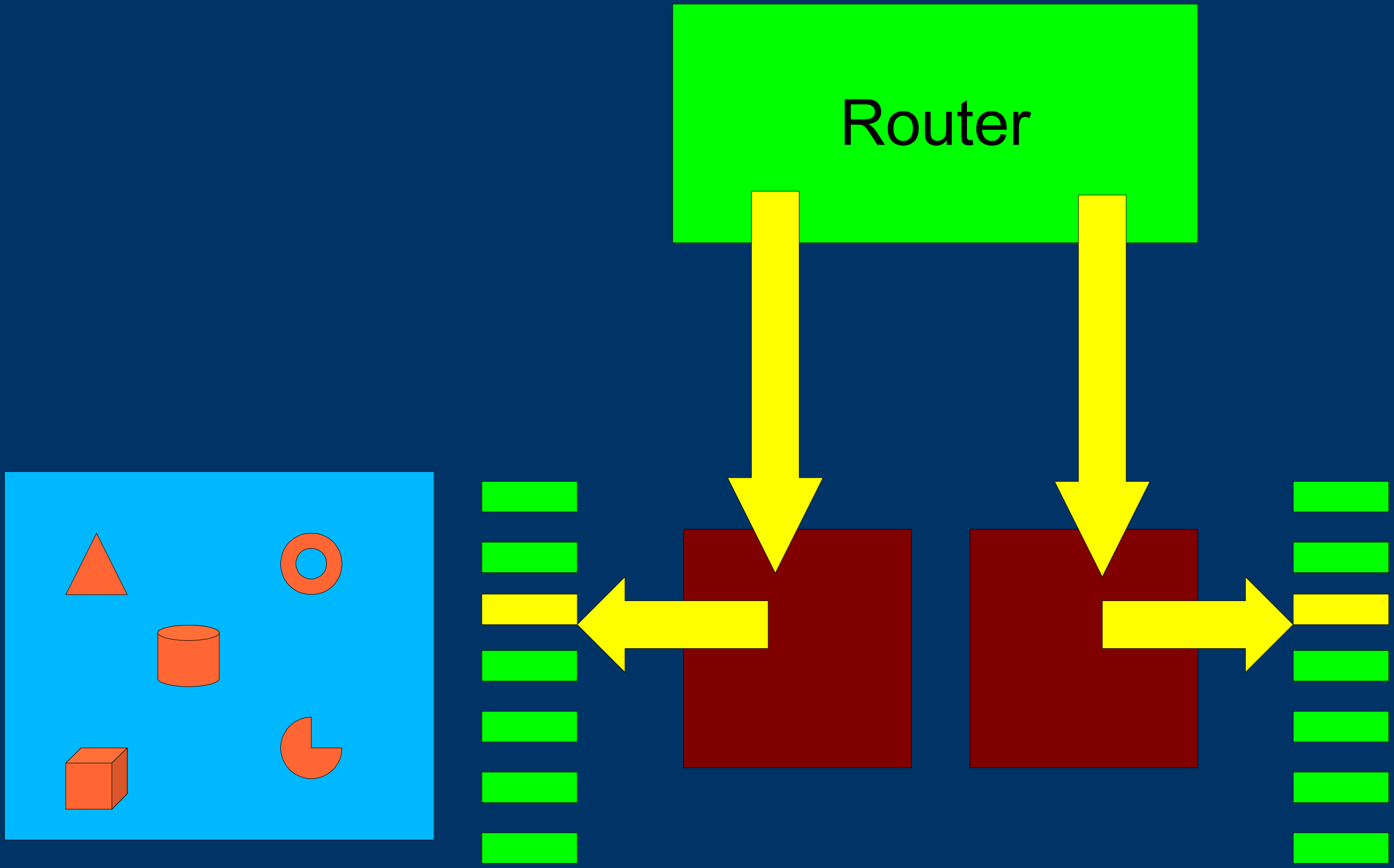
Request to join router



Request to join the router.

Machine A

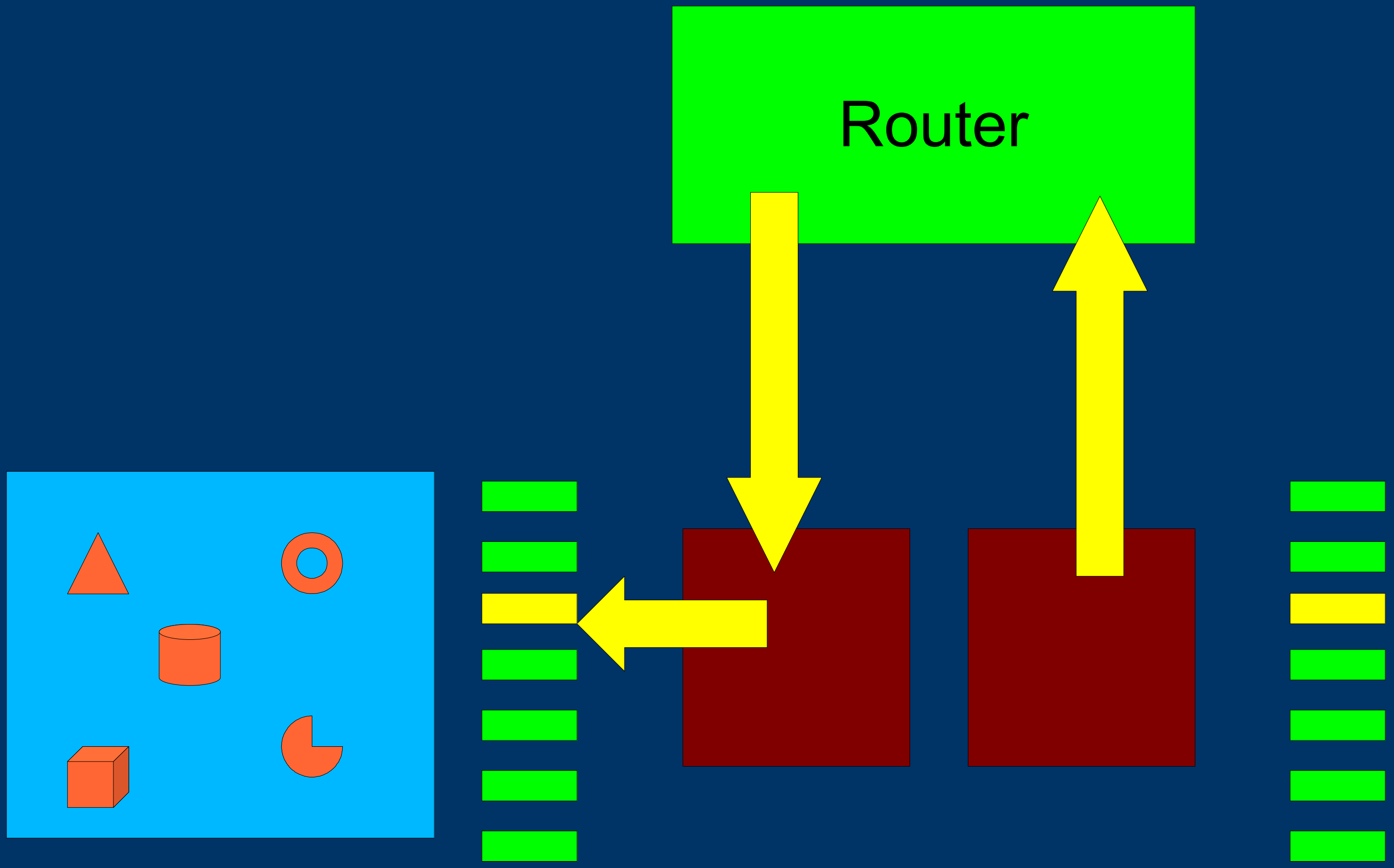
Start receiving messages



Machine A

Once granted, we add new messages into the message queue.

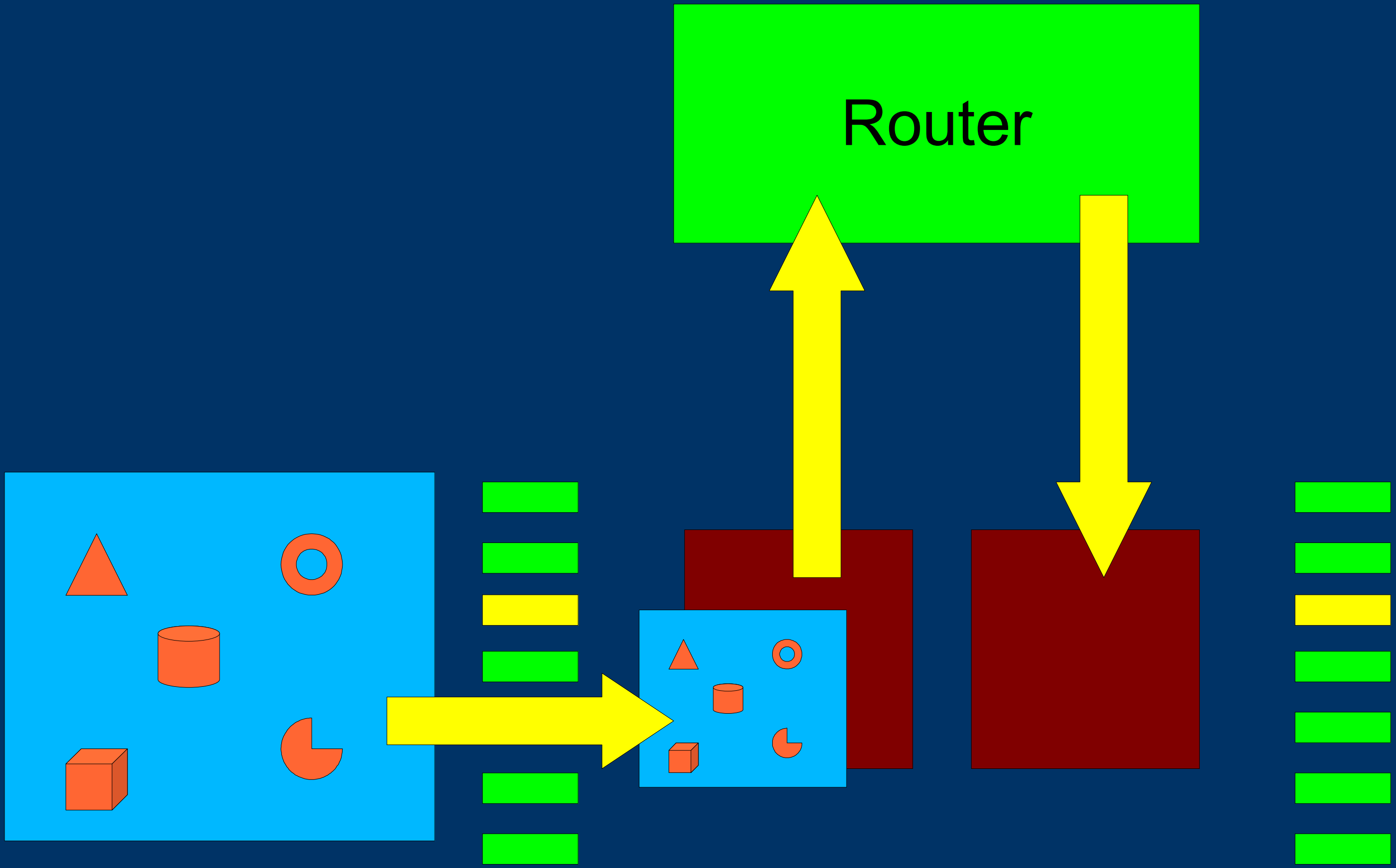
Request Island



Machine A

The controller can now be used to request a copy of the Island.

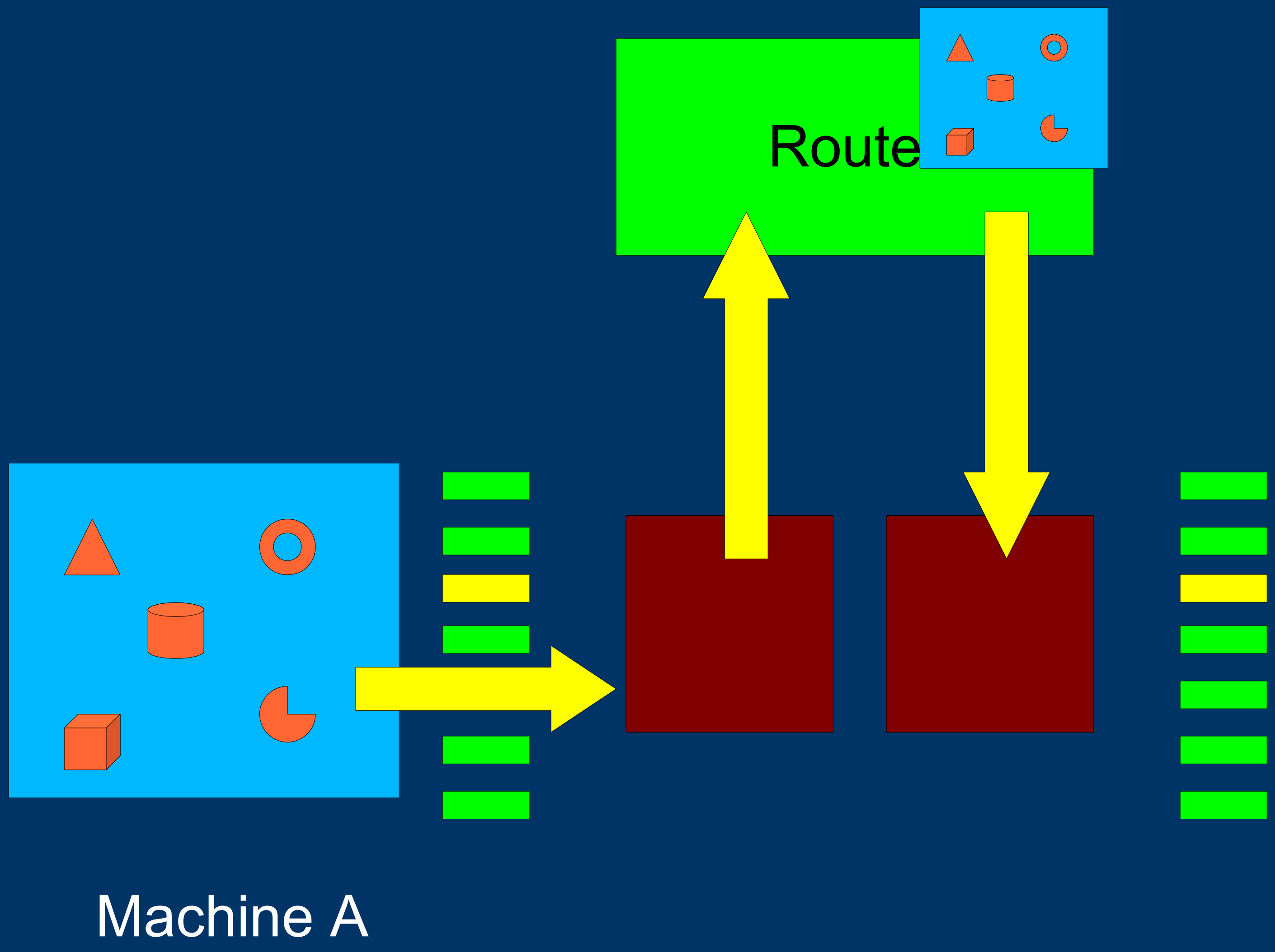
Island checkpointed and sent



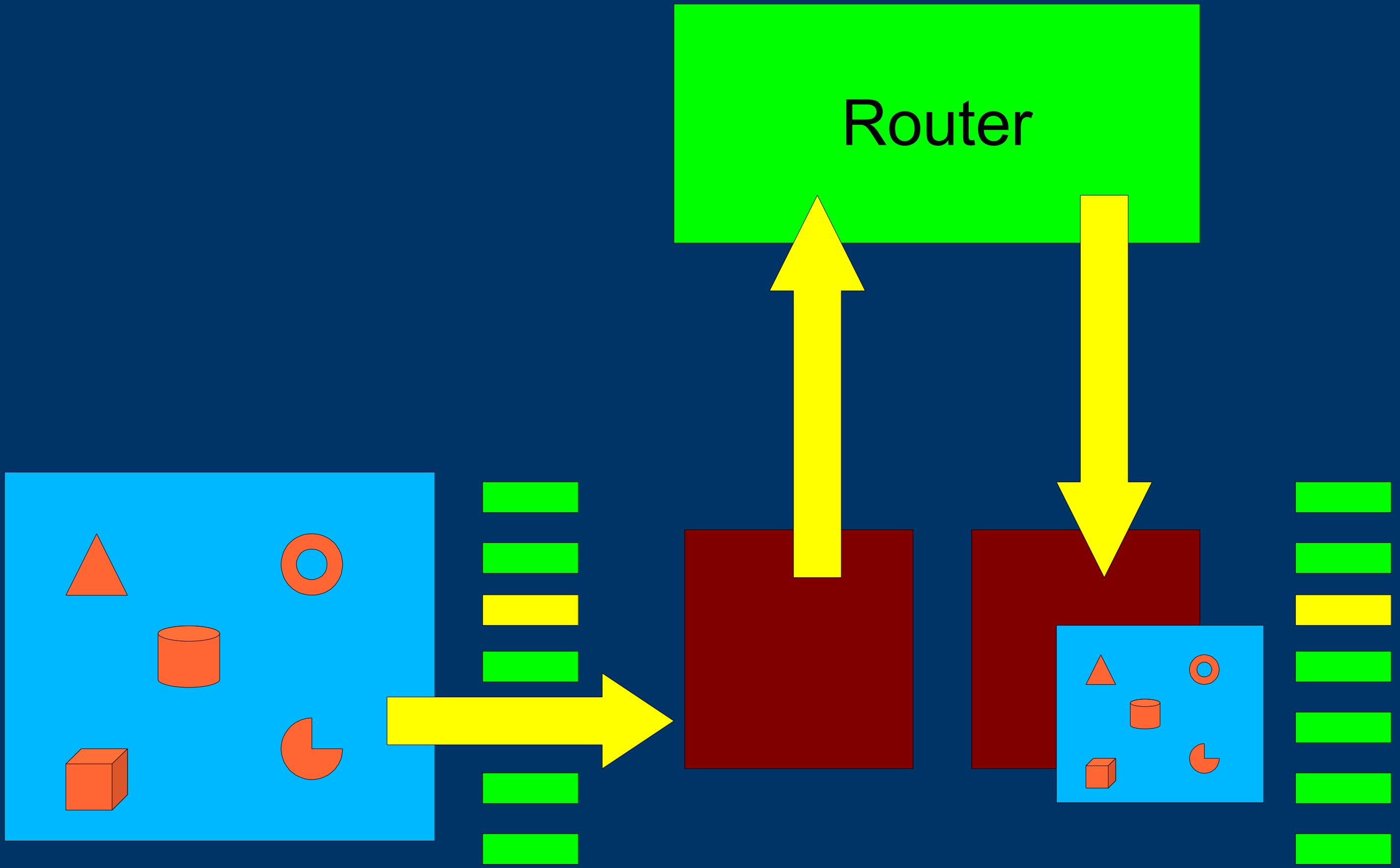
Machine A

The island is checkpoint streamed to the new controller via the router.

Island checkpointed and sent



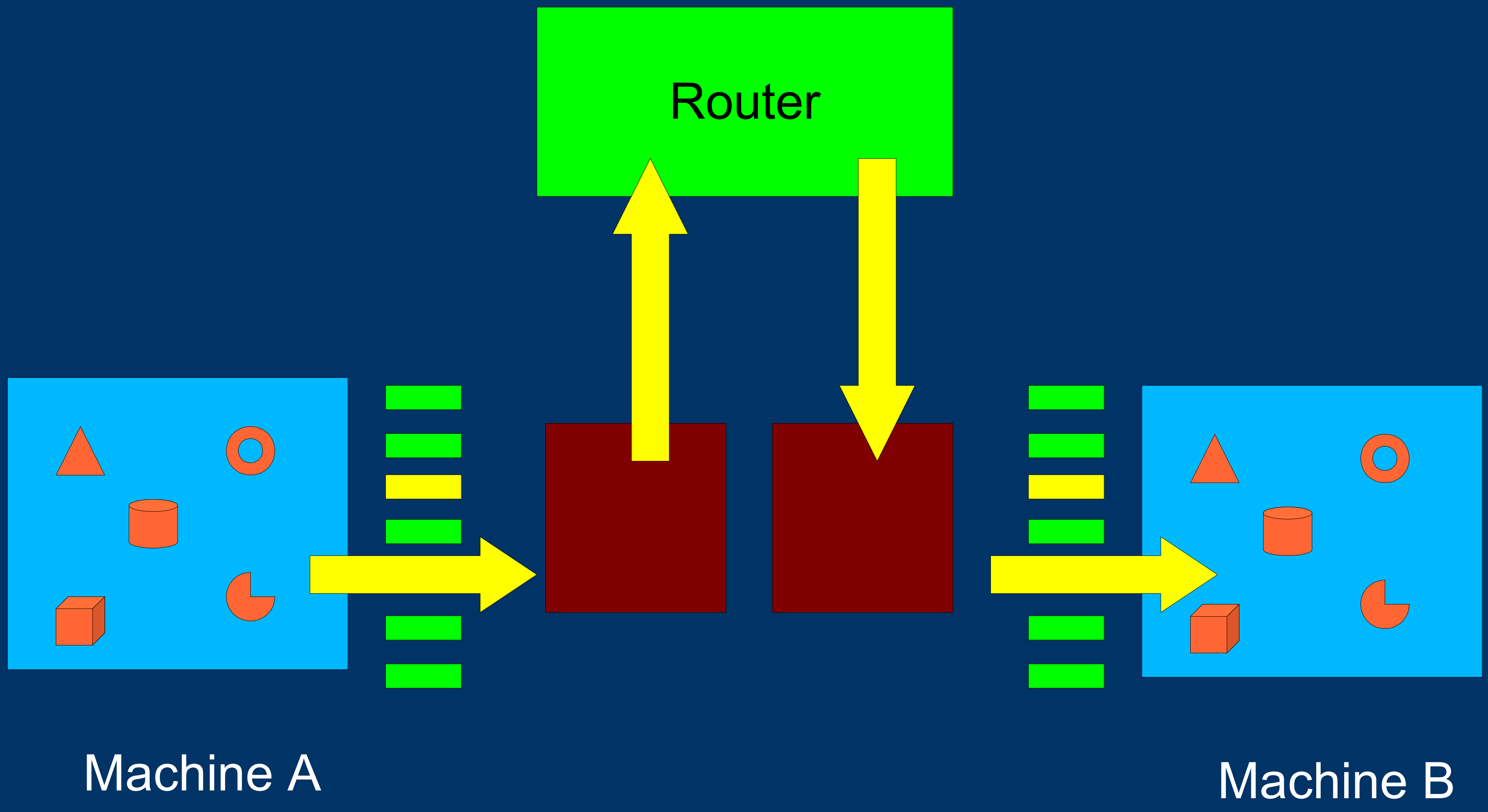
Island checkpointed and sent



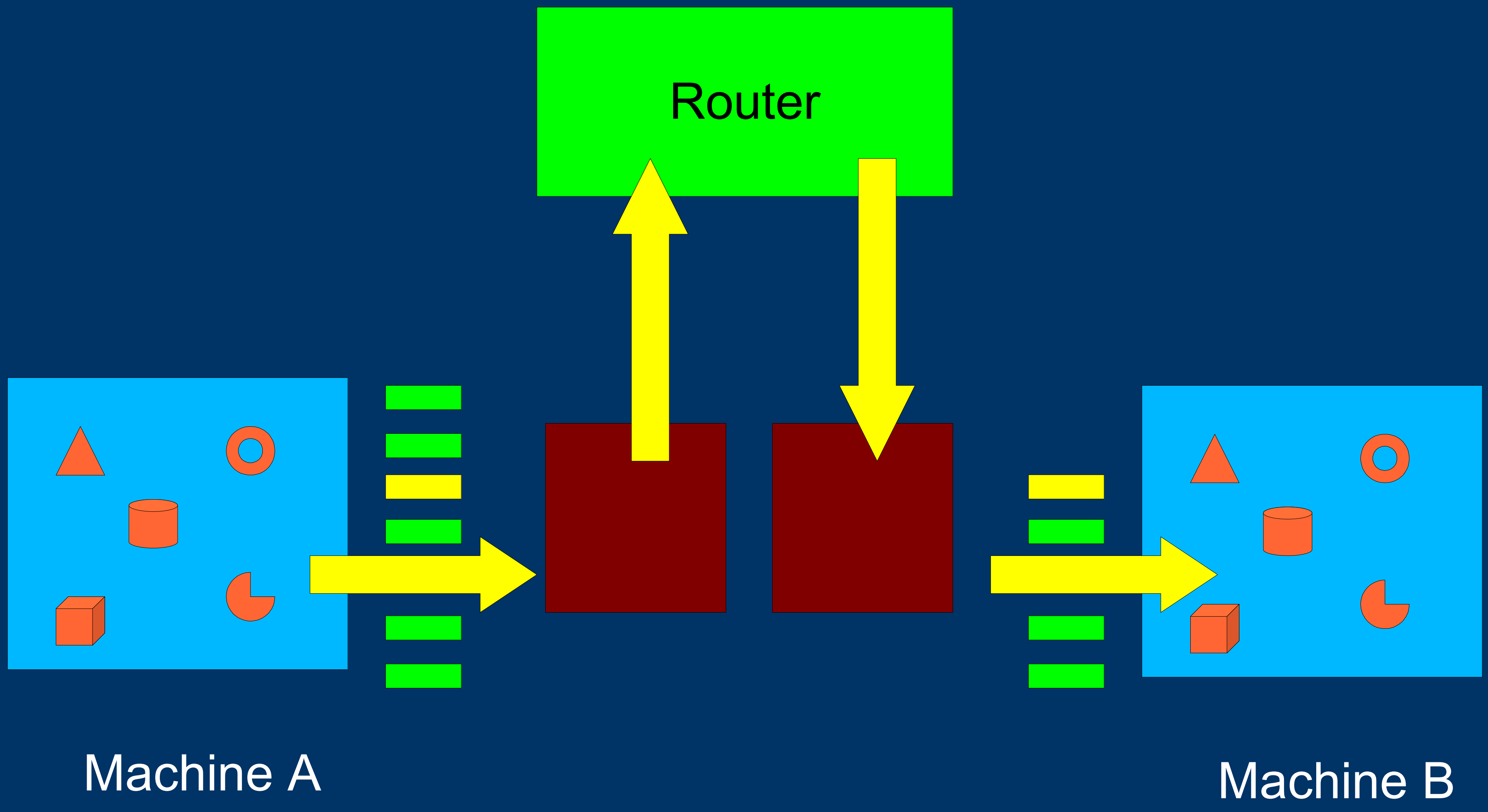
Machine A

The controller resurrects the island locally.

Island is resurrected and can now be displayed.



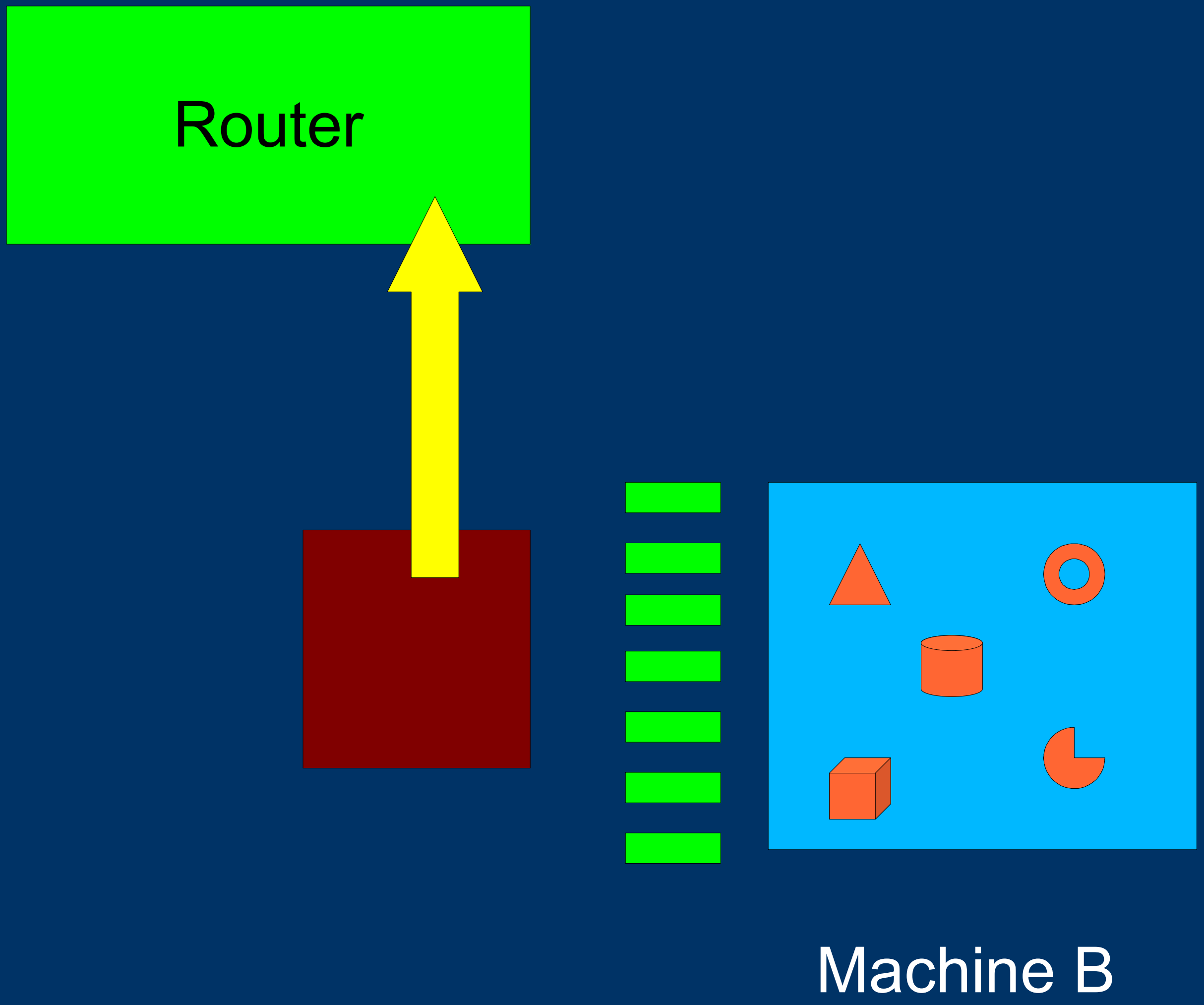
Message queue is culled to >= Island current time.



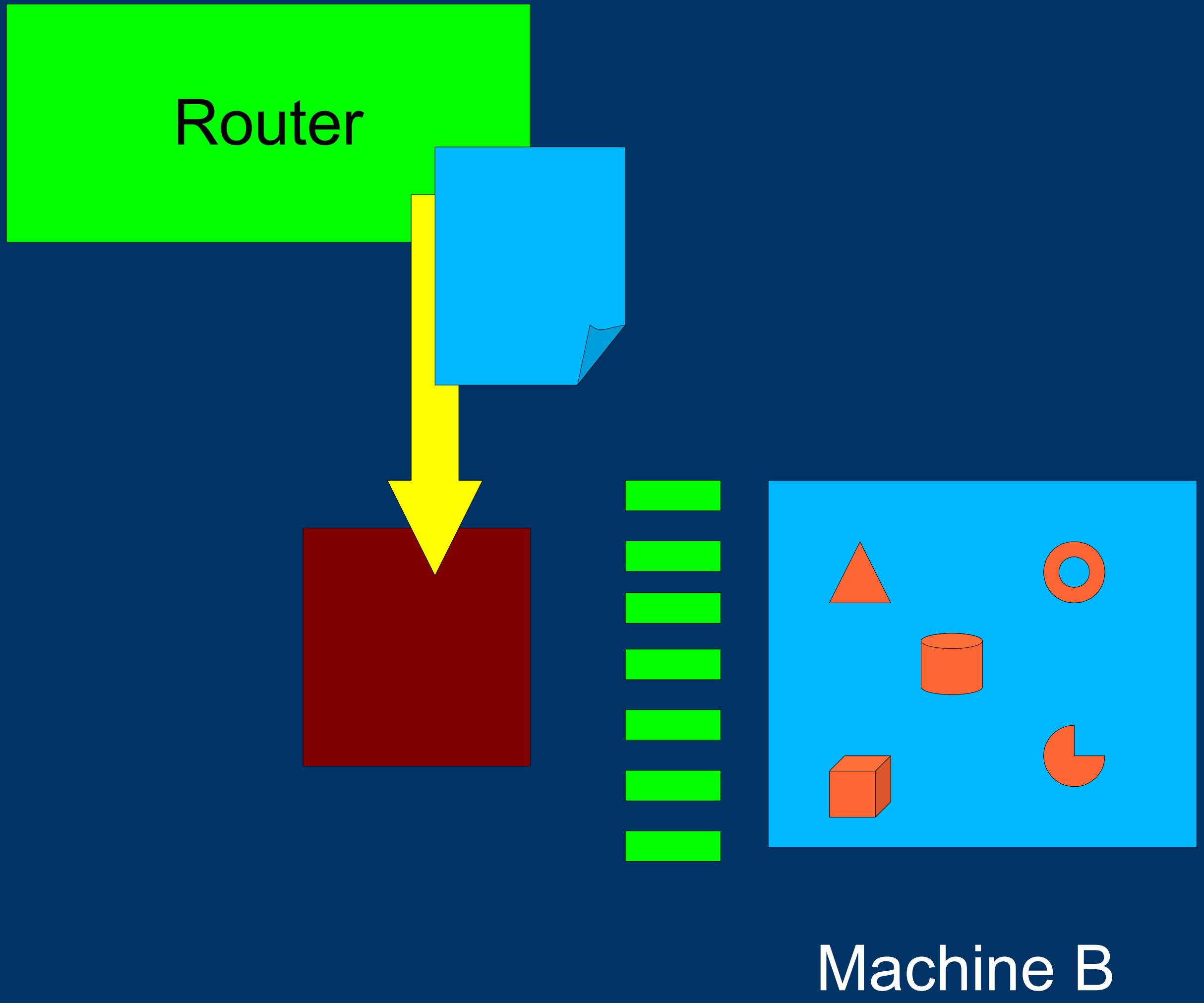
Participating

- Joining is “view only” interaction – the user can not modify the Island contents until he gets per-mission to participate.
- The user must request permission from the Router to participate.
- The router grants participation capability via “facets”
- Interesting aside – we can manage any number of “joined” users simply by broadcasting the message stream to them. This allows arena type interactions.

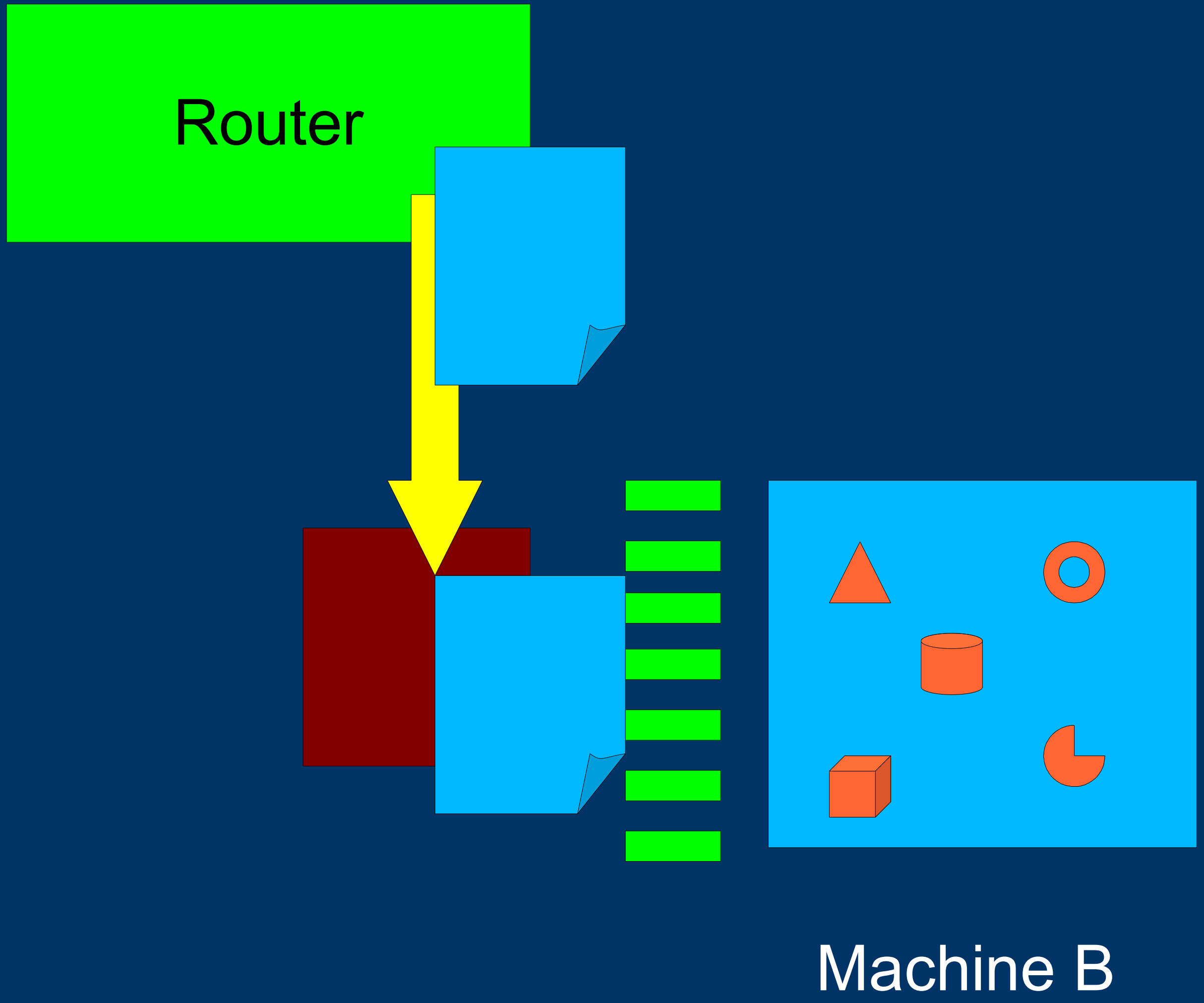
Request right to participate from Router



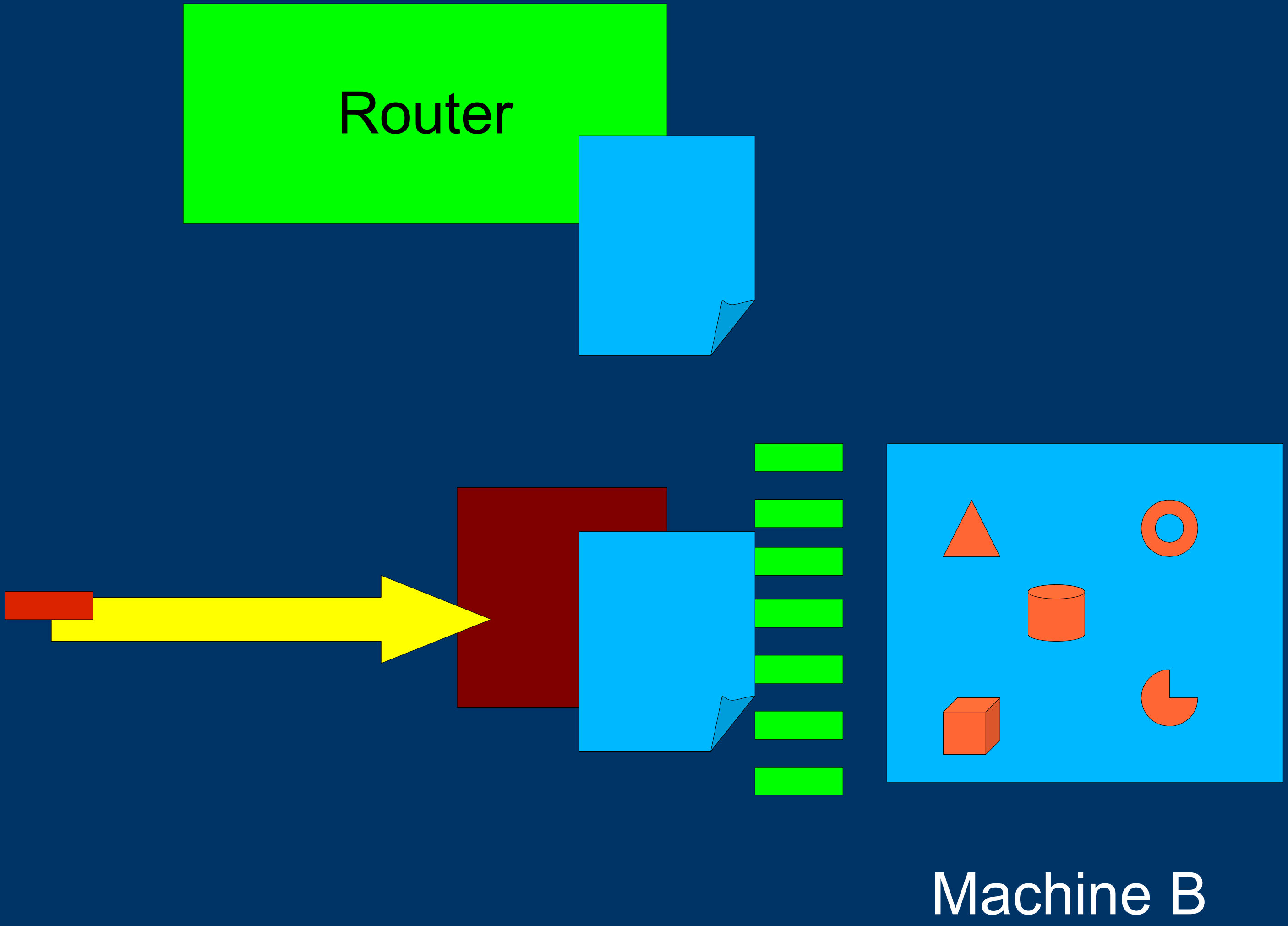
Router passes a list of facets, or interfaces to controller.



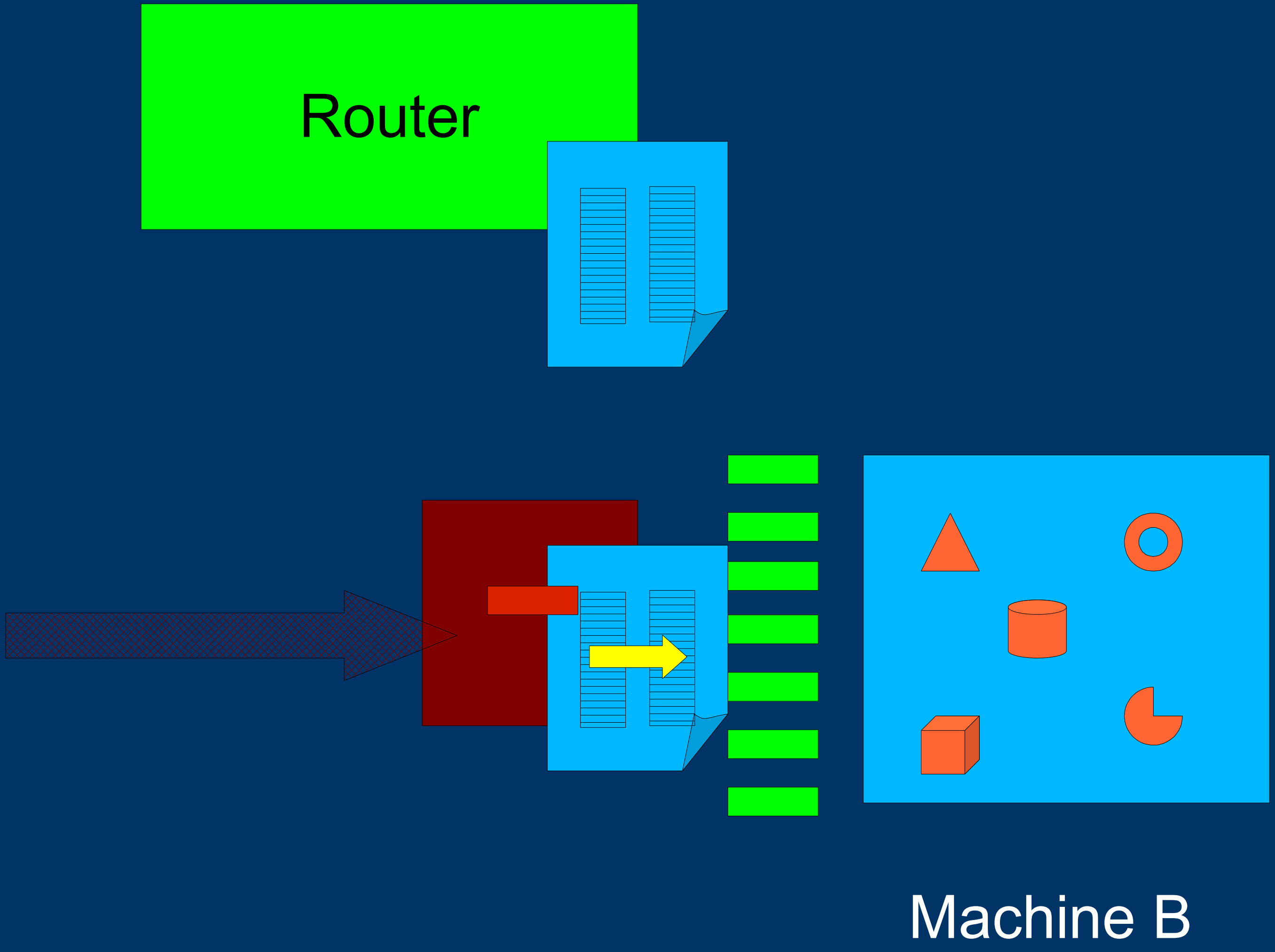
Router passes a list of facets, or interfaces to controller.



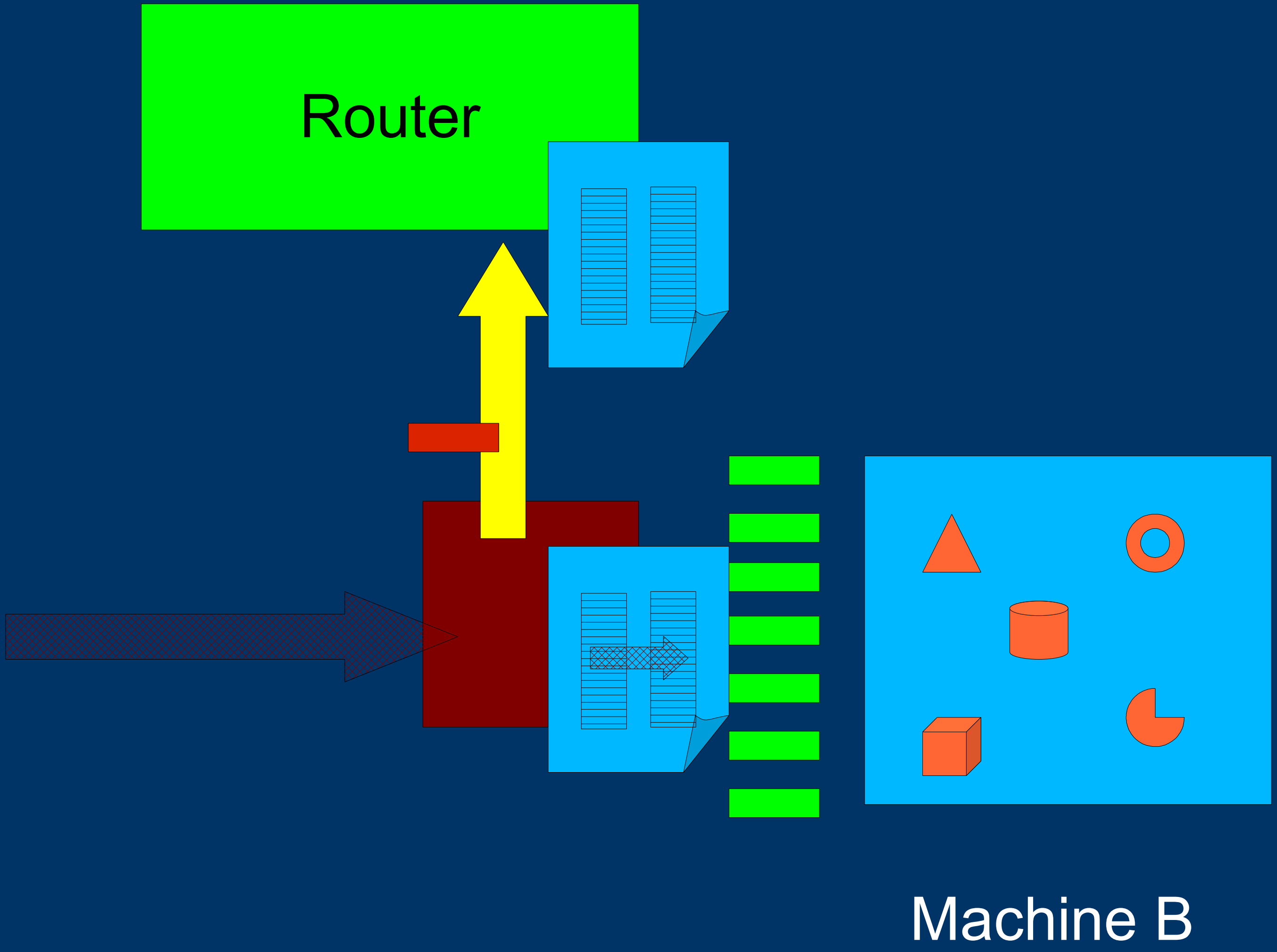
External message is sent to controller.



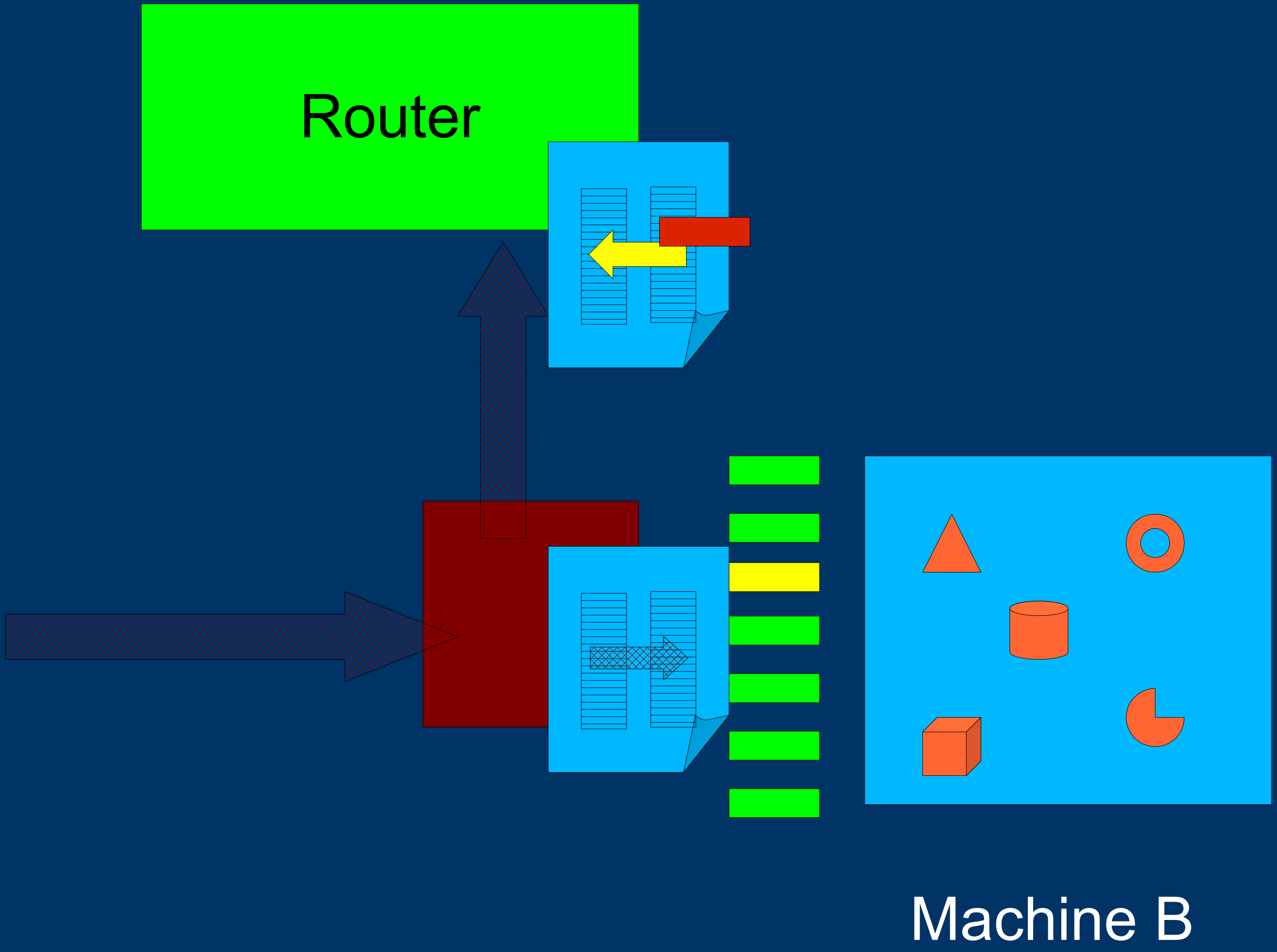
Controller looks up object/message pair in facet dictionary.



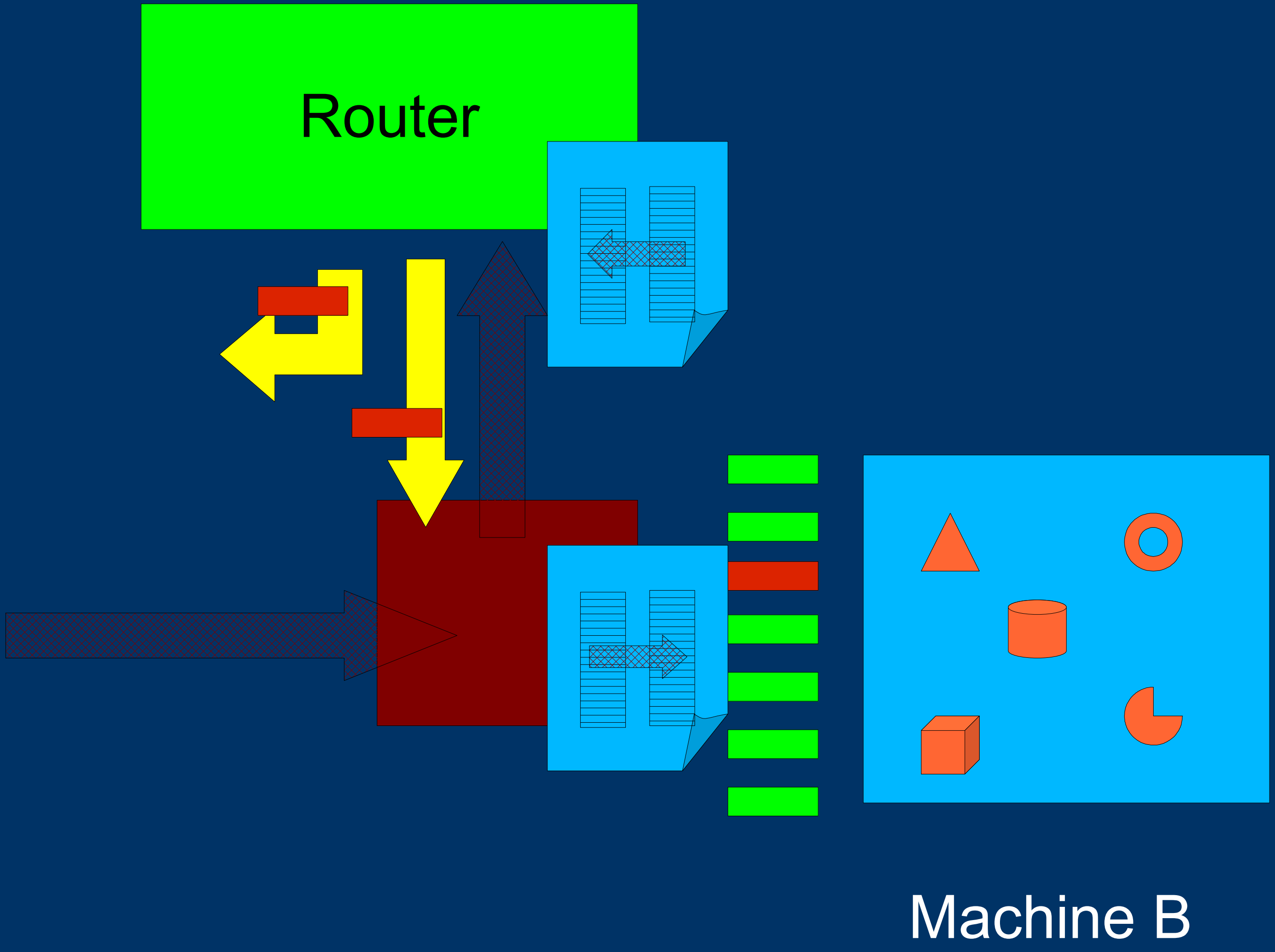
Facet is used to invoke replicated message, sent to router



Router performs reverse look-up to find original message.



Actual message is sent to all controllers.



Capability Facets

- Each facet dictionary is unique to a controller/island pair.
- Different users may have different size dictionaries – they may be granted fewer or more capabilities than other users.
- The controller cannot send a message if it is not in the facet dictionary.
- Facets ensure that only trusted users have ability to modify a given state. They must be explicitly granted this access.
- This does not mean they won't abuse it!

Overlay Portals

- Portals are the main access to Islands.
- They can be overlaid on top of each other such that an island portal may be overlaid with a user interface portal, and system control portal.
- They also contain the user interface objects used to manipulate the contents of an island.

Ghost Objects

- Ghost objects are objects that do not actually exist inside of an Island, but act as if they do.
- They are in a separate island that is accessed by an overlay portal.
- Examples of ghosts might be window frames, UI handles, billboards, and portals into other islands
- This means that though islands are never directly connected to each other, they can still look and act as if they are.

Reference Objects

- Like ghost objects, reference objects are not actually inside of an island either.
- They are objects that are typically not replicated, or are reused across multiple islands.
- Examples of reference objects are:
  - TForms – reused across multiple islands
  - OpenGL objects such as display lists and textures
  - Any kind of non-replicated object required by an Island.
- When we render a scene with a reference object, only the opengl object is aware of it.

Rendering is not replicated!

- Rendering occurs locally to a machine. It is a non-replicated process, hence does not employ the #future message. Instead, it is one of the few places where we use the #send: pattern described earlier.
- We are exploring a write-protect capability to ensure that the island state is not modified by the render process.

Graphics and Transforms

- The TFrame based graphics architecture is being cleaned up and simplified.
- In particular, looking at immutable object management (textures, meshes) via worldbase servers
- Will be incorporating the new texture management and shader capabilities being developed at UofW (using the reference architecture of course)
- Matrices are being cleaned up. Will be adding a real scale capability though it will be symmetrical.

Tweak

- The new replicated island model dramatically simplifies making Tweak and the 3D aspects inter-operate properly.
- Next release will begin to take advantage of Tweak both as the basis of UI and as a development platform.
- Morphic will be going away as soon as Tweak's development tools reach a critical mass (soon).

What is the difference between TT and STT

- TT presumes majority rules, and allows for aborted message sends. STT elects a king – properly sent messages must be executed.
- TT manages interactions between TeaParties to minimize replicated messages. STT does not allow messages between Islands.
- TT is actually a new model of computation that includes time as a central aspect. Time is a continuous function. STT model of time is discrete, time is quantized.
- Coding to STT should work with TT with few or even no changes.