A Virtual Worlds Architecture Framework VWF

David A. Smith
Chief Innovation Officer
Senior Fellow
Lockheed Martin GTL
david.alan.smith@lmco.com

VWFVision

Frank DiGiovanni, SES
Director, Training Readiness and Strategy

Office of the Deputy Under Secretary of Defense (Readiness)

The Pentagon, Washington D.C.

Summary

- What it is.
- Why it matters.
- Demonstration.
- How it works.

Essential Features of a Next Generation Virtual World Platform

- **Platform scalability** – need to work across OSs and devices (desktops – handhelds)
- **Distribution** – Needs to be easily deployed across an entire organization with a 0-install preferred. Needs to work on both sides of a firewall.
- **Security** – needs to work with existing IT capabilities and requirements
- **Open source** – must avoid being a captive solution. Must be accessible to non-business users, especially education
- **Utilize and Define Standards** – for interoperability and scalability. Must interoperate and fully enable the Global Information Grid (GIG)
- **Future Proof** – must scale dynamically with new requirements and new opportunities while protecting investments in content and infrastructure
- **User Base** – must be accessible to large population of developers.
- **Business models** – must provide interesting business ecosystem for small and large organizations. Must lower the cost of content development while raising the level of quality and affordability.

The Web is the Future of Virtual Worlds

- The next generation of browsers incorporating WebGL coupled with HTML 5 will become the de facto foundation for the next generation of shared virtual worlds.
- The next big 3D platform is simply the current WWW with additional capabilities.

Browser
+
WebGL
+
HTML 5
+
JavaScript
+
Jabber
+
Collada

VWF

- Replicated computing platform for multi-user interactive 2D and 3D
- Focused on training, collaboration and entertainment
- Component based model – intelligent objects that can be easily added to existing spaces
- Model view architecture to allow multiple rendering models (2D and 3D) as well as dead-reckoning models
- Open Source platform

Release

Alpha released November 2011

Beta released March 2012

Available here:

http://www.virtualworldframework.com

Hosted on Github.

Includes:

- **Full working system**
- **Online working Demo applications**
- **Demo Servers also distributed as running VMs**
  - Apache (in progress)
  - **Ruby (available now!)**
  - NodeJS (in progress)
- **Documentation (in progress)**
  - System docs
  - Developer docs
  - Cookbook

Where?

http://www.virtualworldframework.com

https://github.com/virtual-world-framework/vwf

Demos

Components

- YAML or JSON based
- Goal is Drag and Drop – user extendable
- Programming does NOT require deep multi-user networking knowledge
- May require managing multiple user events – basically matching mouse, keyboard etc to a given user
- Support both 2D and 3D

Model/View

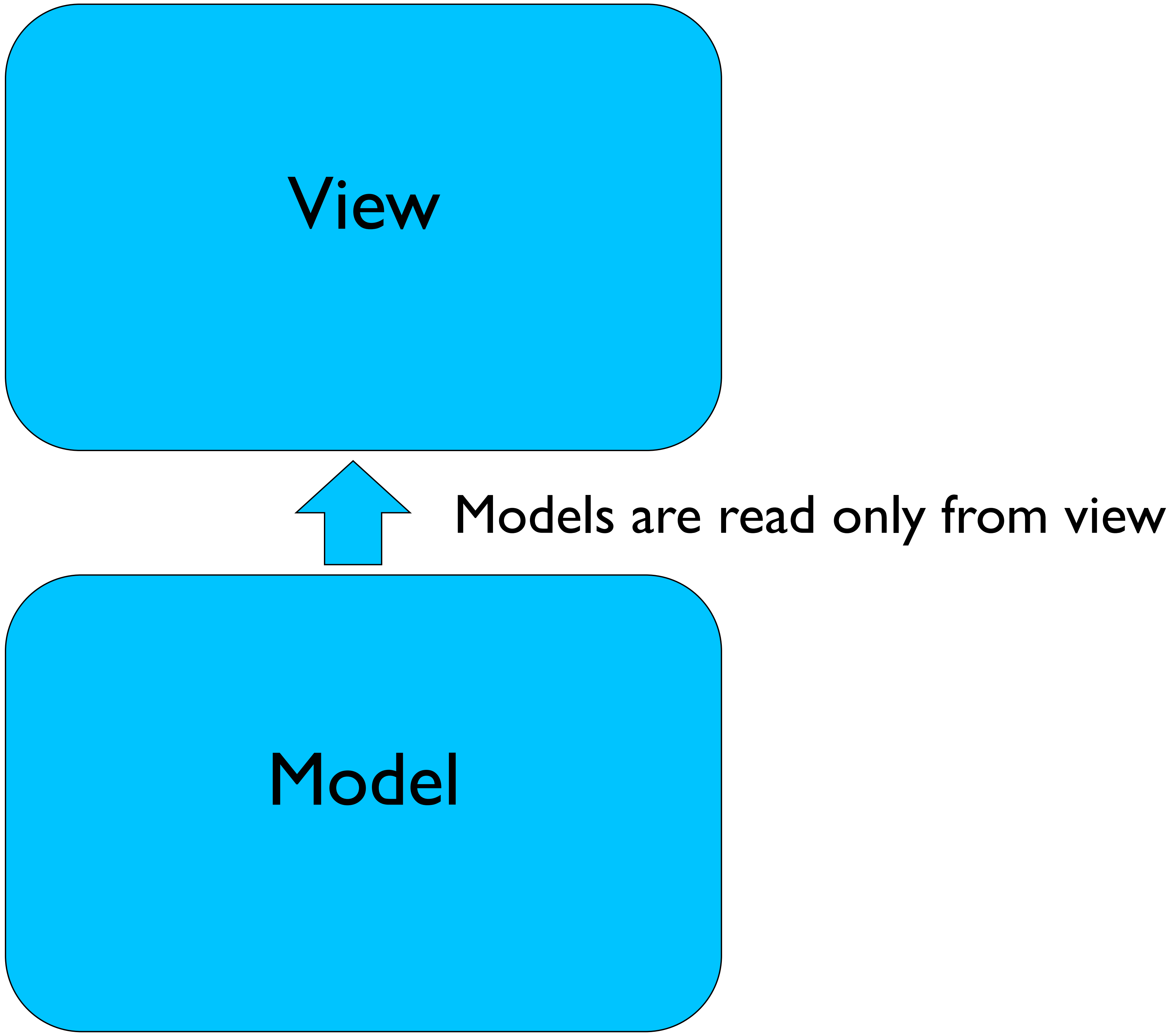
View

Rendered image
Can be 2D, 3D, WebGL, Unity
Allows for approximate representation of system state (eg dead reckoning)

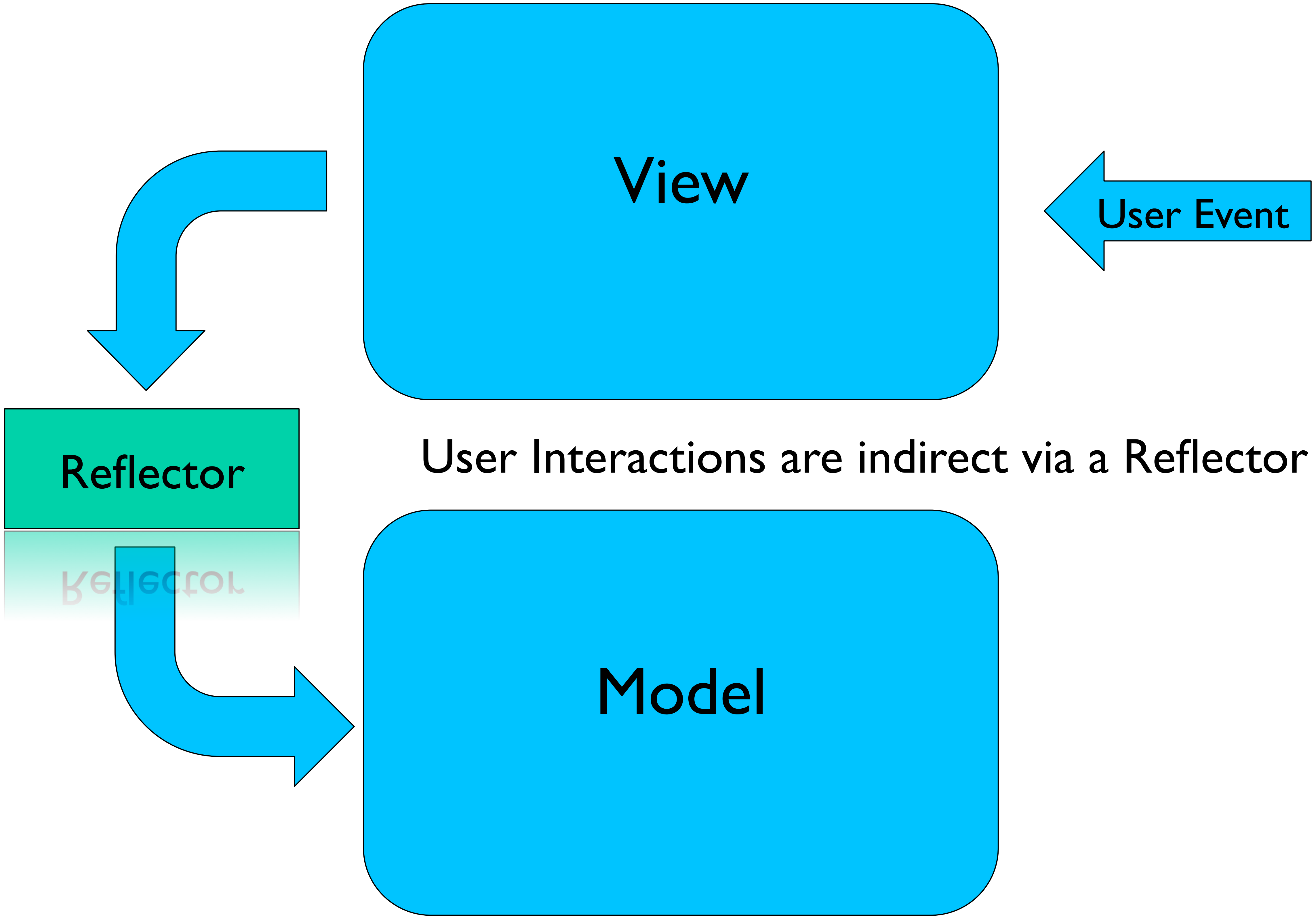
Model

Replicated state
Provides replicated "truth"
Deterministic Simulation Platform

Model/View

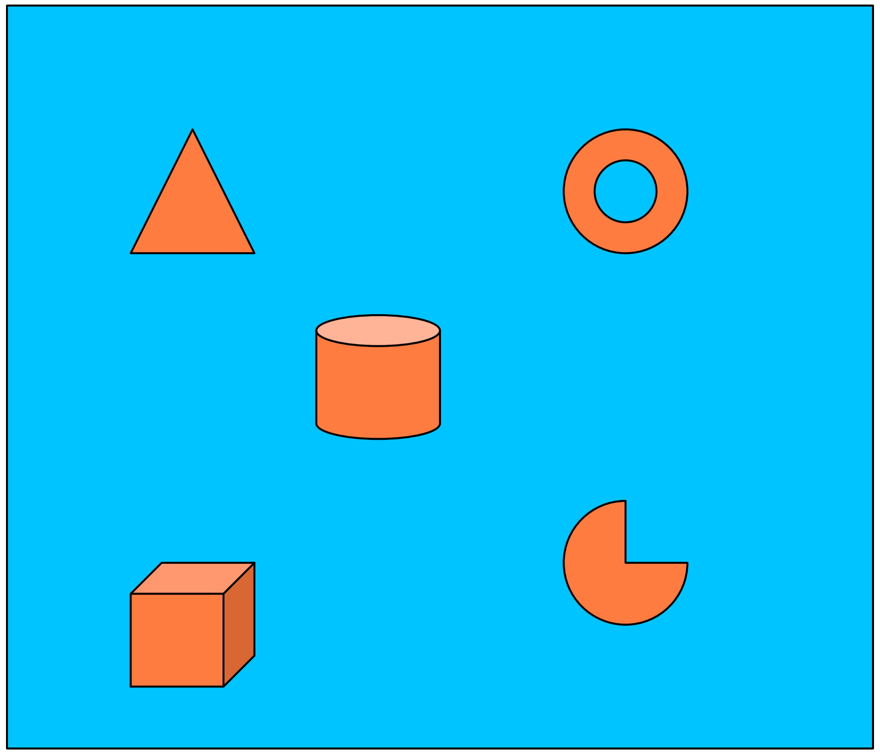


Model/View



User Interactions are indirect via a Reflector server

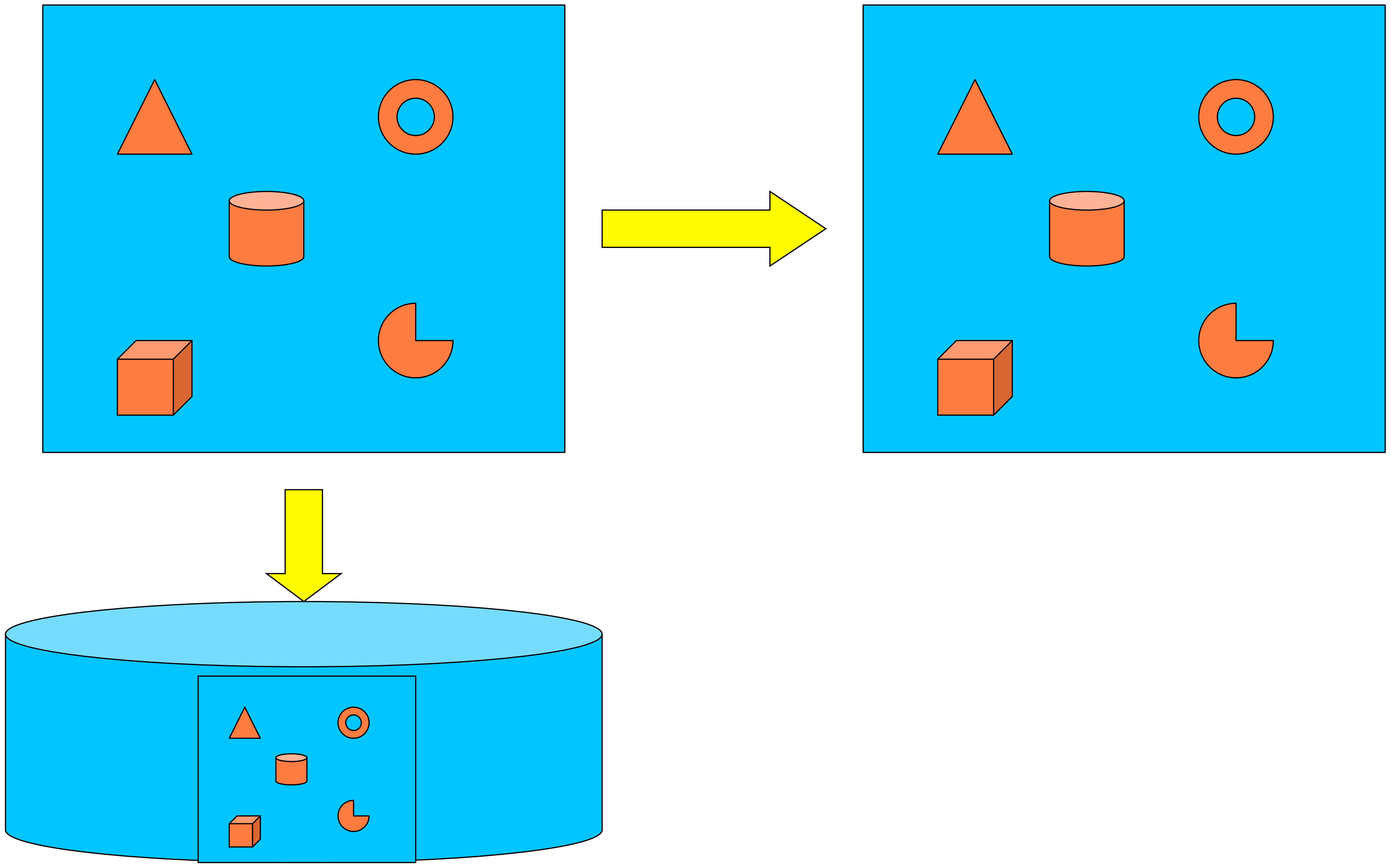
Bobbles* are "safe" replicated containers of other objects



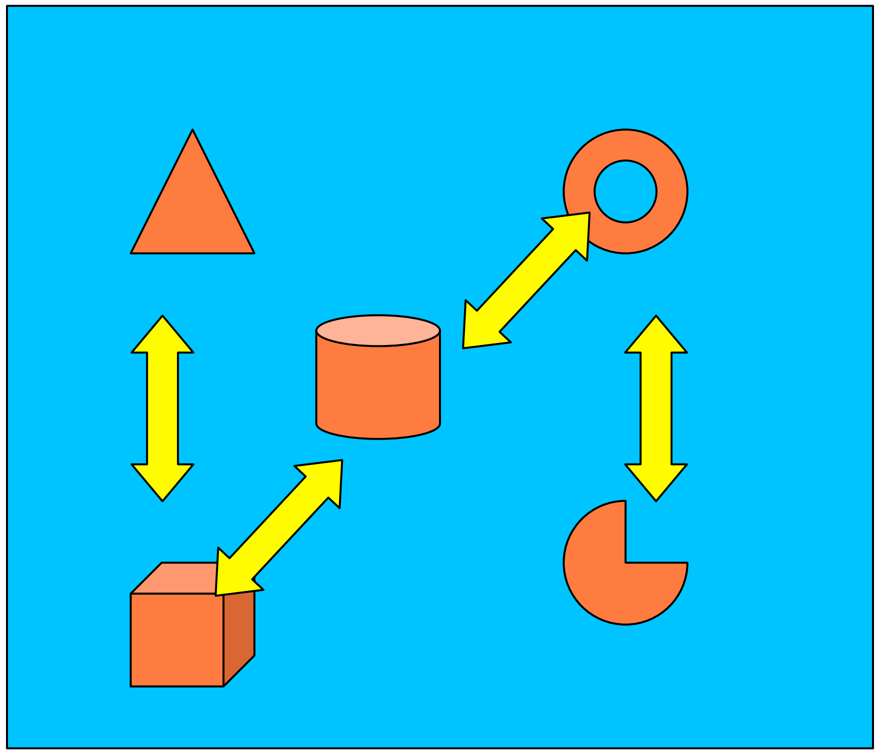
- Any component can be a Bobble
- Bobbles can also contain other components

* Courtesy of Vernor Vinge

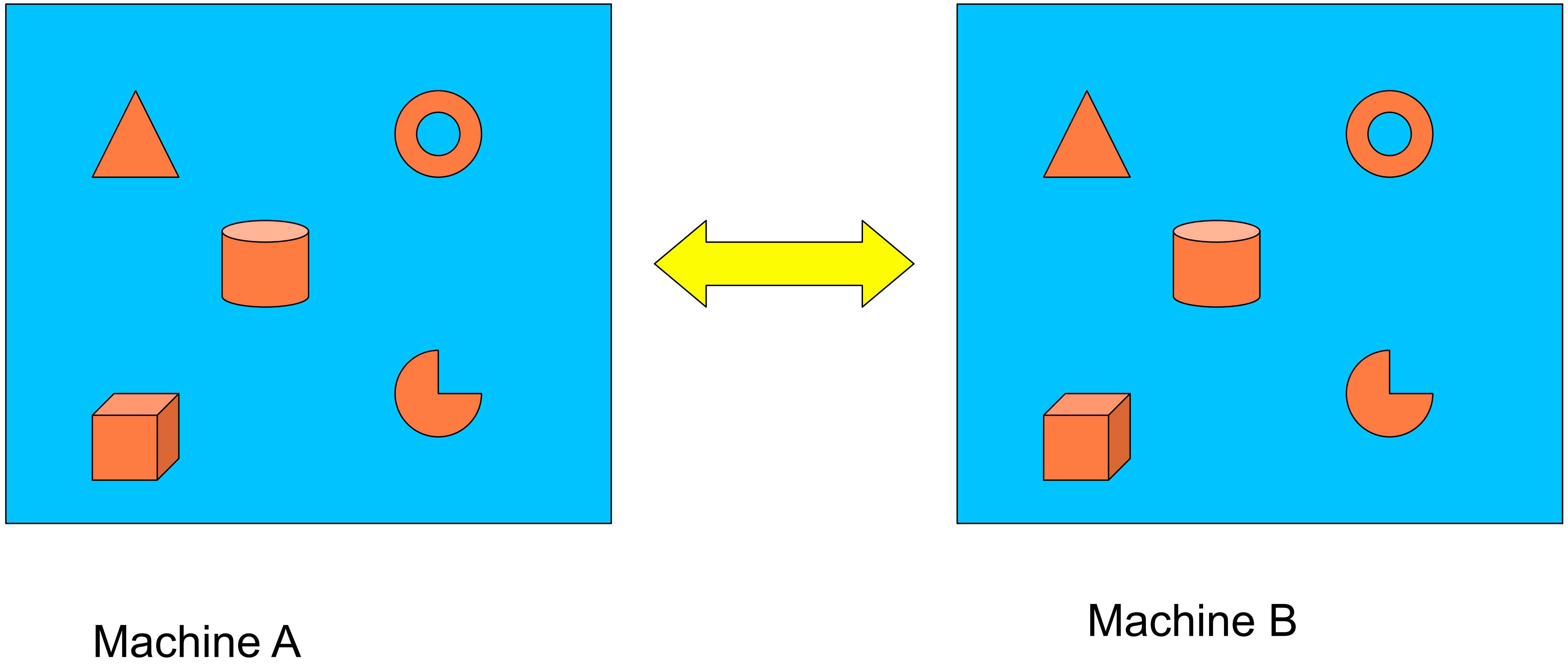
Bobbles can be easily saved and duplicated via JSON or YAML



Objects and components can interact directly with other objects within Bobbles directly



Replicated Bobbles



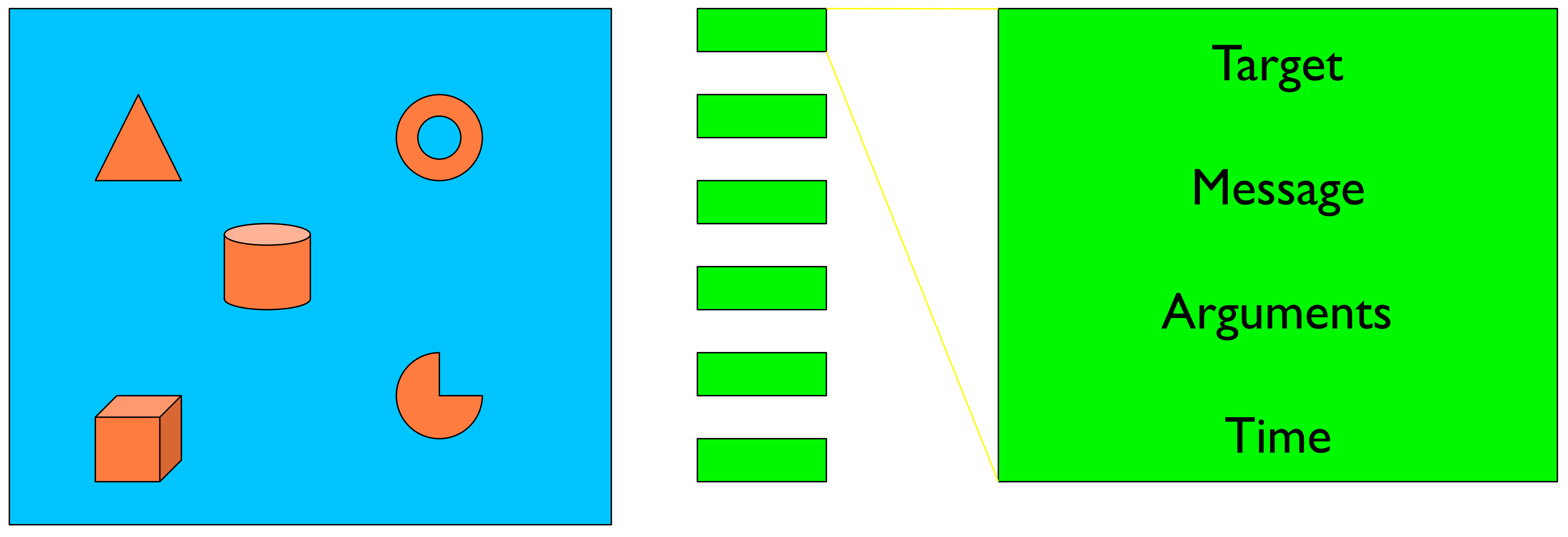
Replicated Bobbles

- Deterministically Equivalent
- Bobbles replicated via checkpoint mechanism
- Internal Future messages implicitly replicated
- External Future amessages explicitly replicated
- External non-replicated messages VERY bad
- New objects: Reflectors and Controllers

Timing is Everything!

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

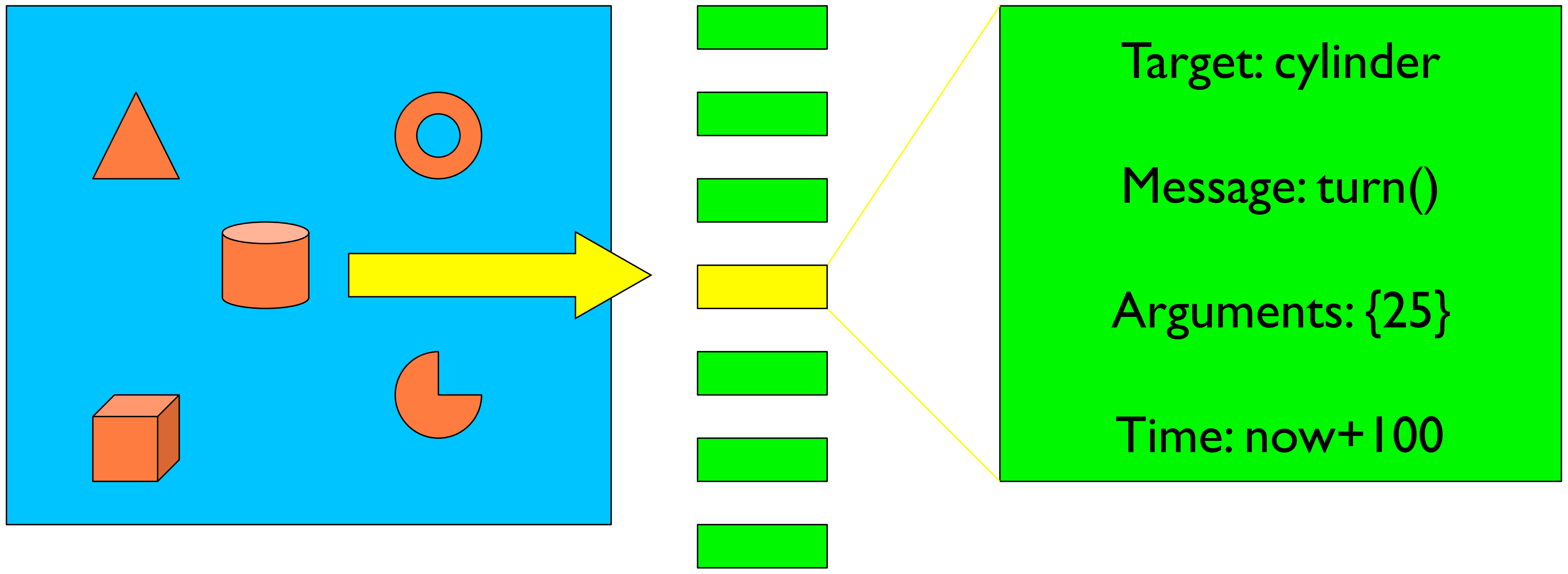
Bobble's View of Time is defined only by message order!



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 (temporal tail recursion):

```
ACylinder.aMessage(arg)

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

function doSomethingWith(arg){
        if (arg>0){
                self.spin(10);
                self.future(100).doSomethingWith(arg-1);
        }
}
```

Reflectors, Controllers and Genuine Time Based Replication

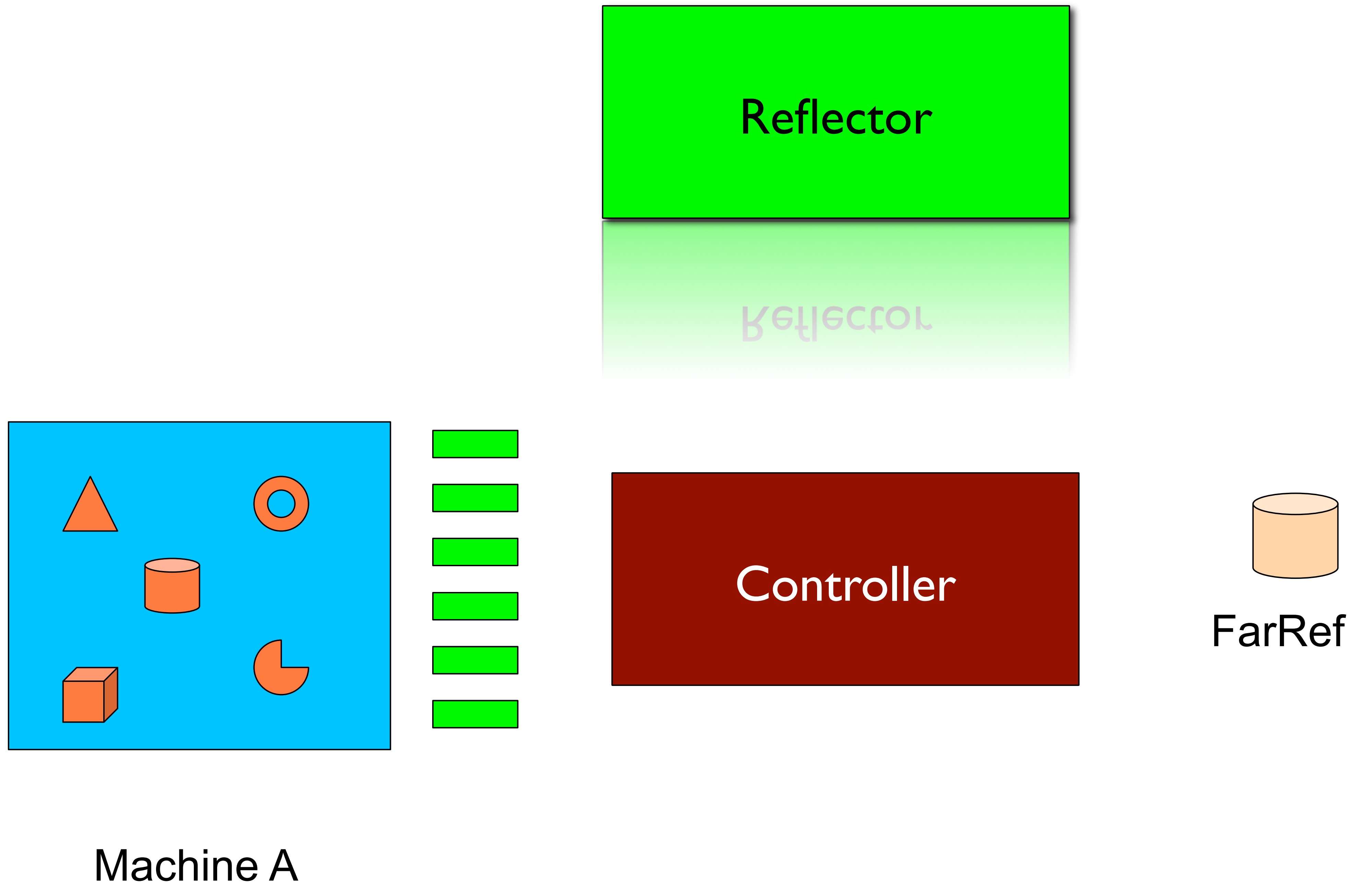
The Reflector

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

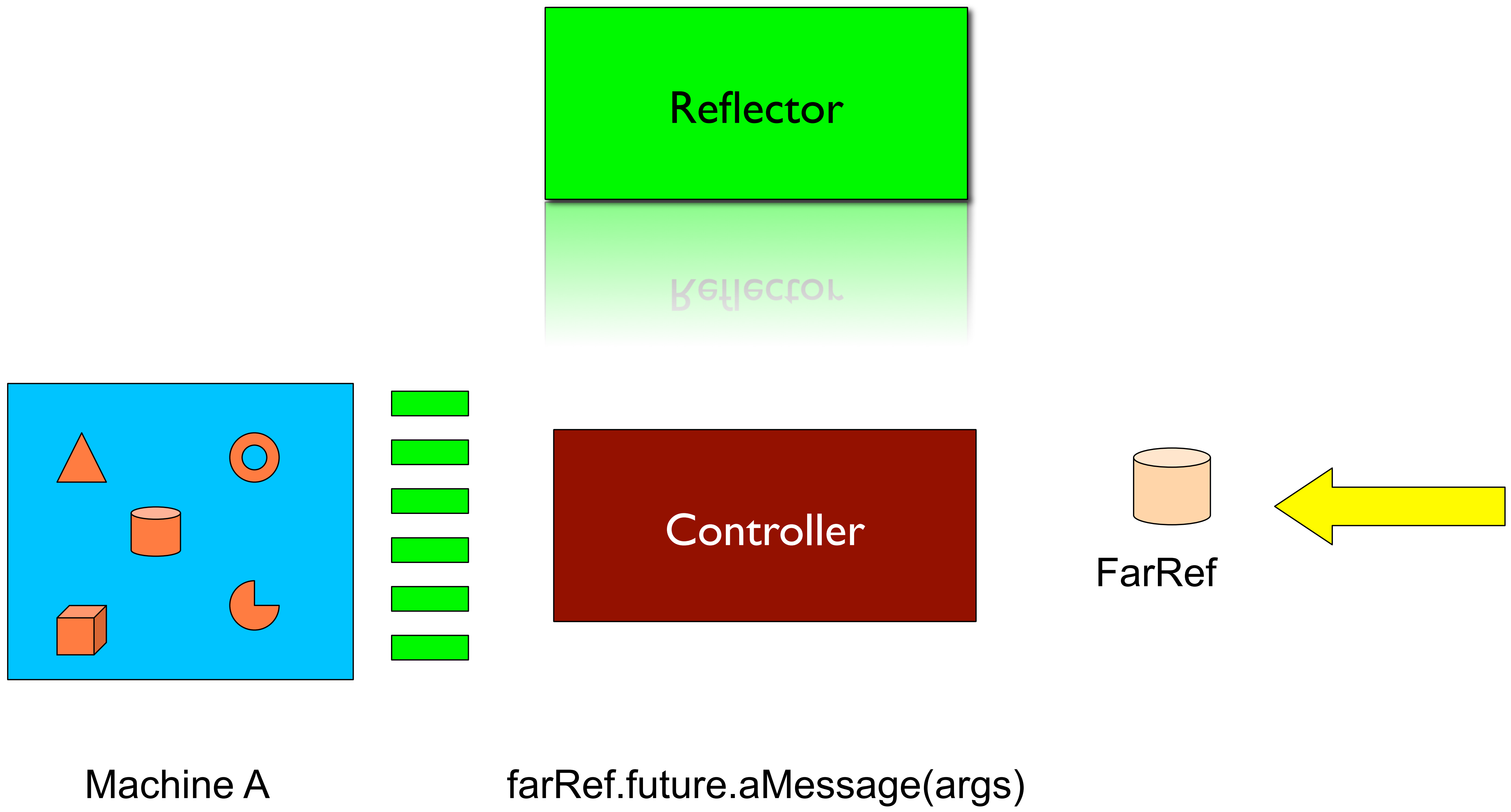
The Controller

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

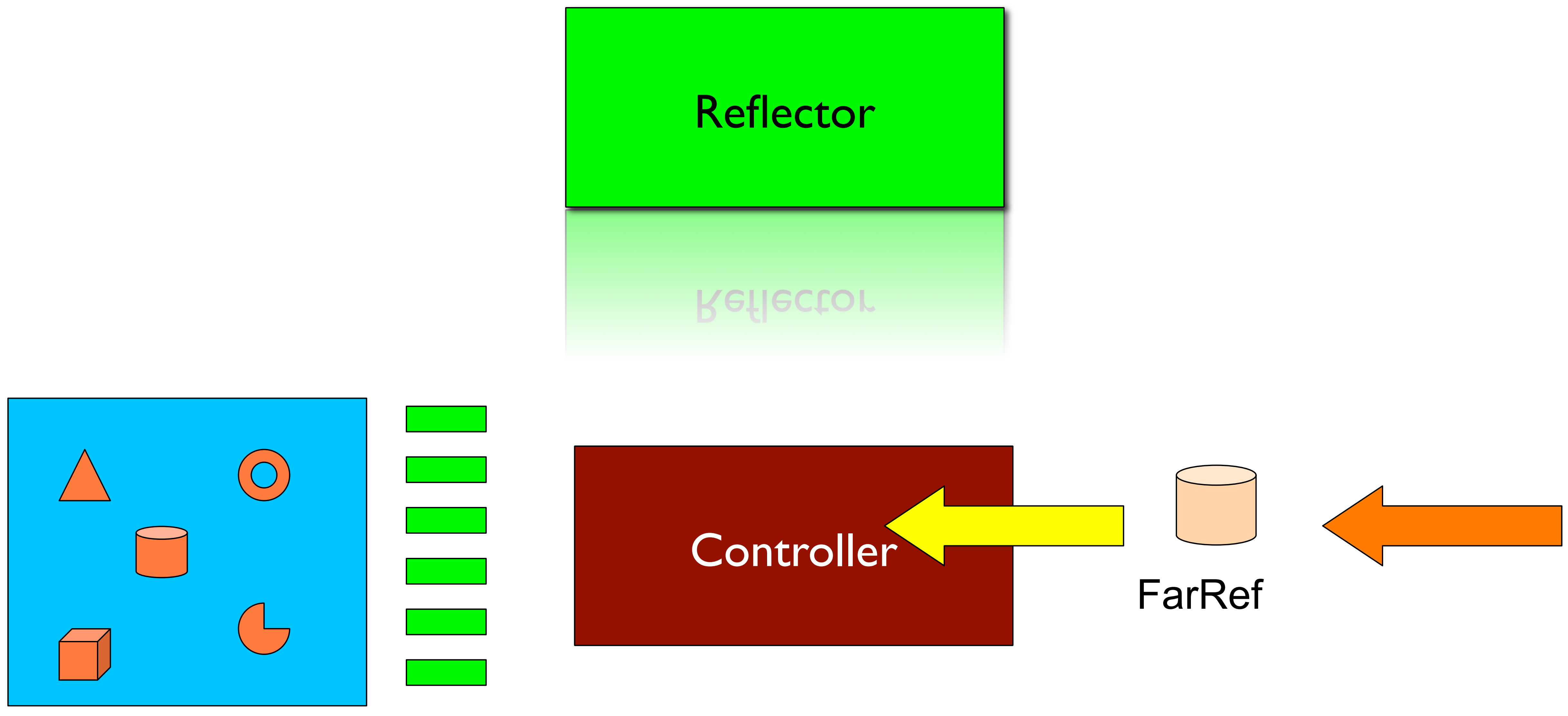
The Reflector/Controller



Message sent to farRef – no time is specified



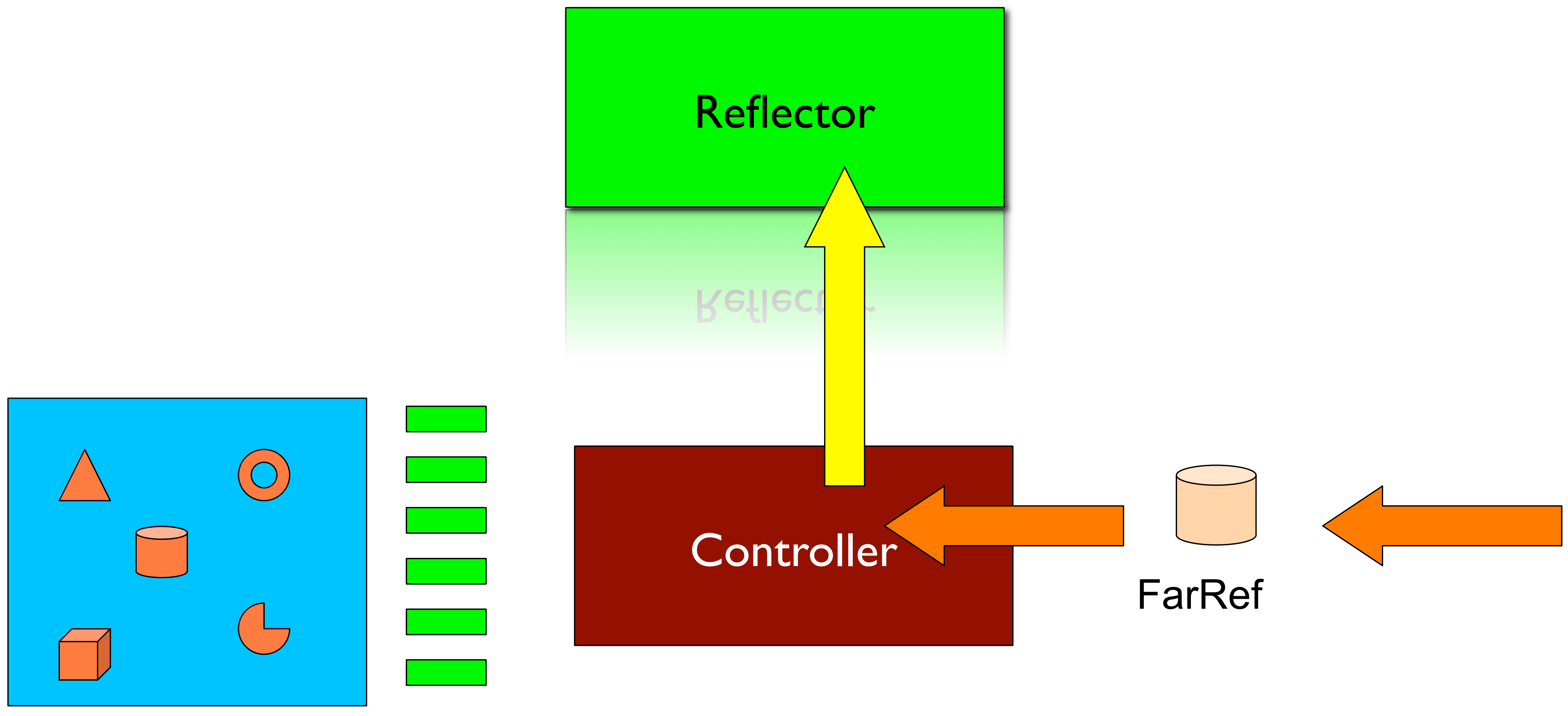
farRef forwards to controller



Machine A

farRef.future.aMessage(args)

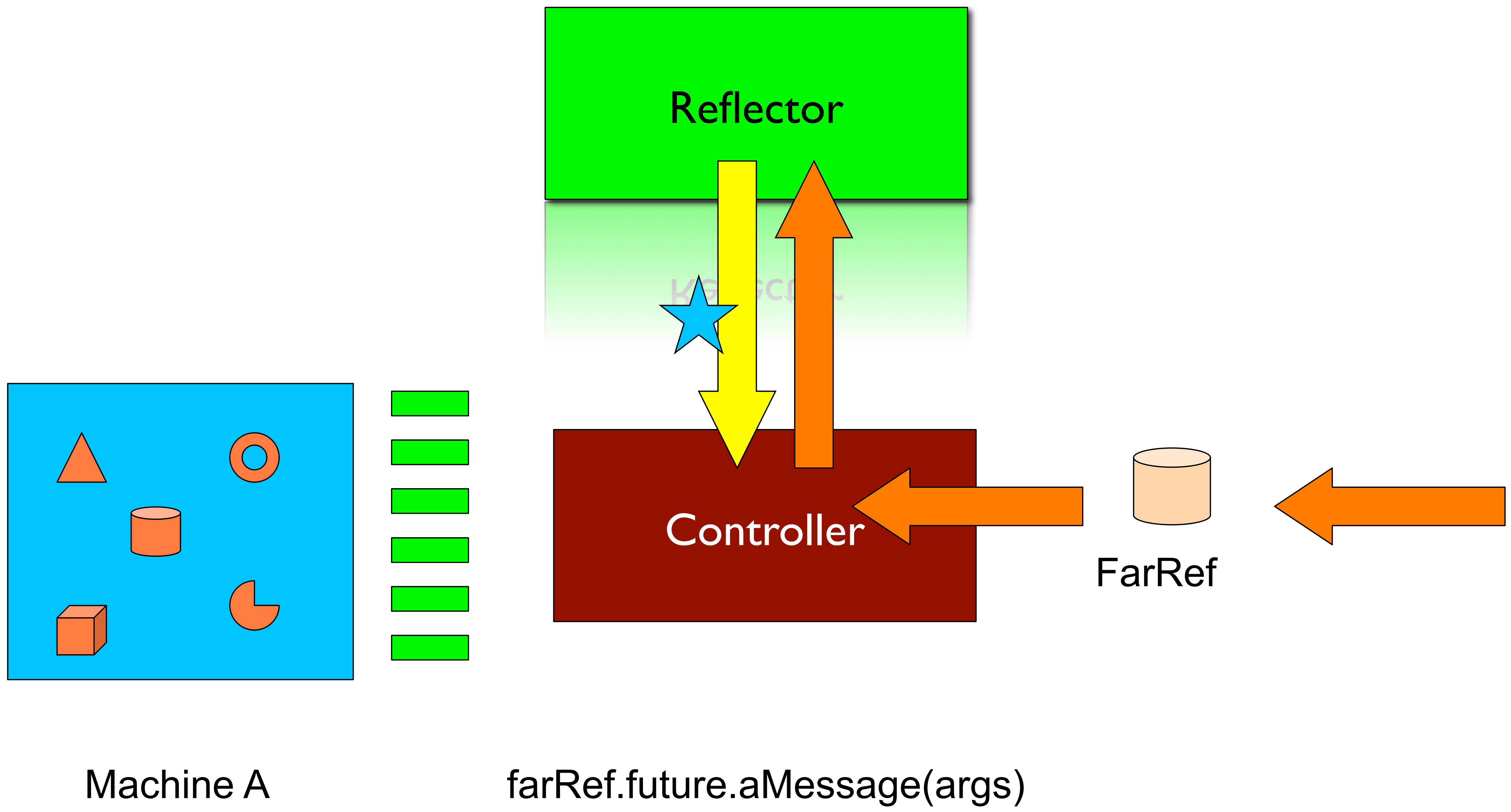
Controller forwards to Reflector



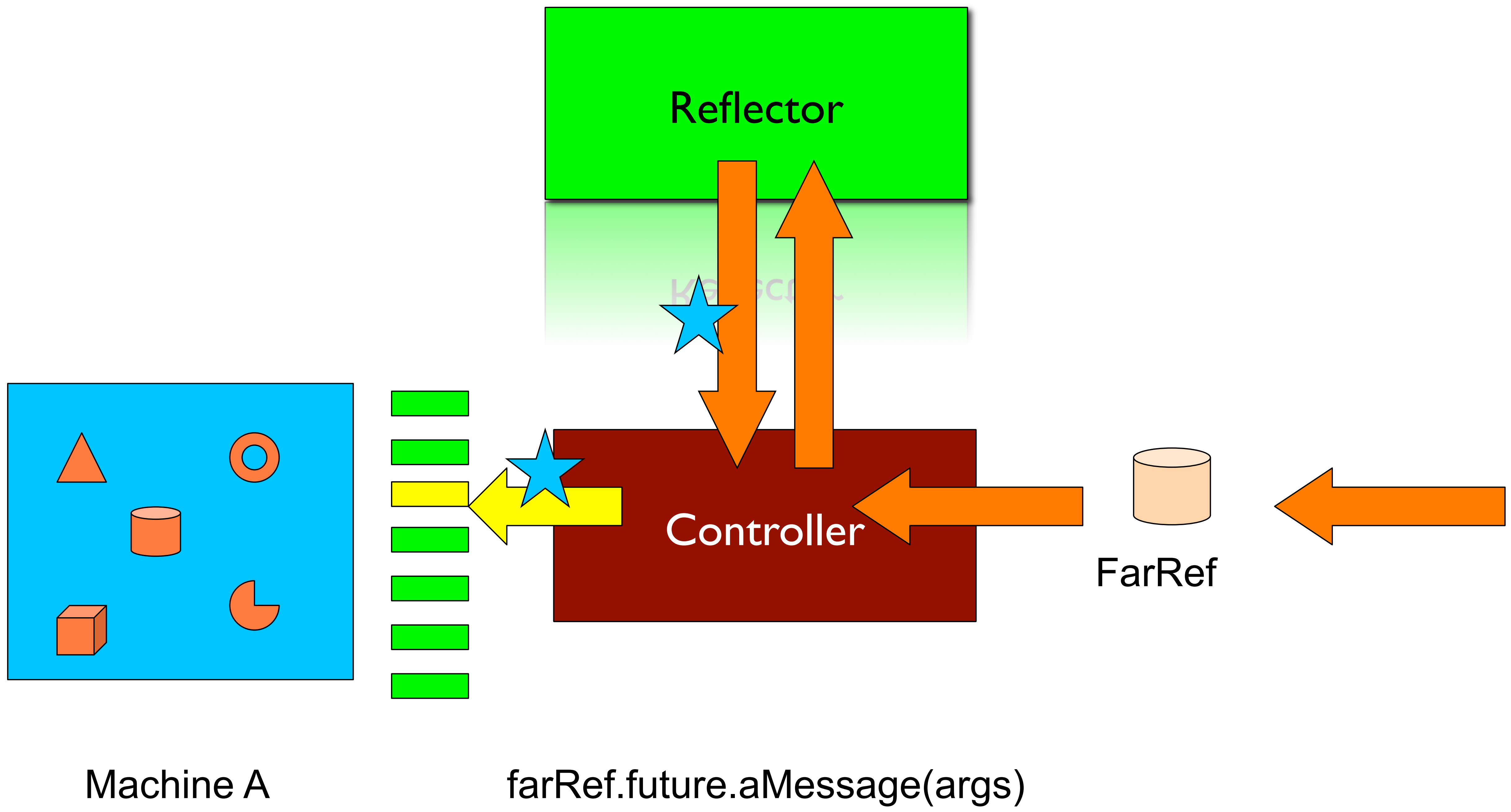
Machine A

farRef.future.aMessage(args)

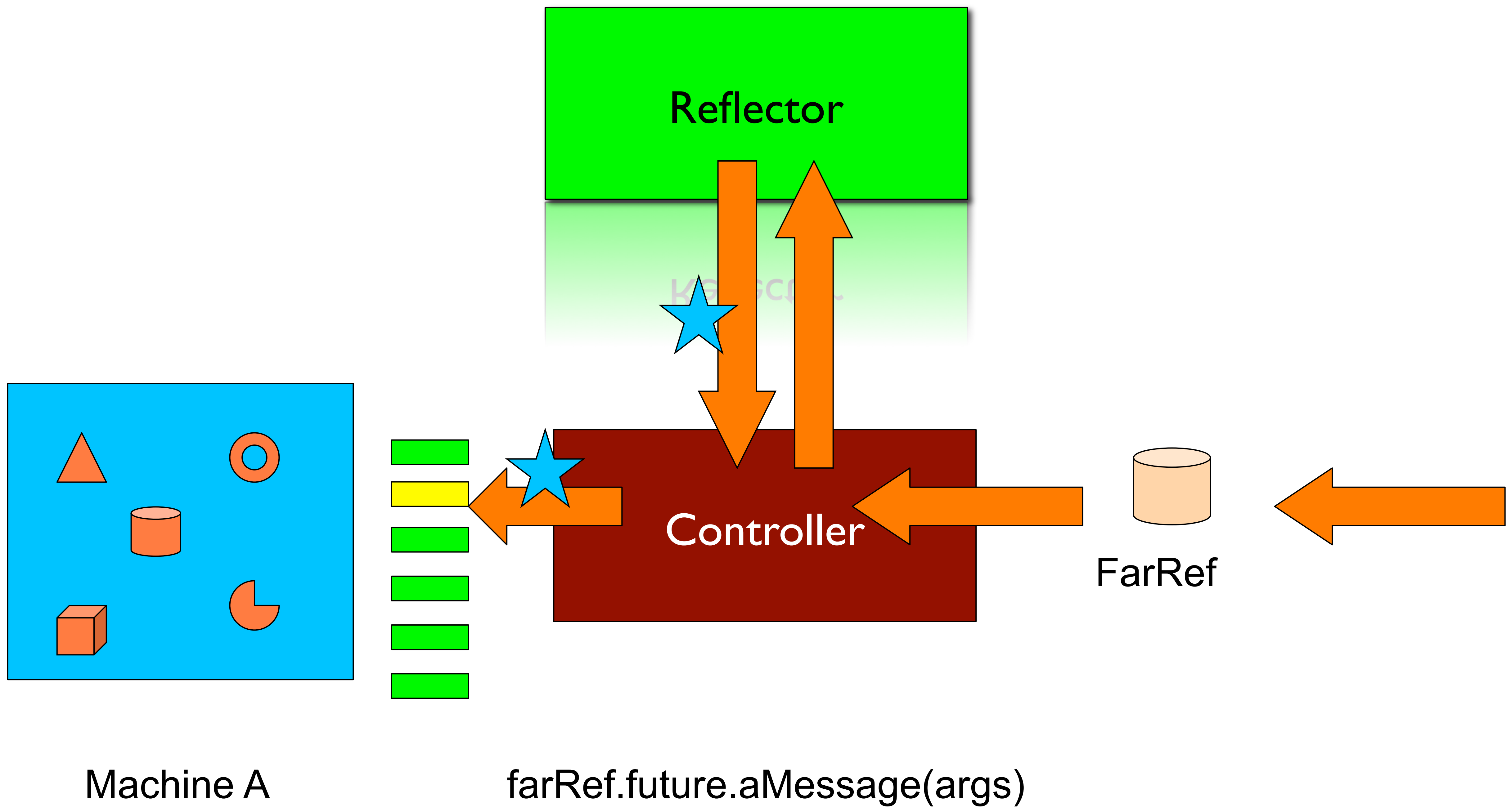
Reflector adds time stamp (and enumeration), forwards back to controller



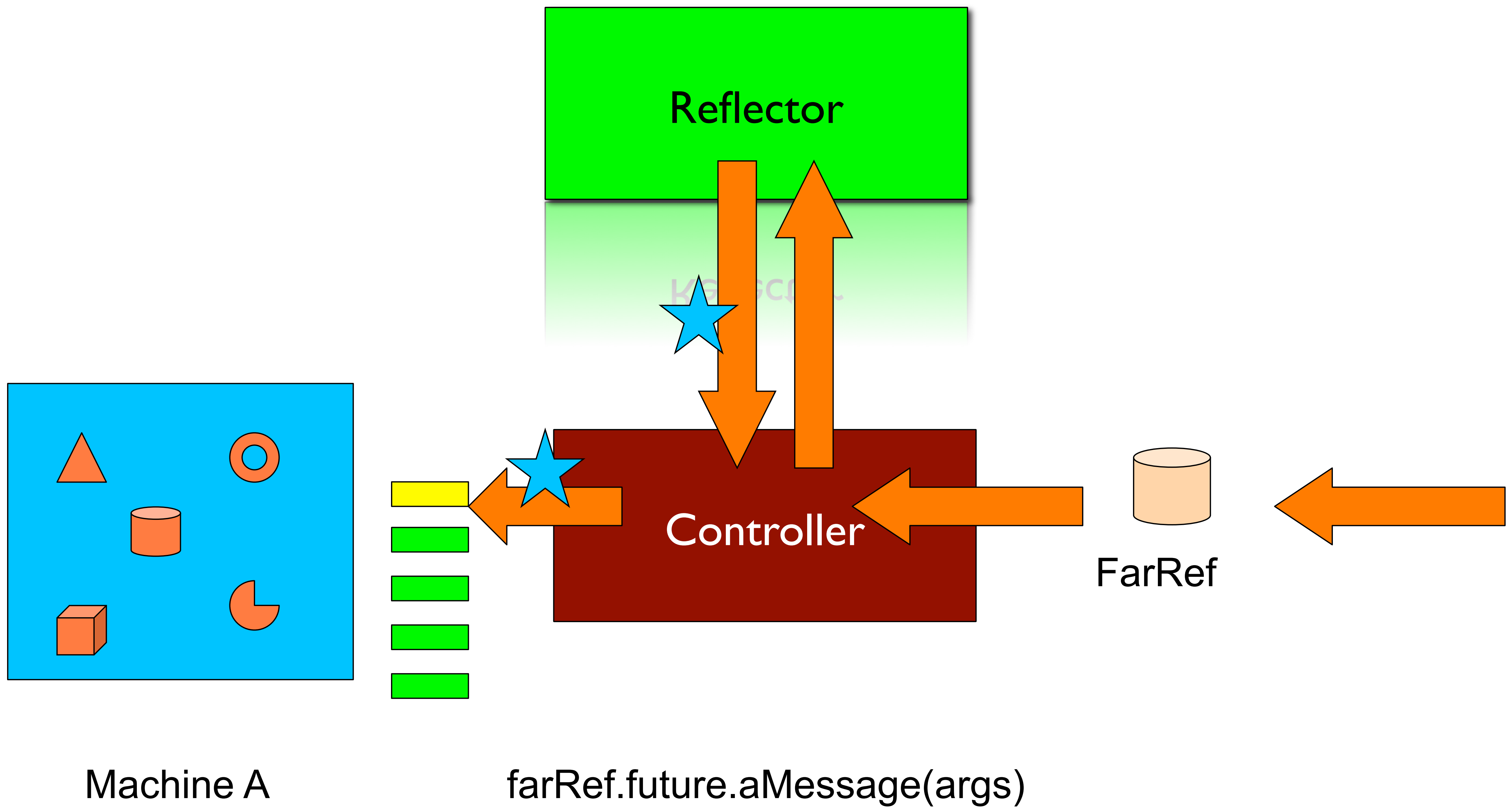
Controller forwards time-stamped message to add to message queue.



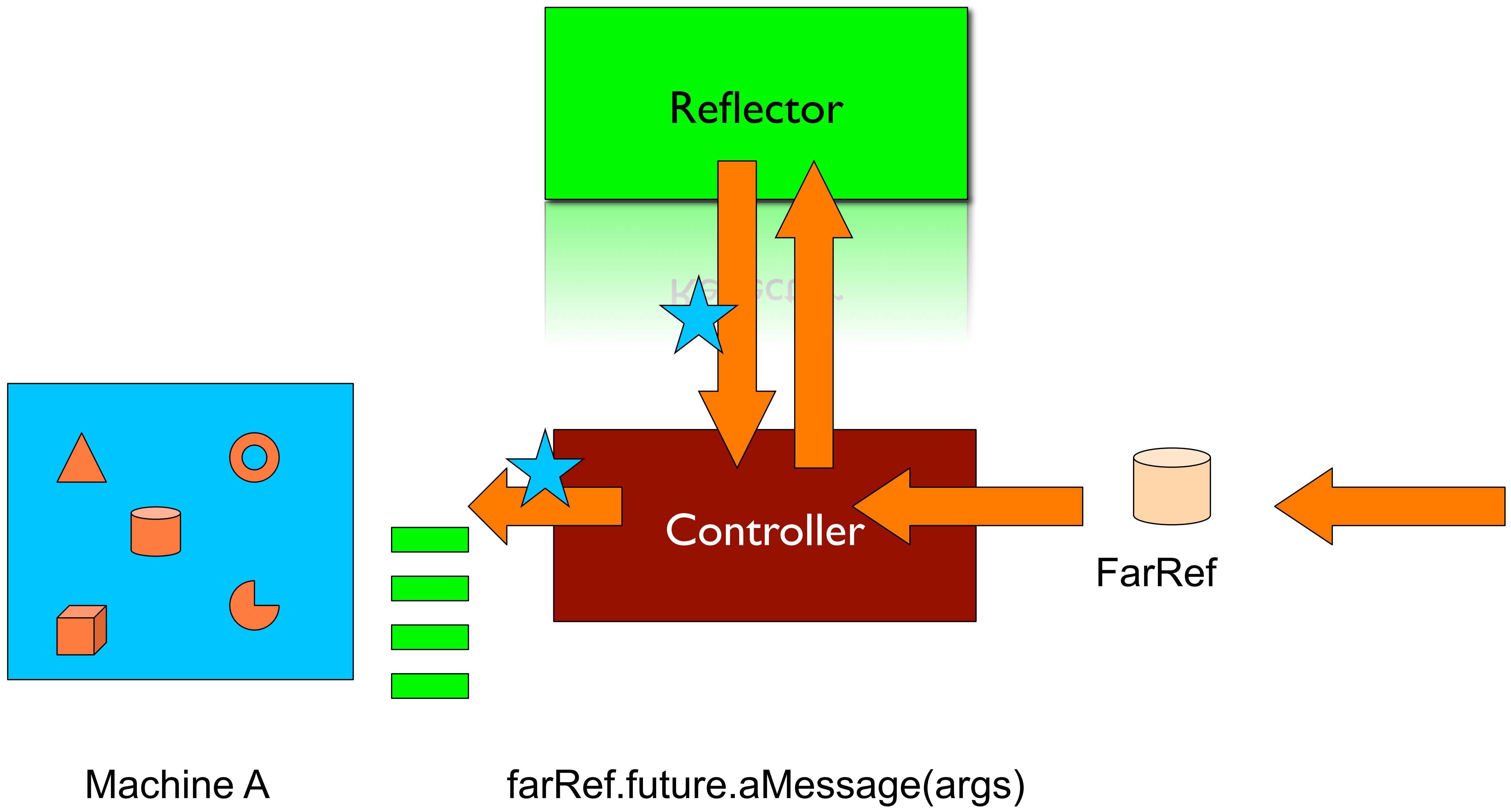
Bobble executes all messages up to the new external message



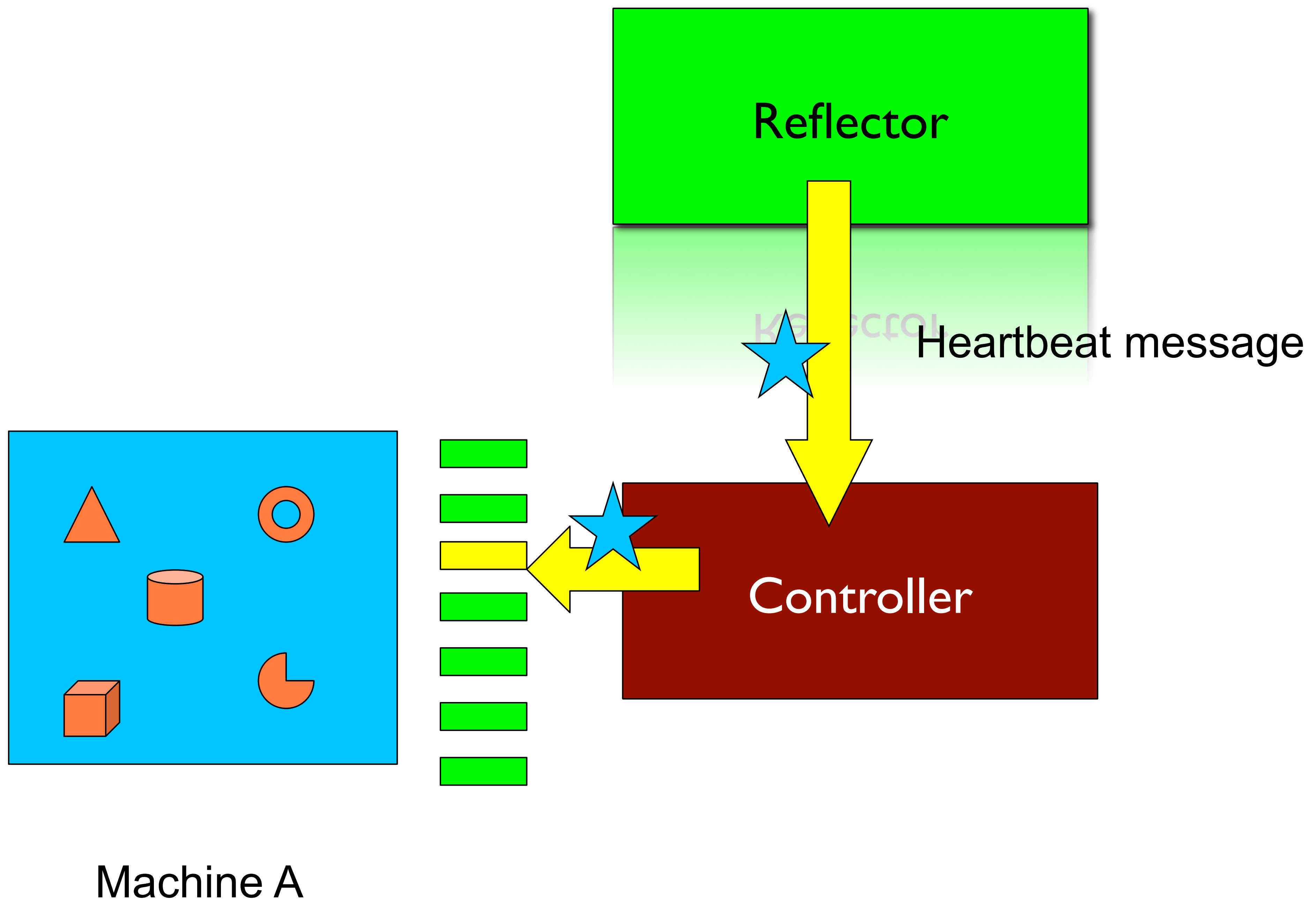
Bobble executes all messages up to the new external message



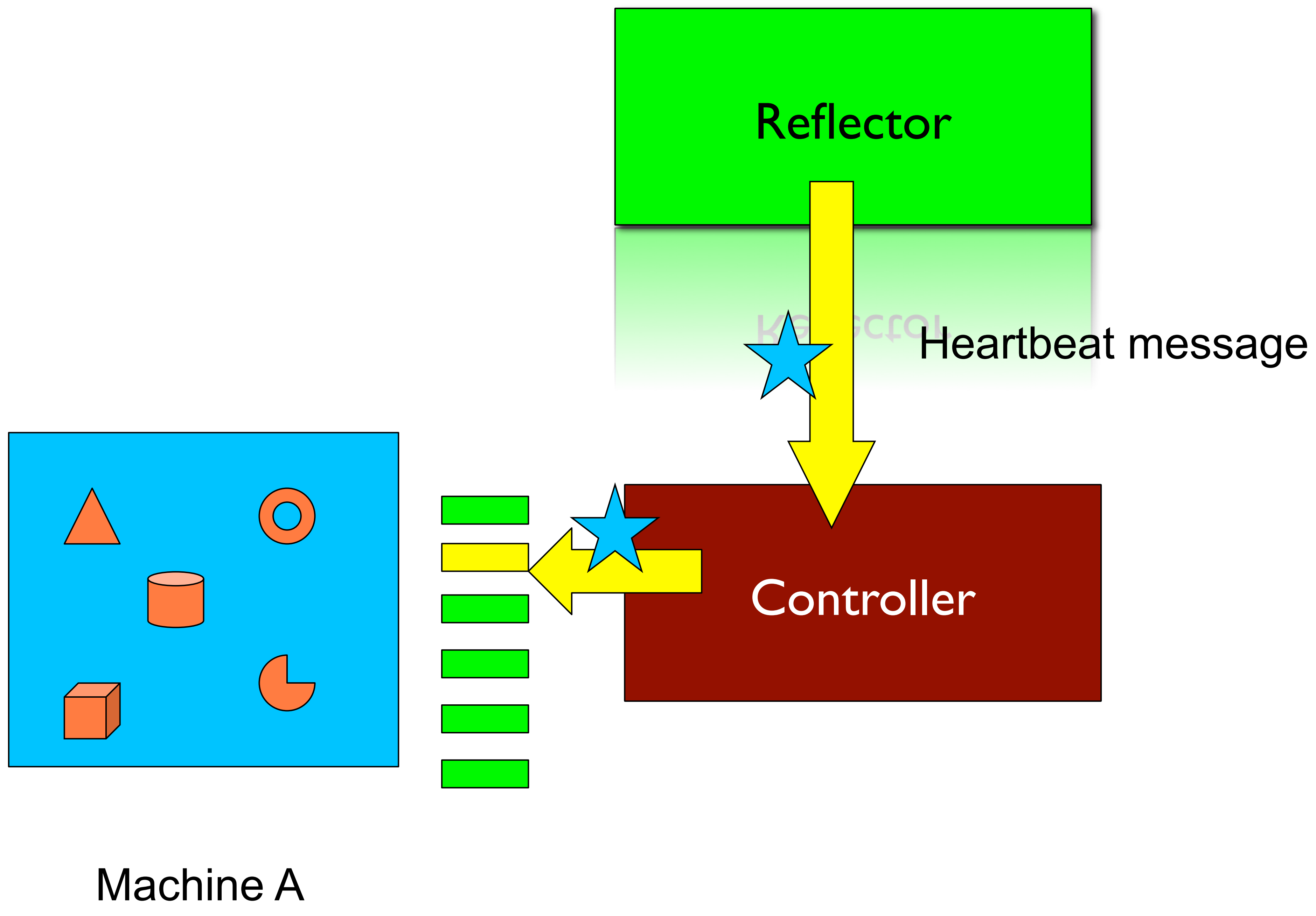
Bobble executes all messages up to the new external message



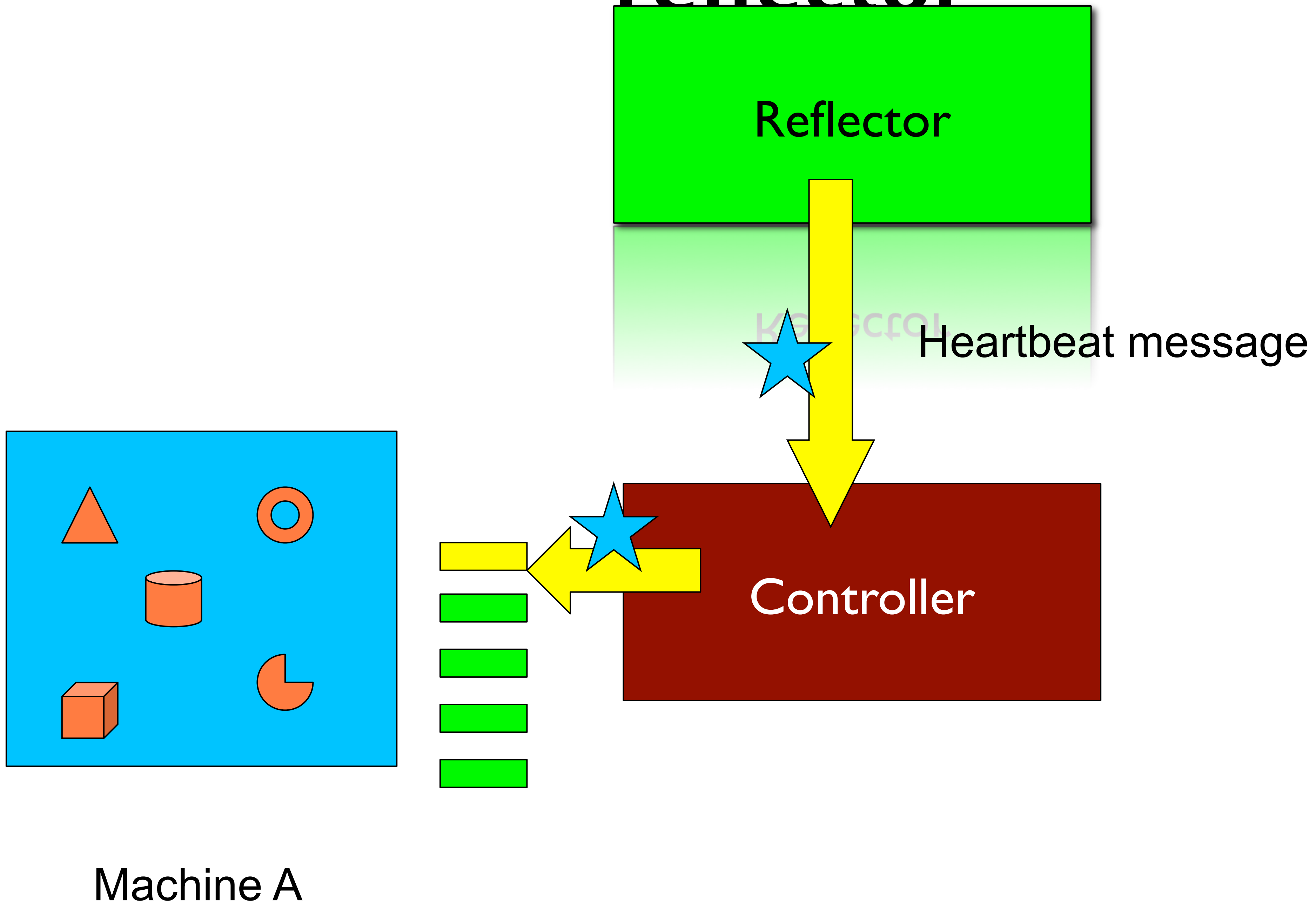
If there is no external message to move things forward, the reflector will manufacture one.



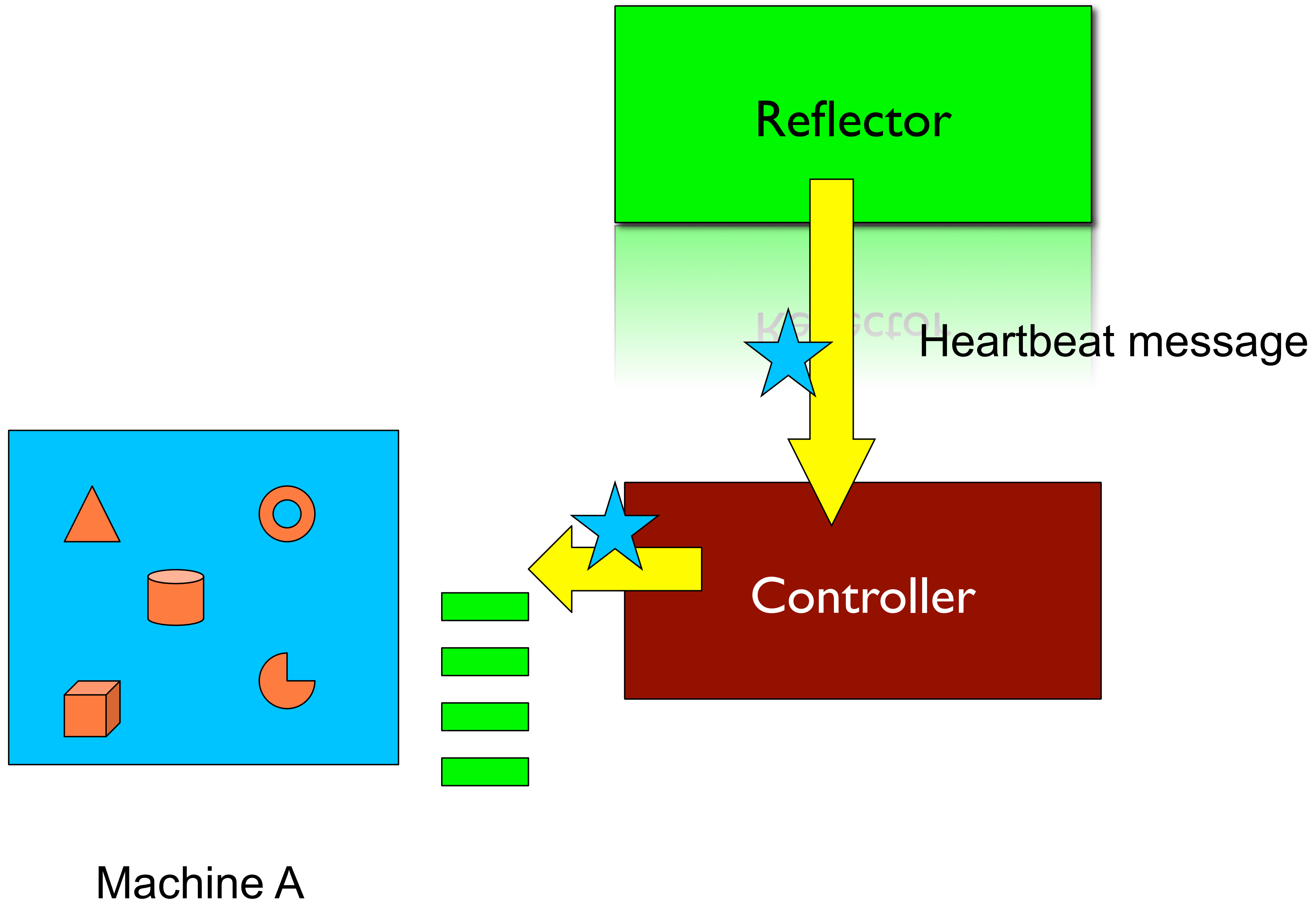
Messages are then executed by bobble up to and including the heartbeat message from Reflector



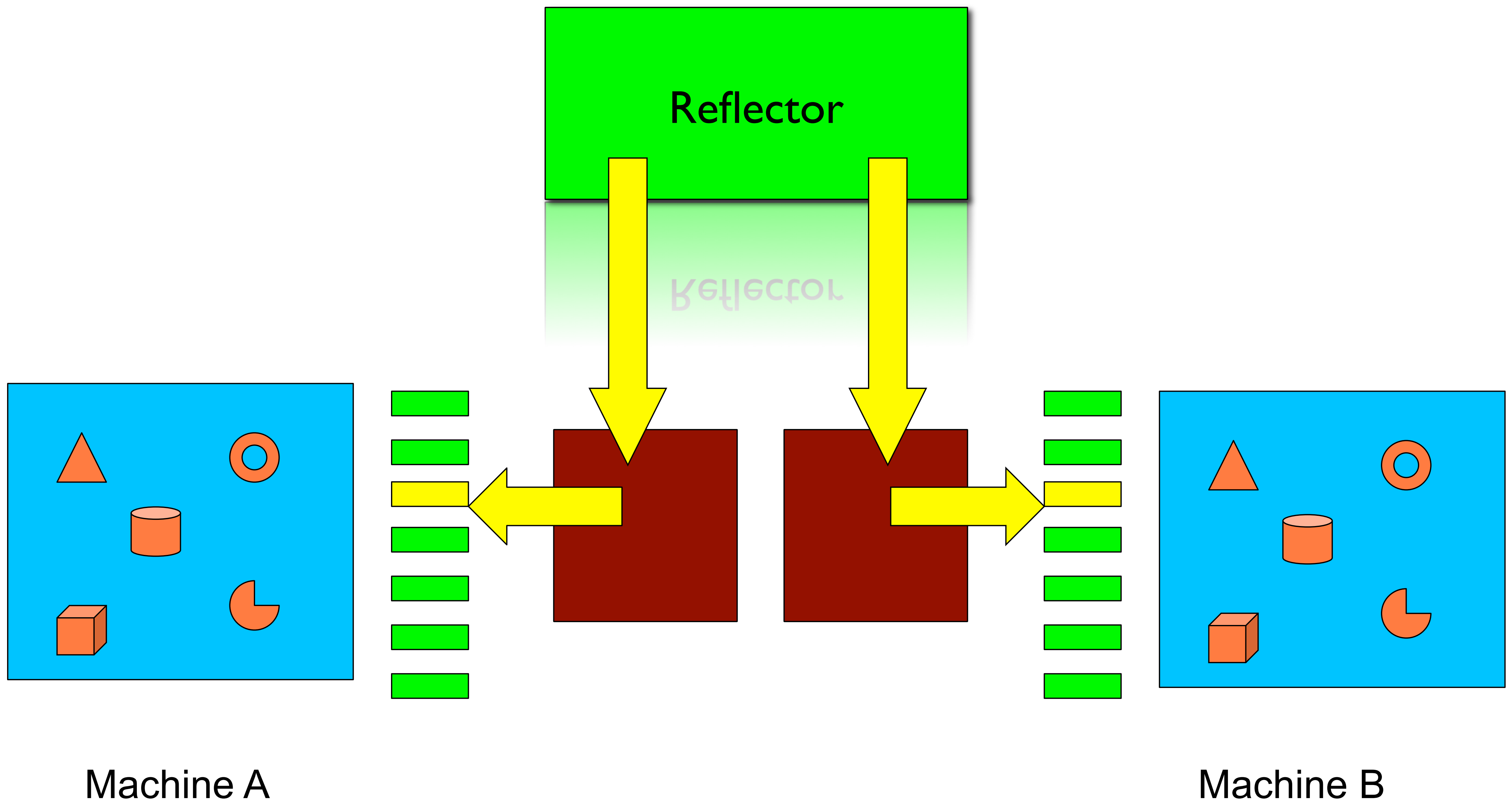
Messages are then executed by bobbles up to and including the heartbeat message from reflector



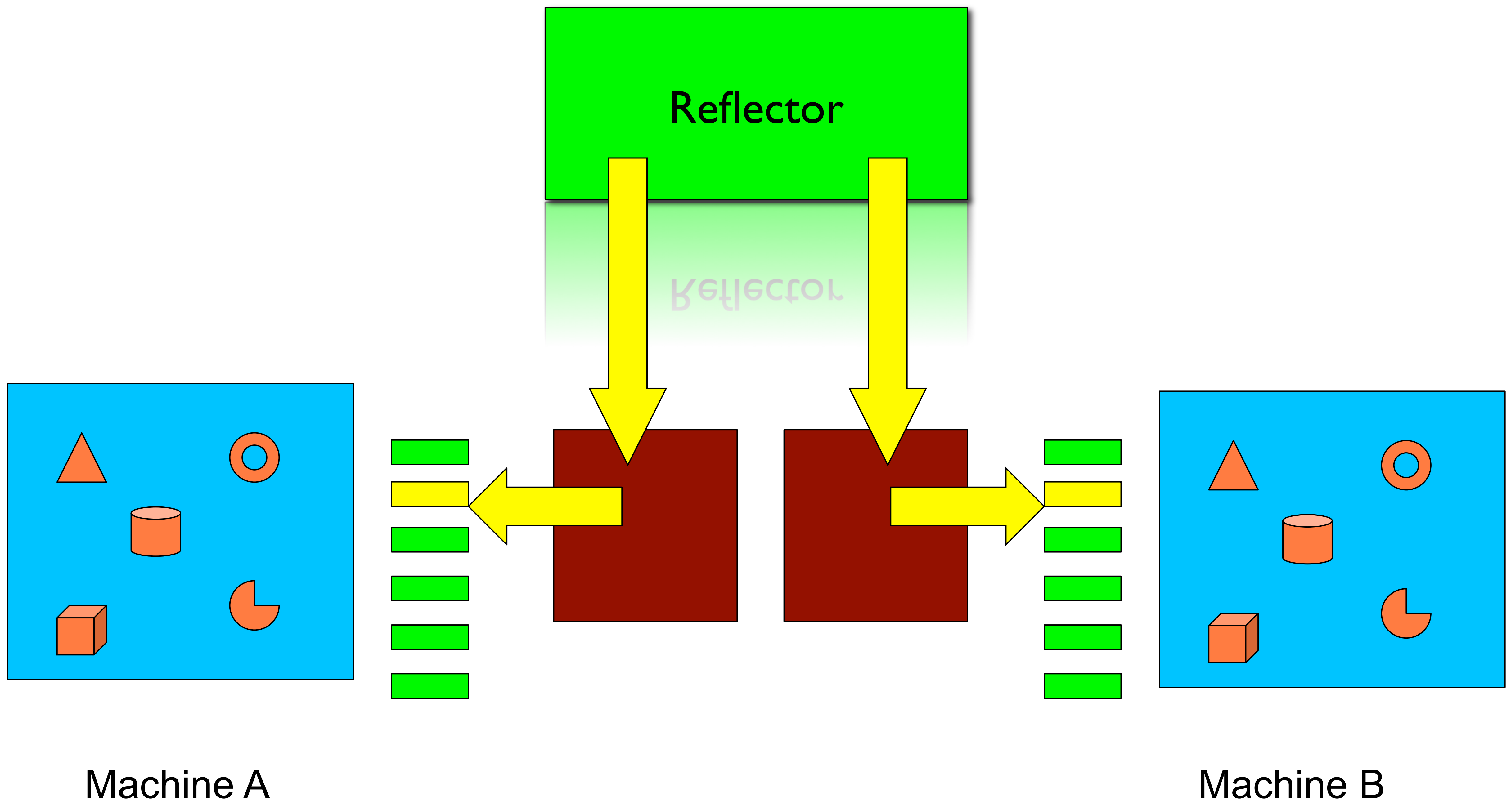
Messages are then executed by bobble up to and including the heartbeat message from Reflector



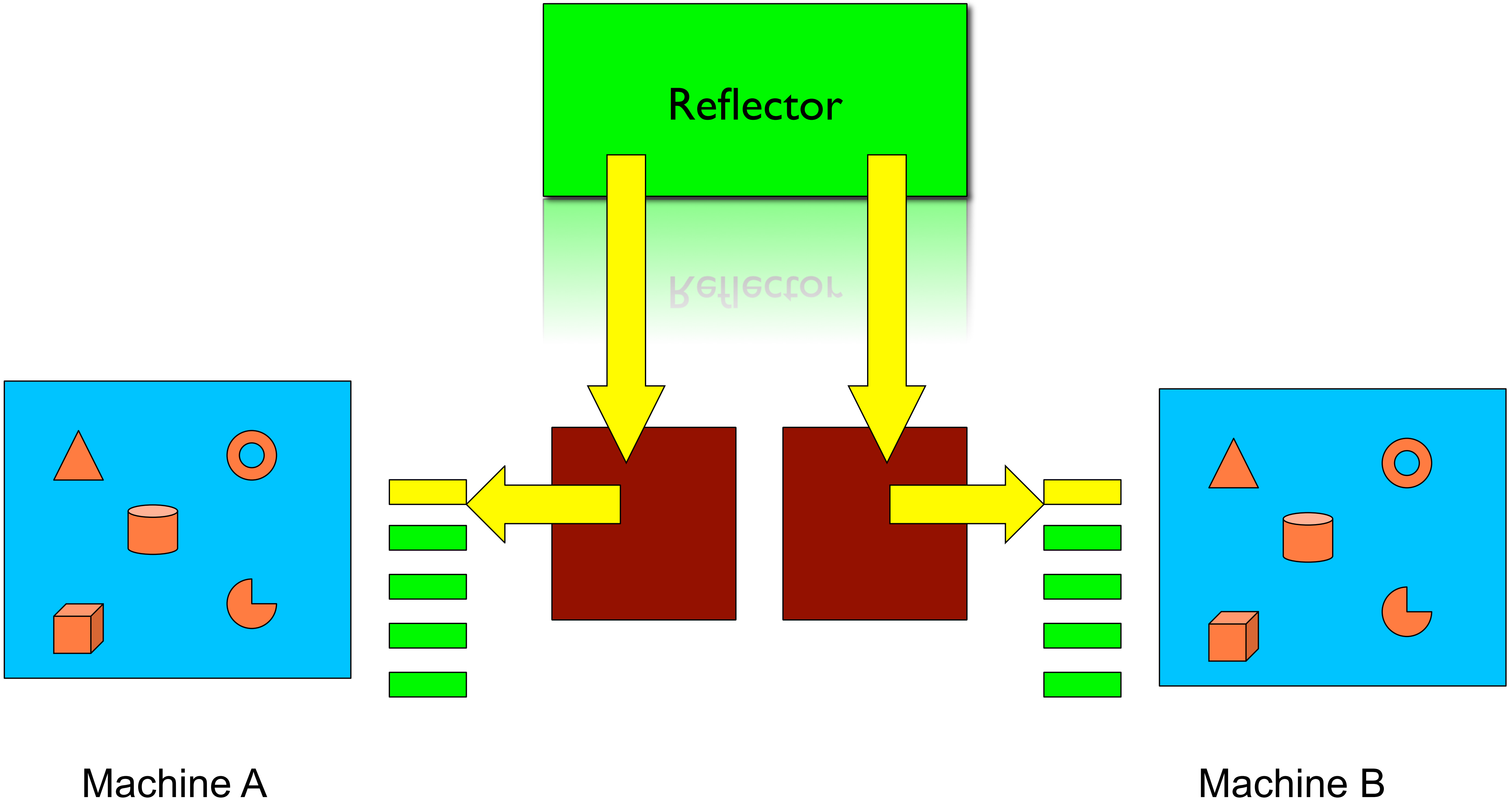
This works for any number of replicated bobbles.



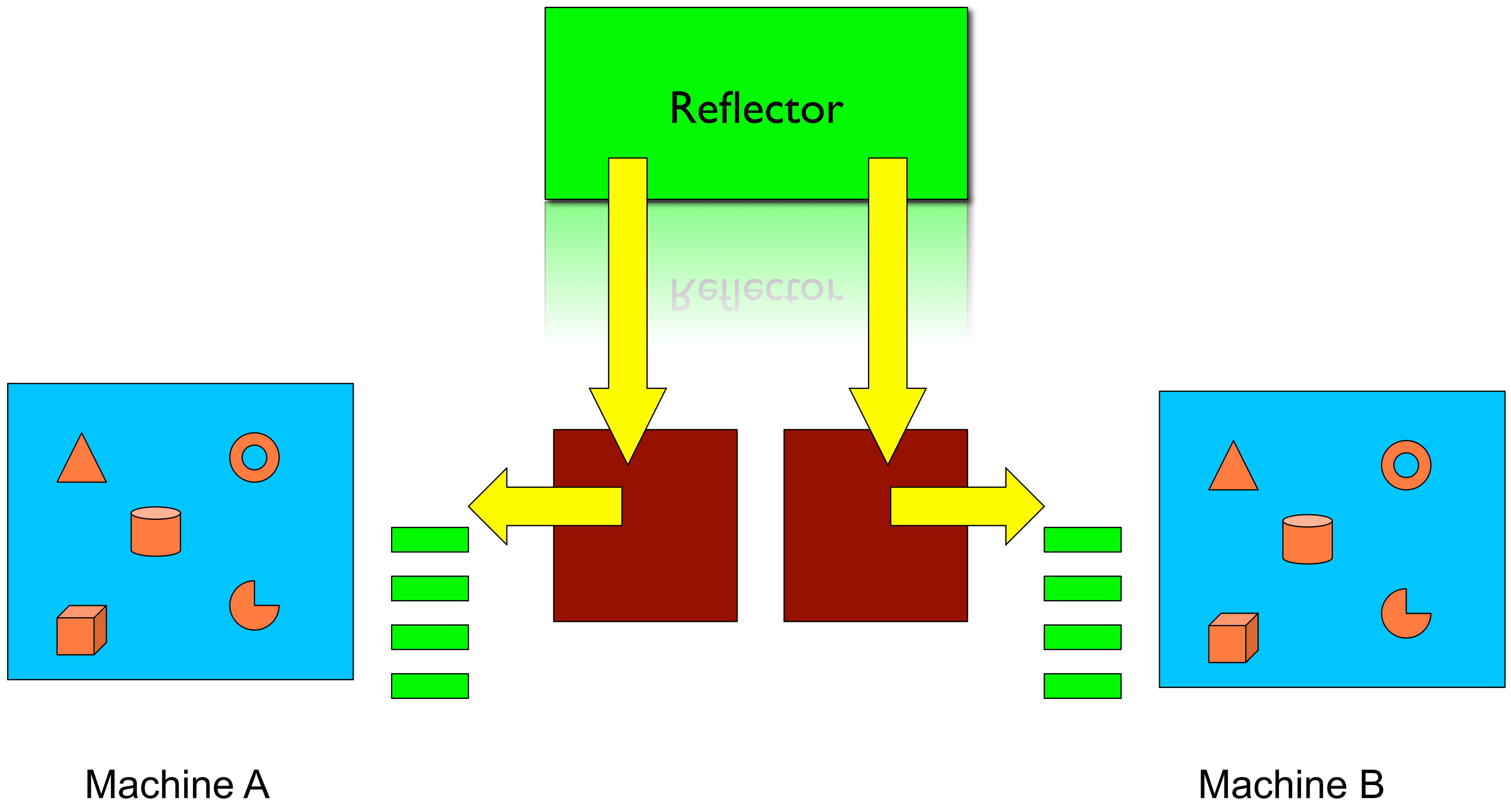
This works for any number of replicated bobbles.



This works for any number of replicated bobbles.



This works for any number of replicated bobbles.



Reflector/Controller/Bobbles

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

Reflector enumerates messages

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

Bobbles view of time

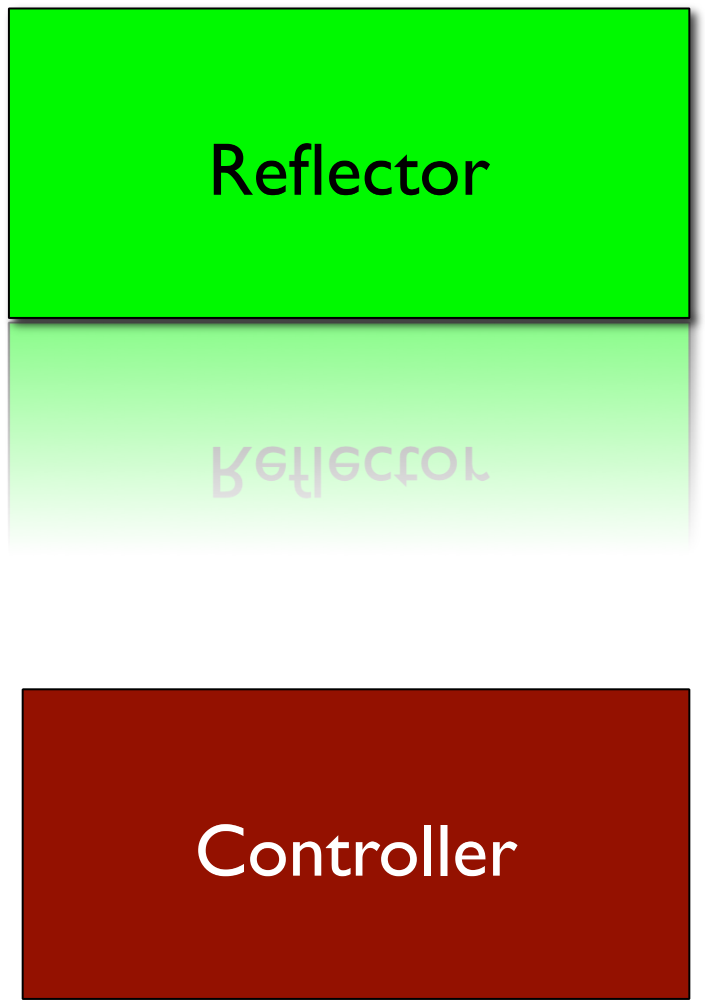
- Bobbles only understand time in quantized terms – there is only now (when message is executed)
- – and now + x (when future message will be executed)
- Reflector controls execution time for all bobbles.
- Reflector needs to send heartbeat messages to ensure smooth animations.
- Heartbeat messages can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

Starting and Joining

First there was the server/reflector...

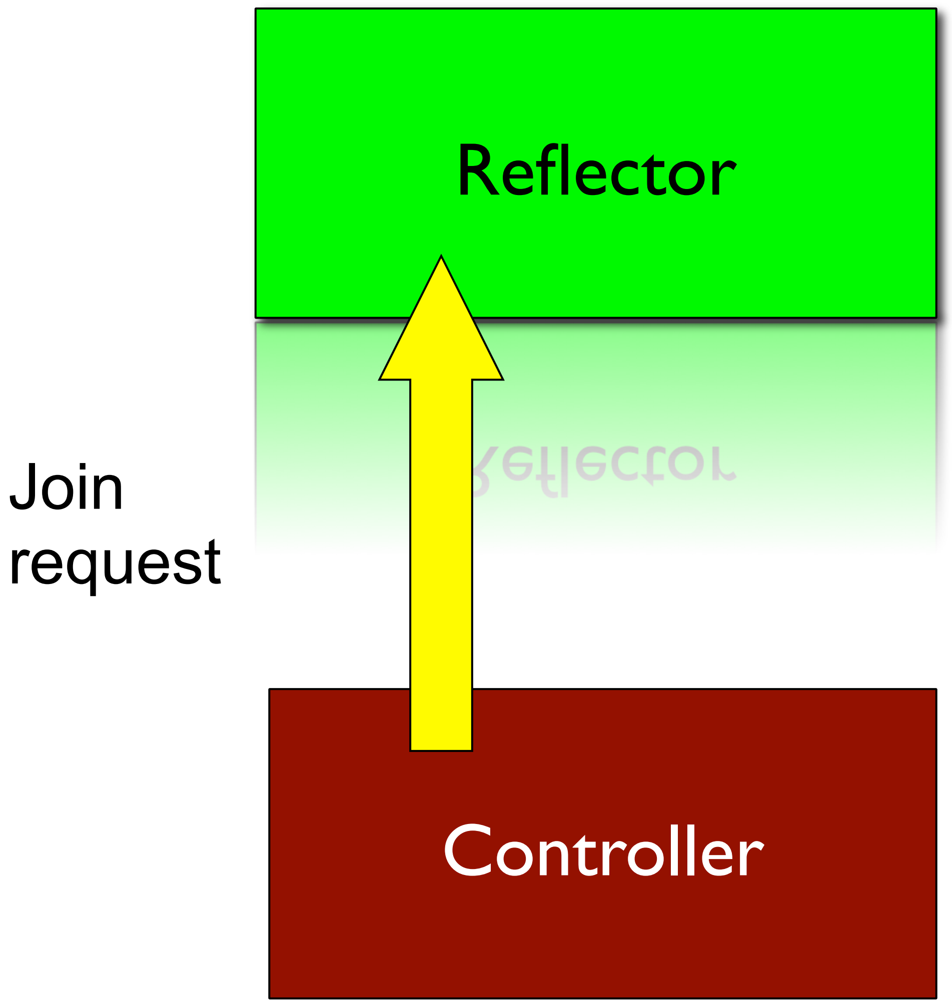
The new reflector can be on any machine, not just the users.

User creates a Controller



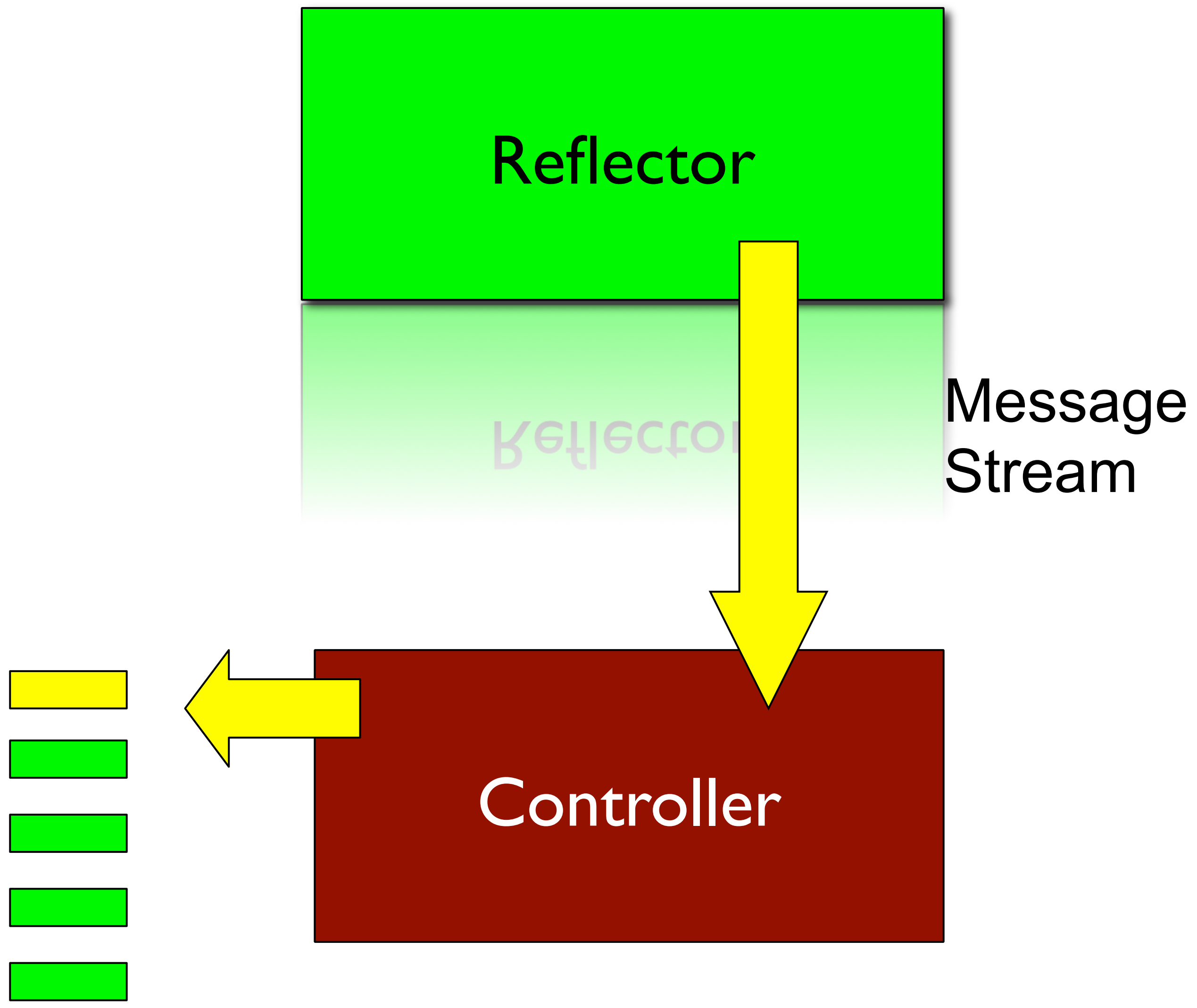
The new Controller will be on the users machine. It is given the Reflector address and port number. Since it will be used to construct the initial bobble, we call it the master controller.

Request to join



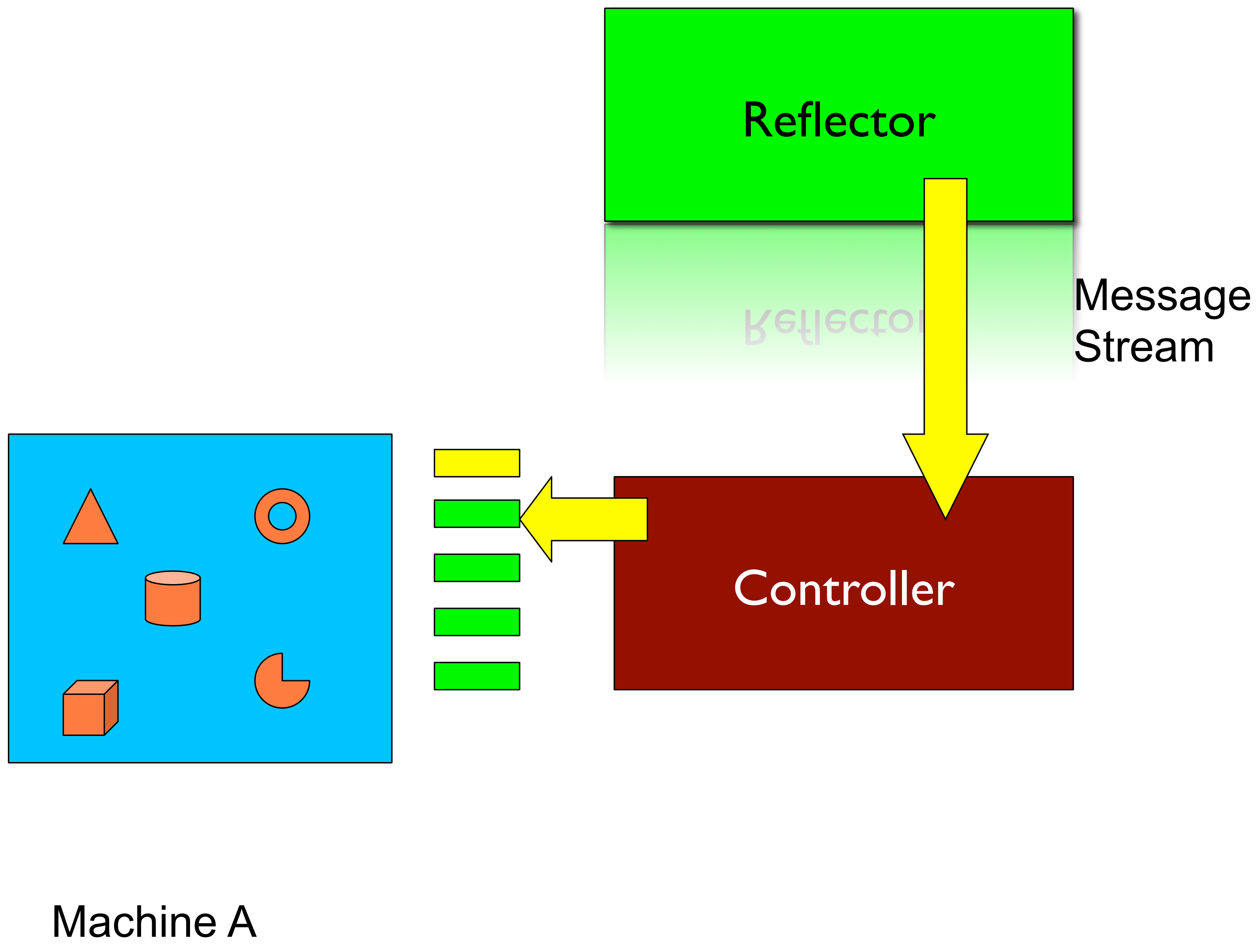
The controller sends a message to the Reflector asking for messages. The Reflector (if it is authorized) begins publishing its message stream to the controller.

Controller joins Reflector message stream

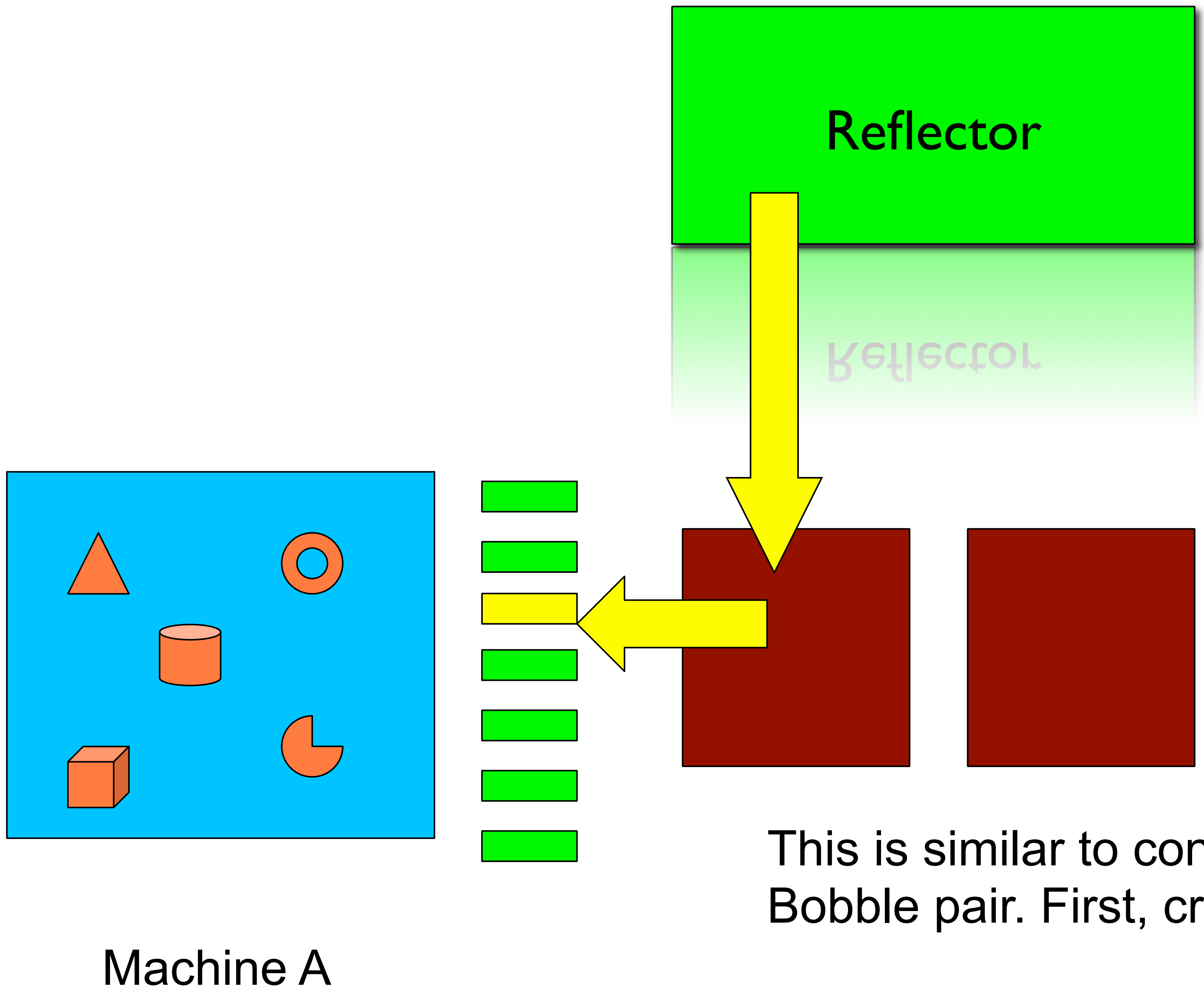


Once the join is accepted, the Reflector sends all replicated messages and heartbeats to the controller. The controller saves these into a message queue.

Controller constructs new Bobble

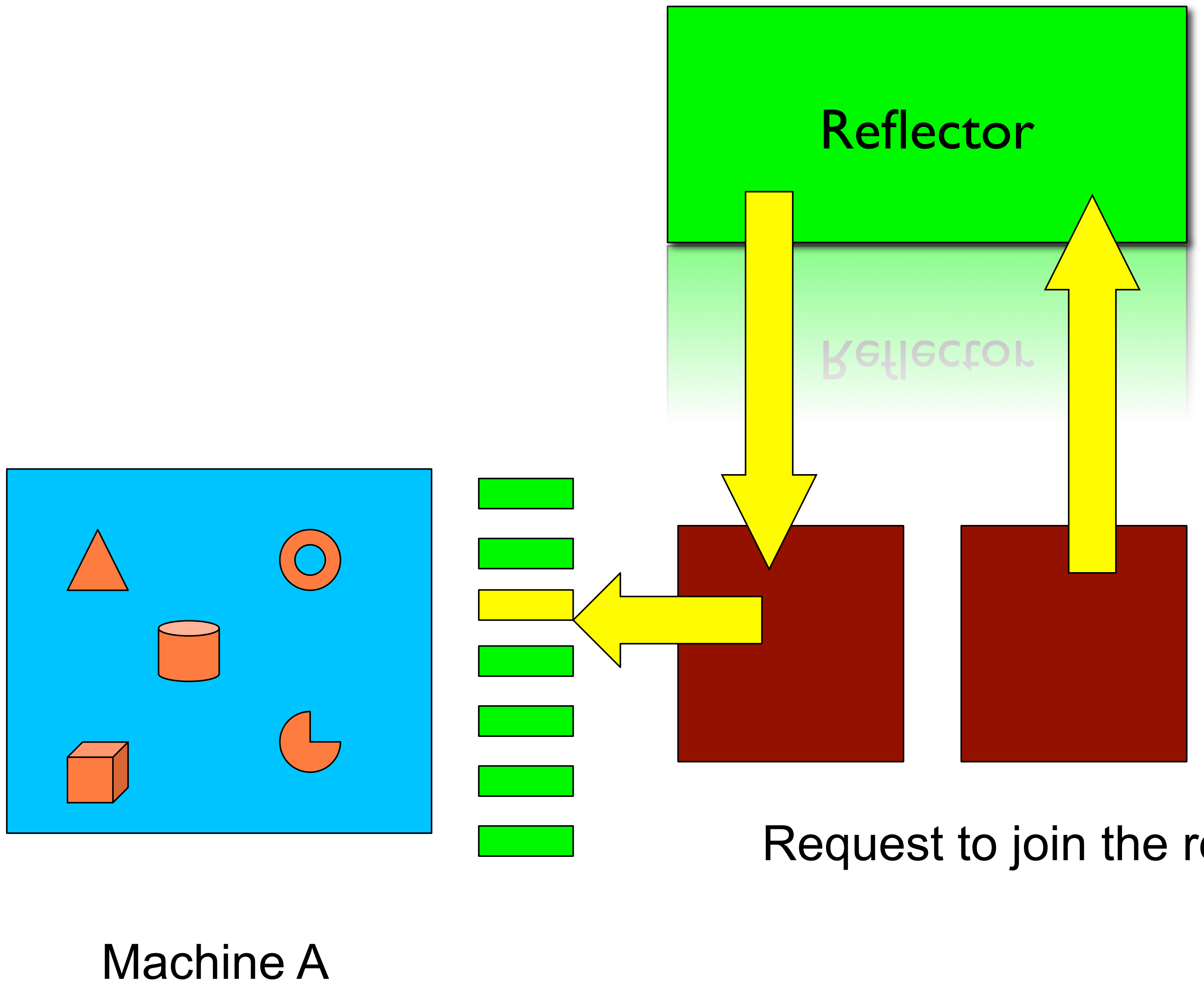


Adding a new user – construct controller



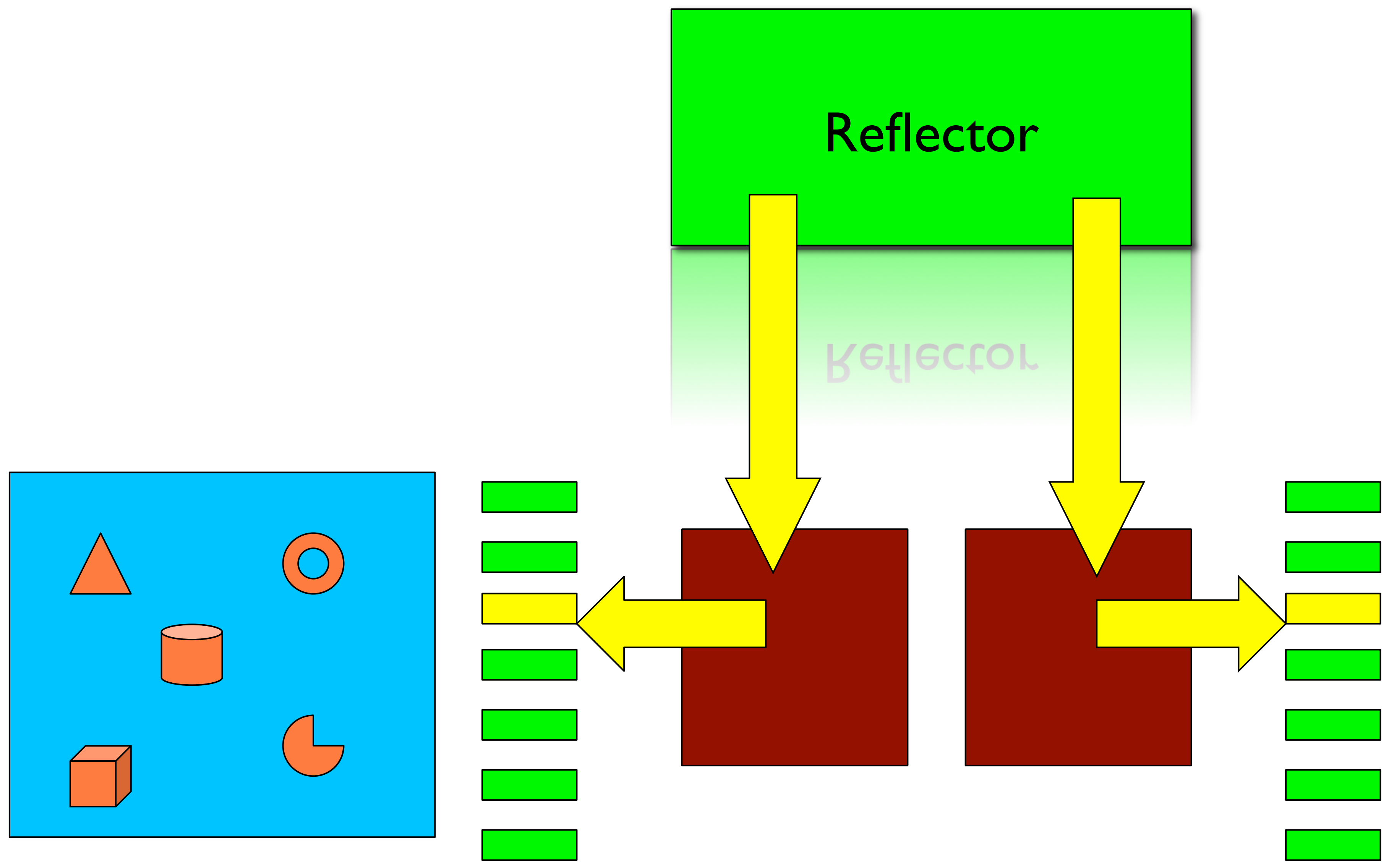
This is similar to constructing the initial controller/ Bobble pair. First, create the controller.

Request to join Reflector



Request to join the reflector.

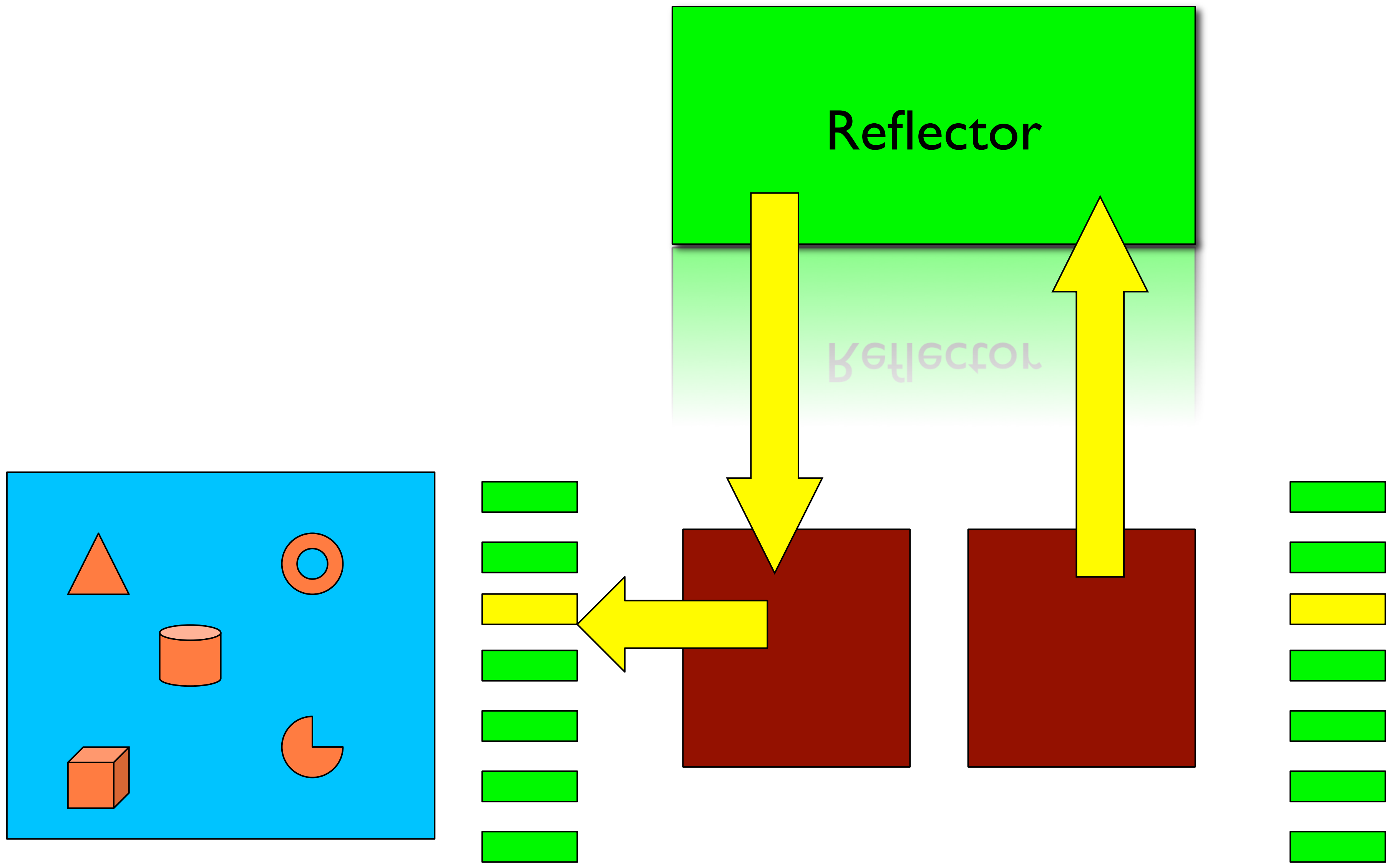
Start receiving messages



Machine A

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

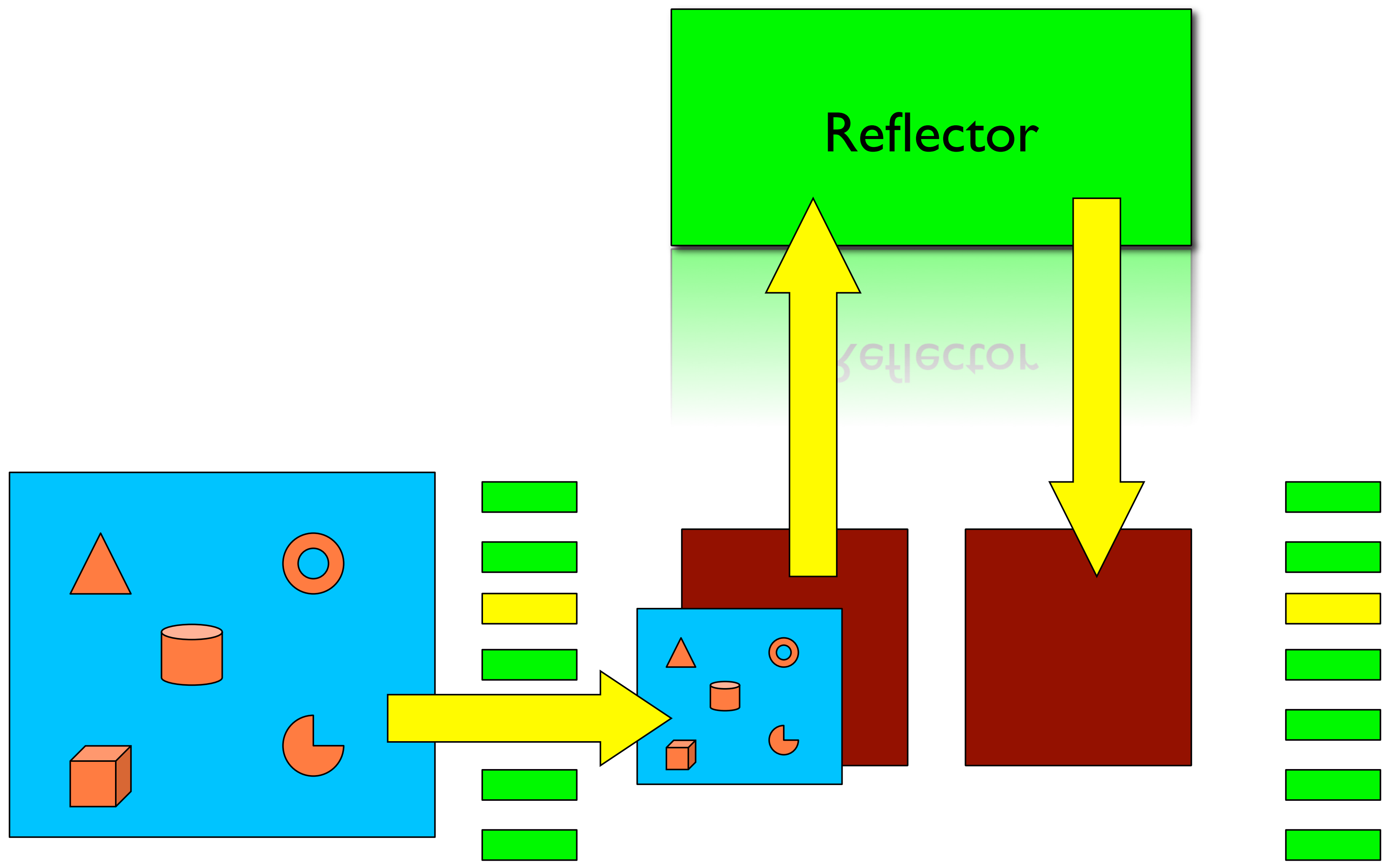
Request Bobble



Machine A

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

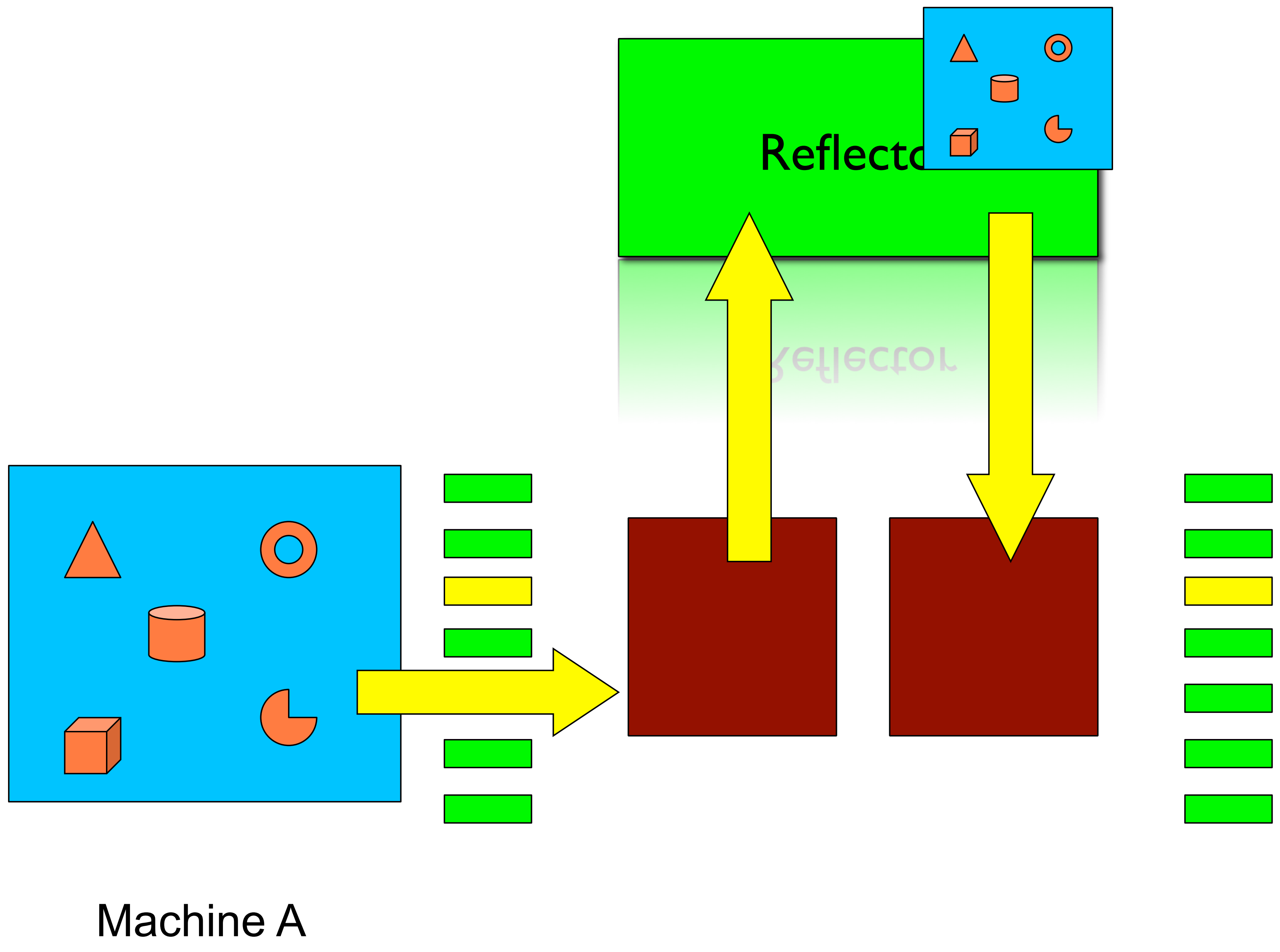
Bobble checkpointed and sent



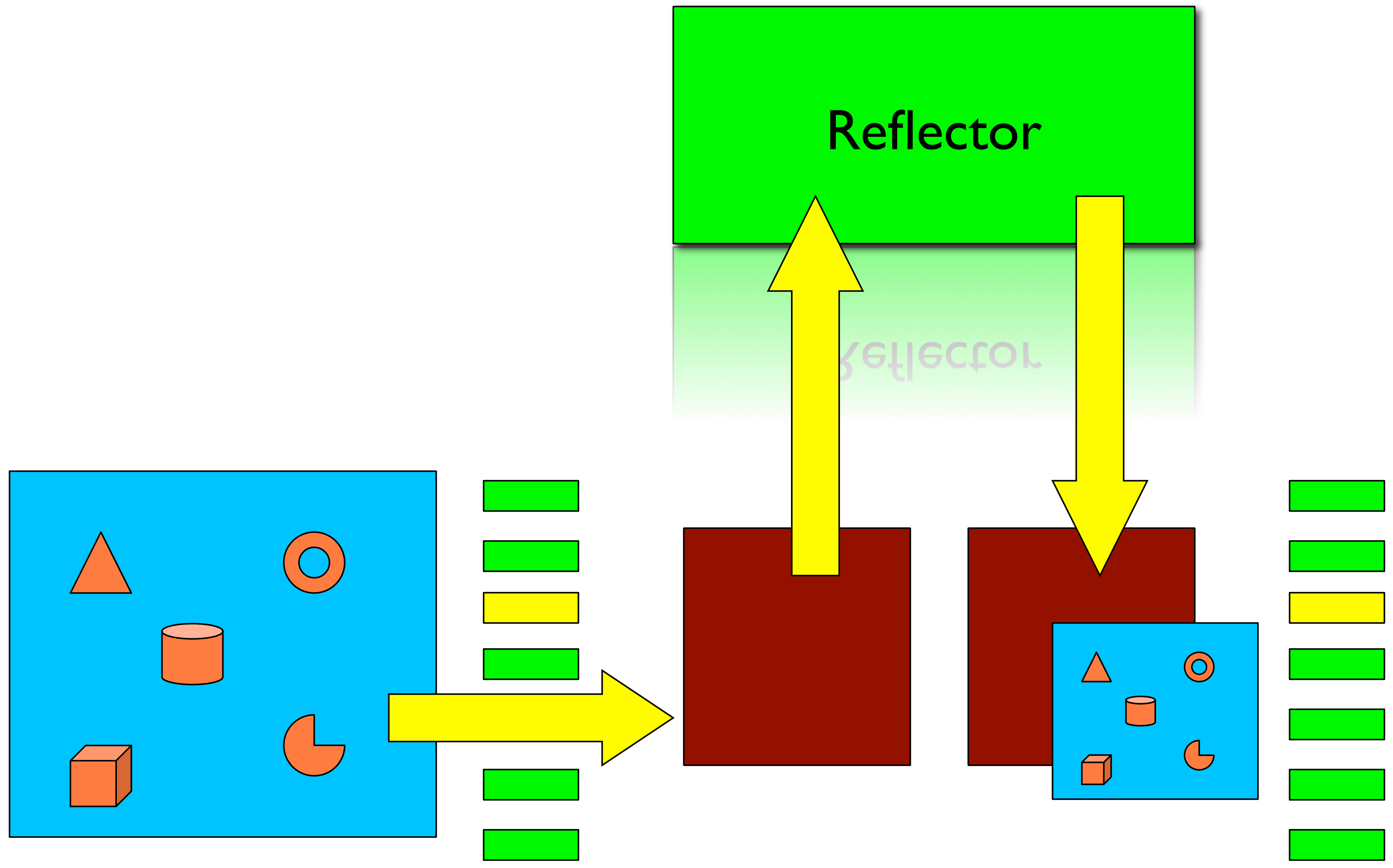
Machine A

The bobble is a checkpoint streamed to the new controller via the reflector.

Bobble checkpointed and sent



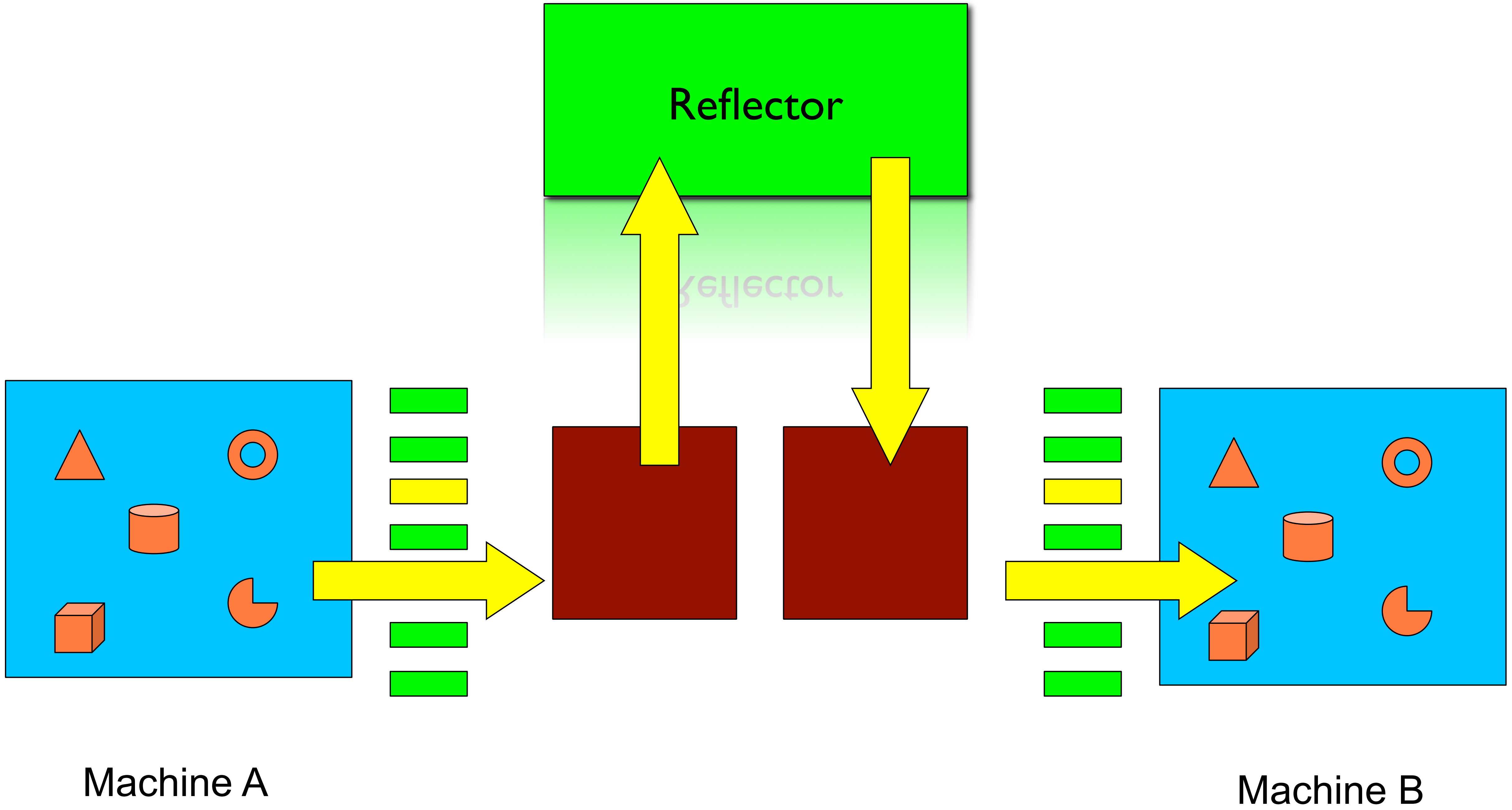
Bobble checkpointed and sent



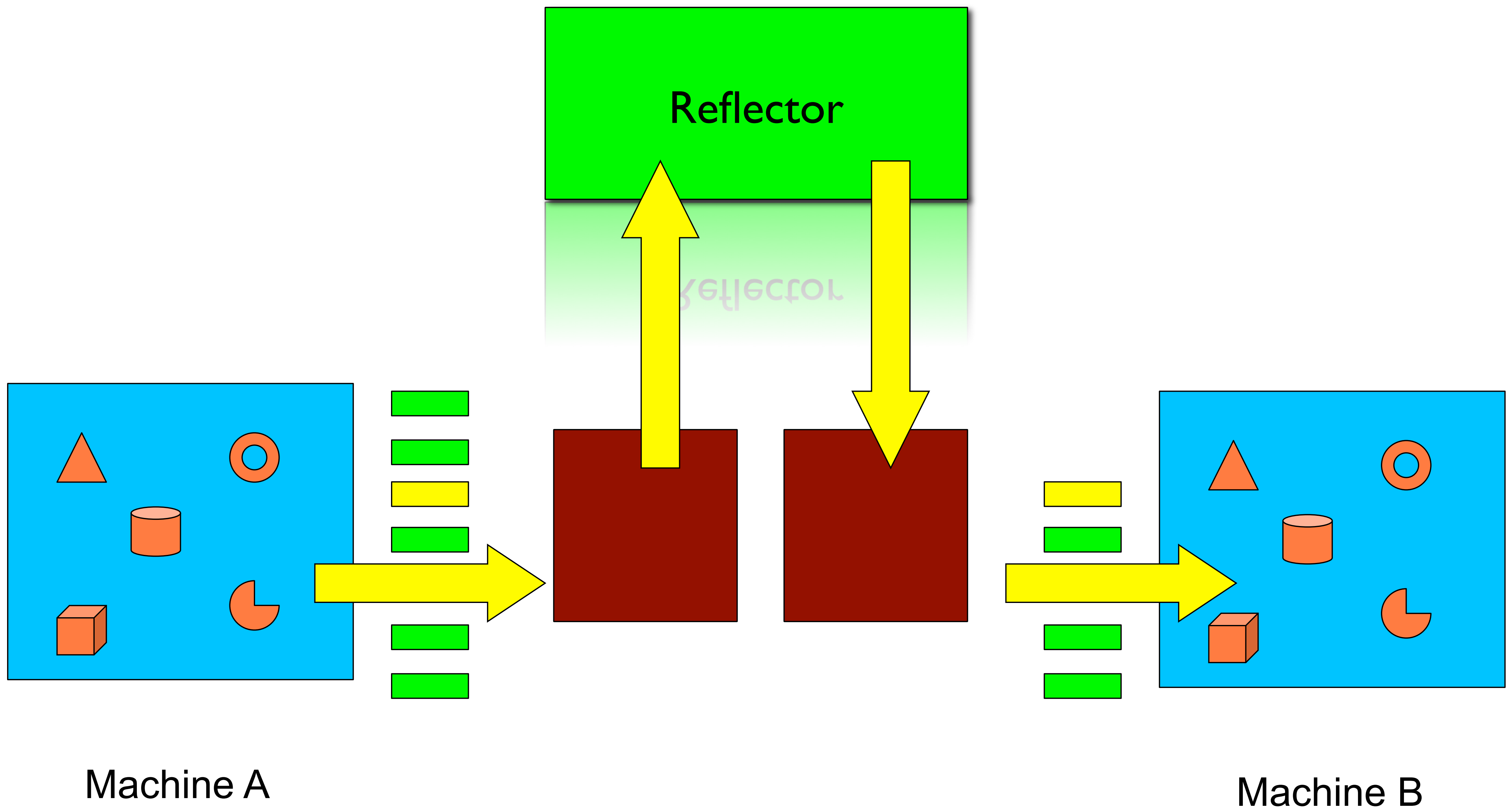
Machine A

The controller resurrects the bobble locally.

Bobble is resurrected and can now be displayed.



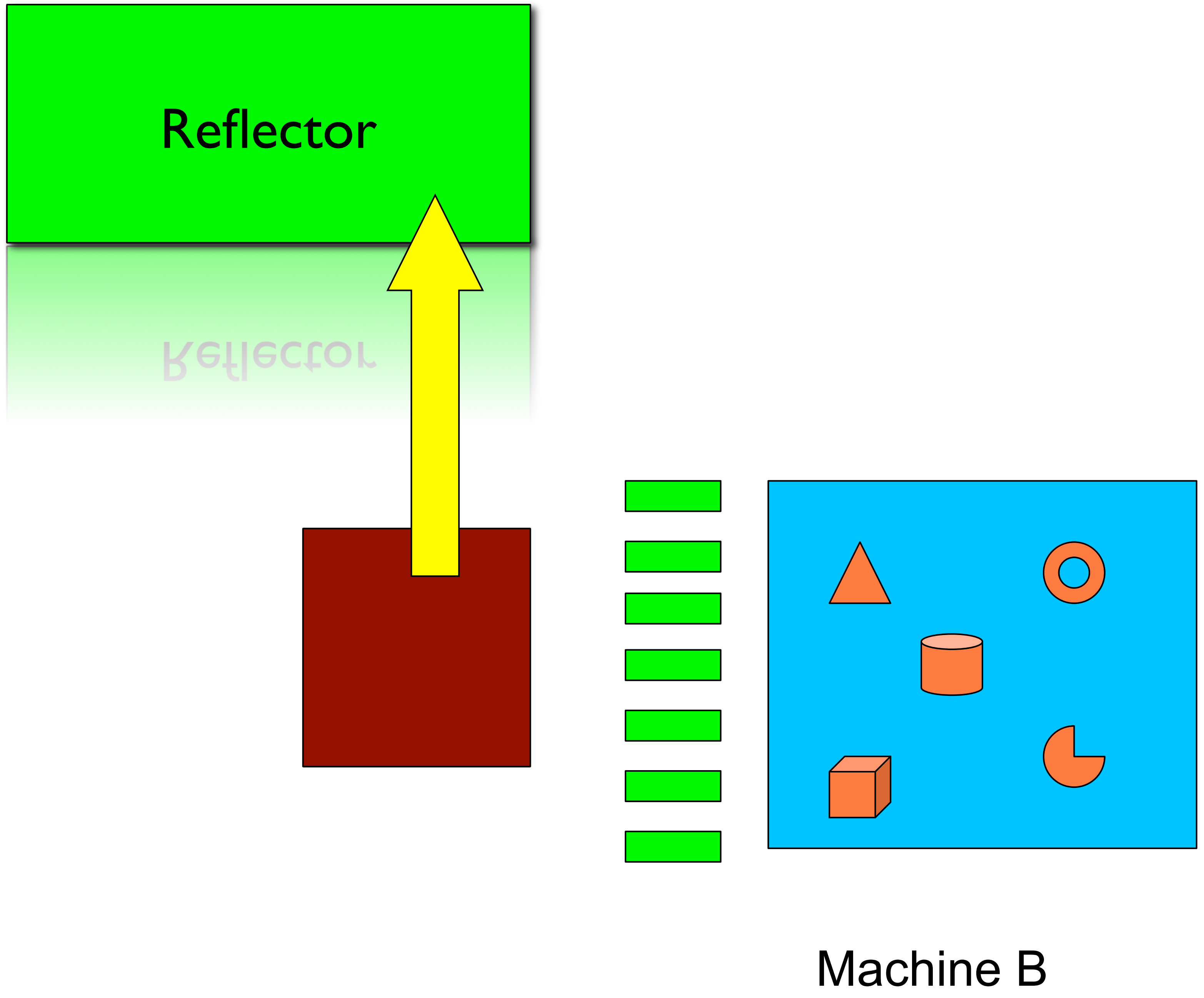
Message queue is culled to >= Bobble current time.



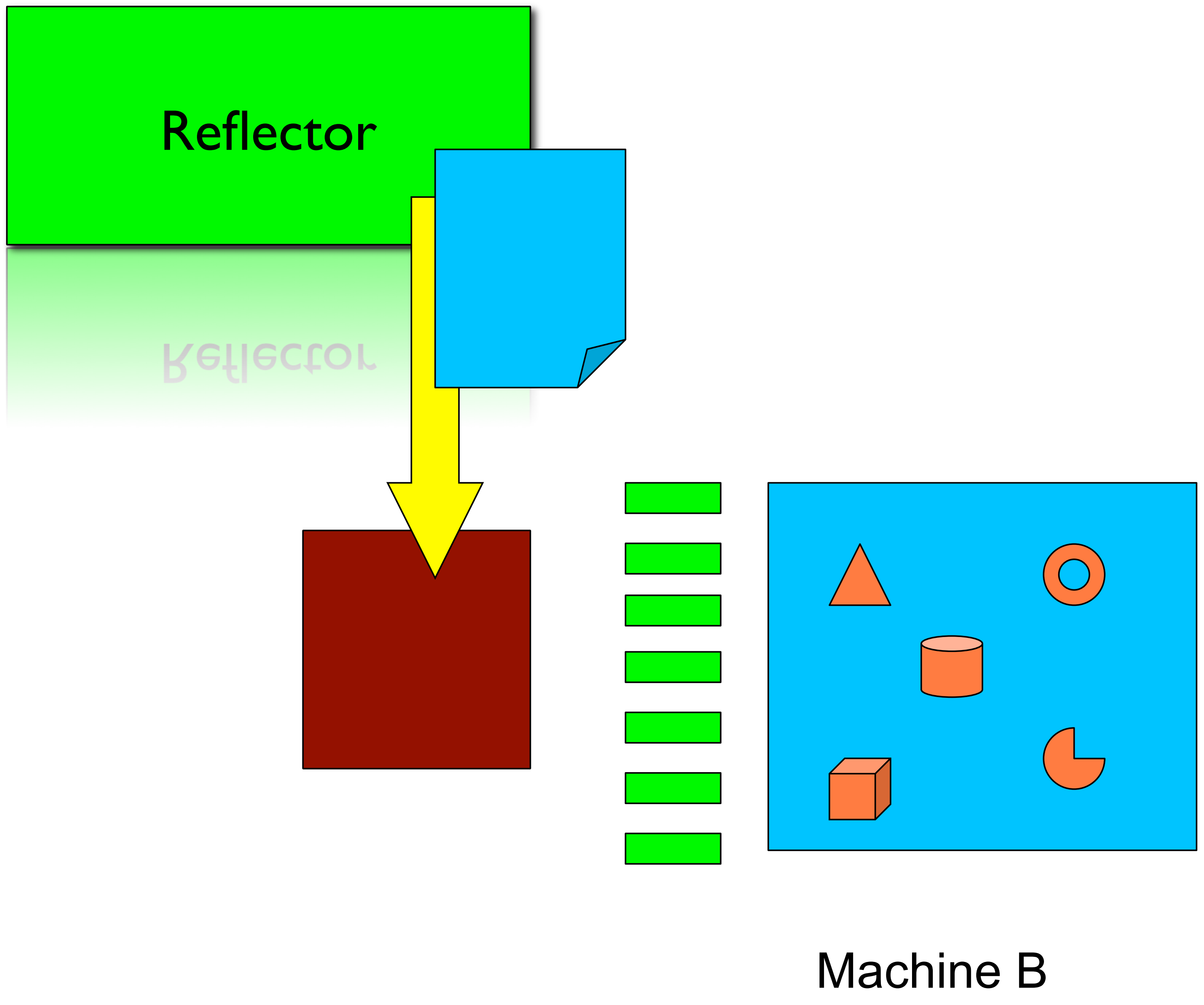
Participating

- Joining is “view only” interaction – the user can not modify the Bobble contents until he gets permission to participate.
- The user must request permission from the Reflector to participate.
- The Reflector 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.

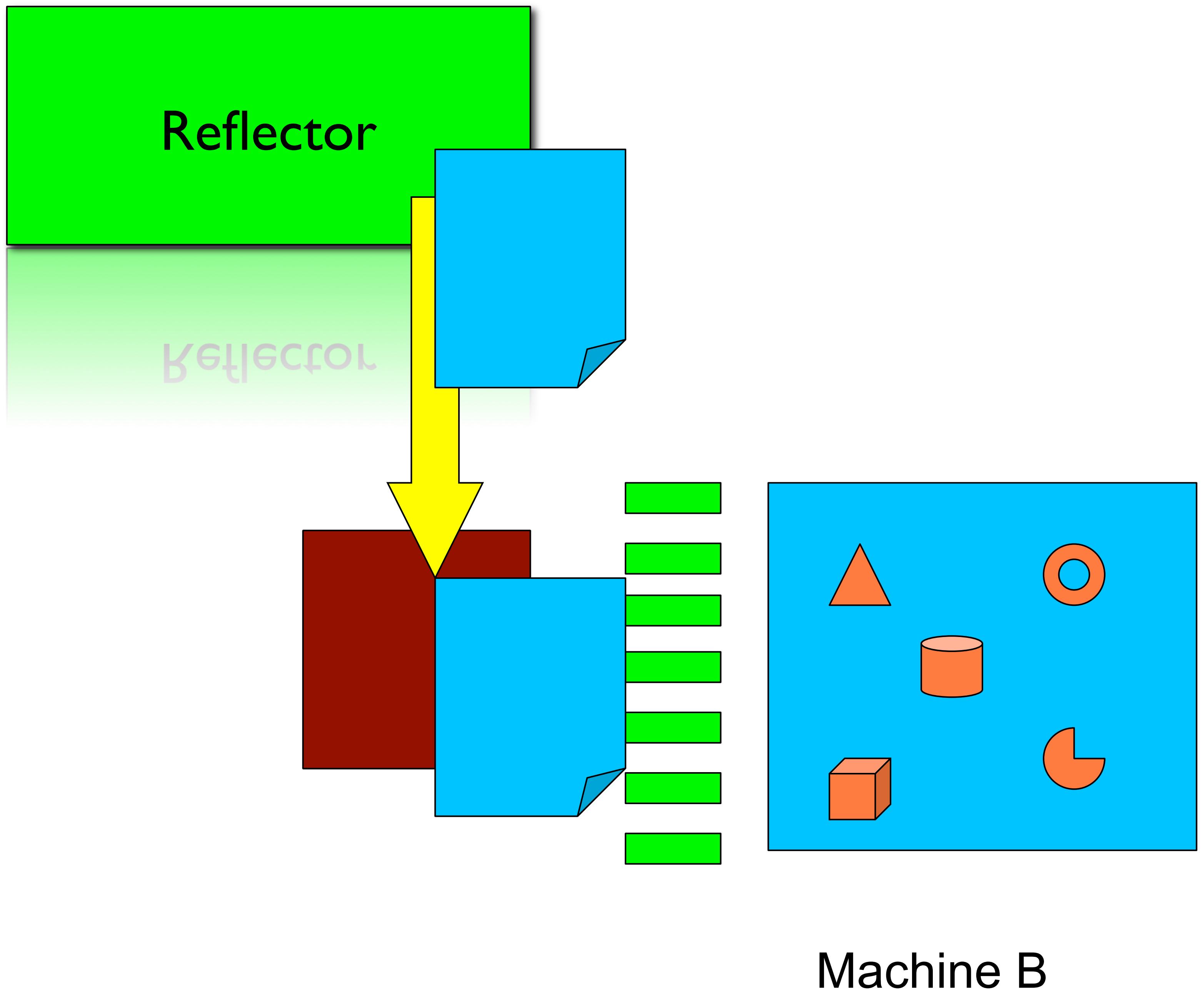
Request right to participate from Reflector



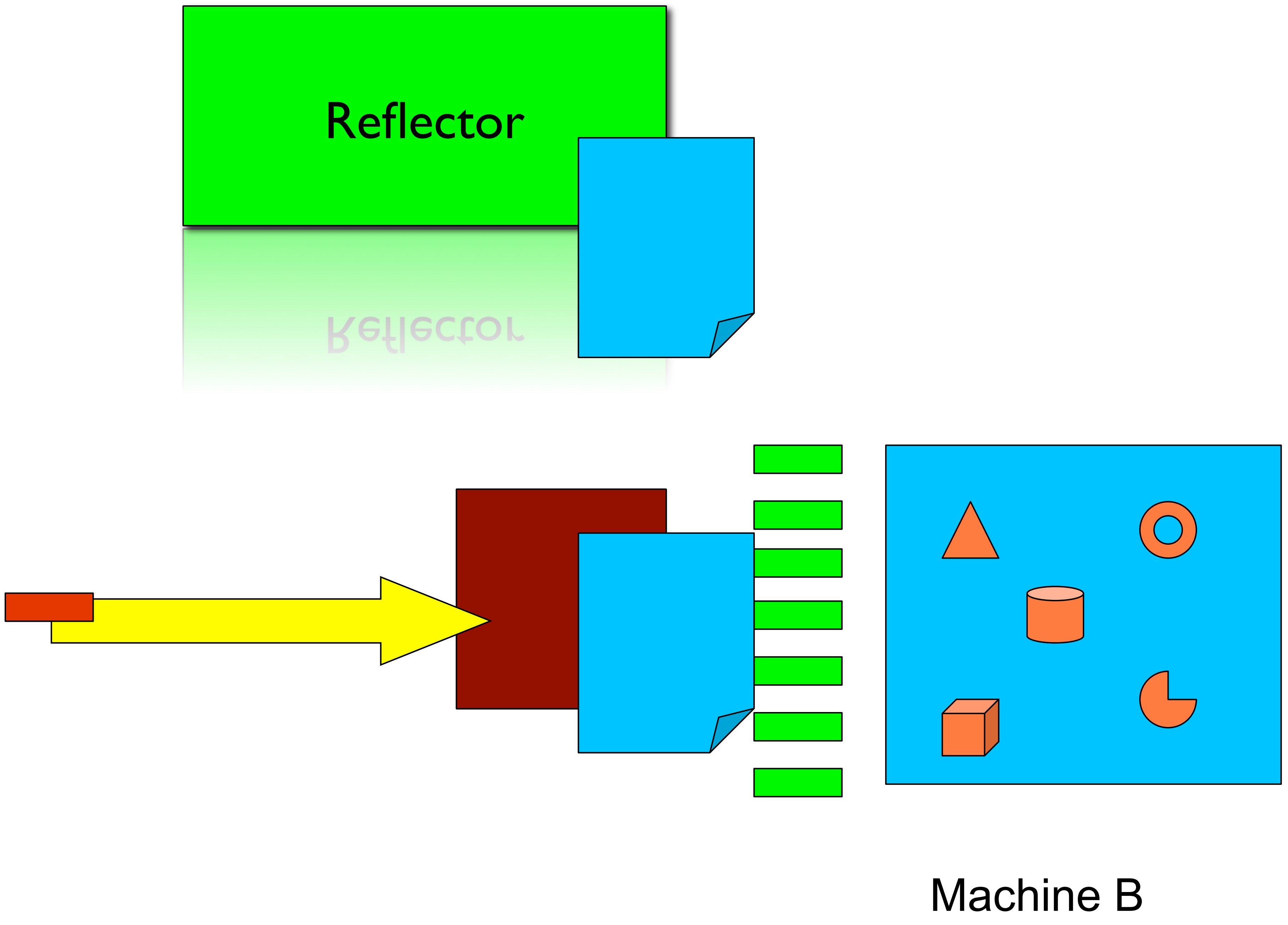
Reflector passes a list of facets, or interfaces to controller.



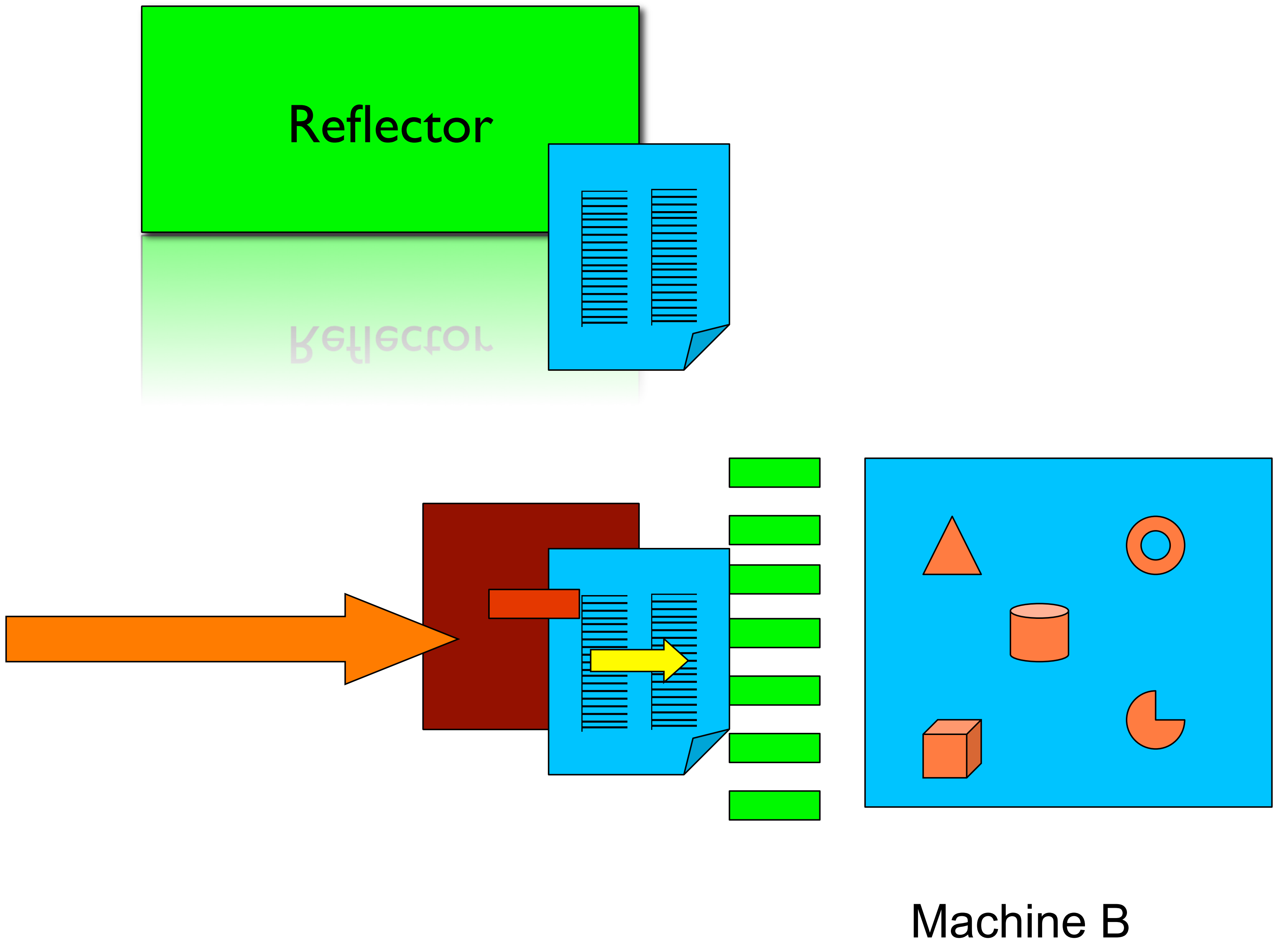
Reflector passes a list of facets, or interfaces to controller.



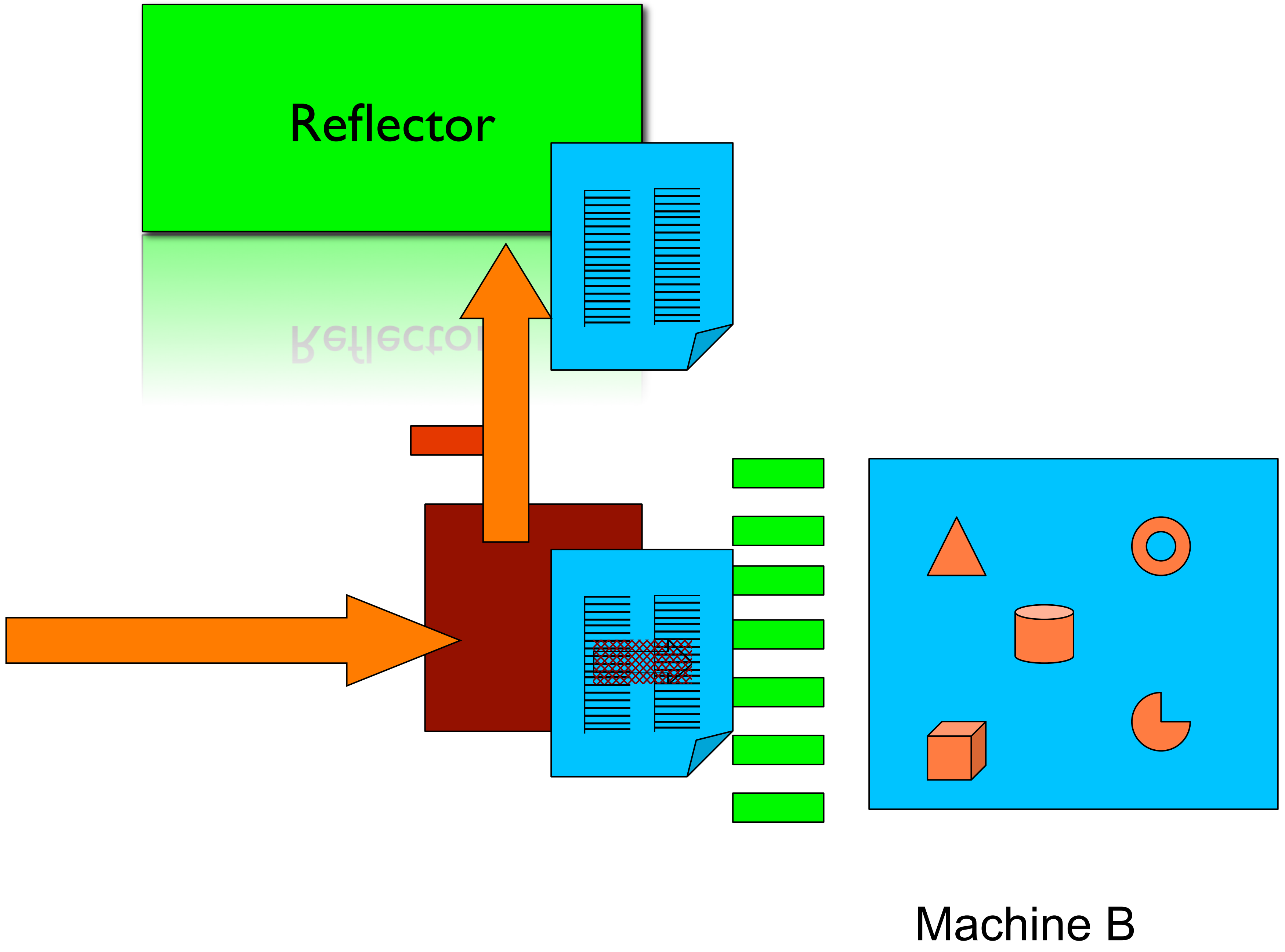
External message is sent to controller.



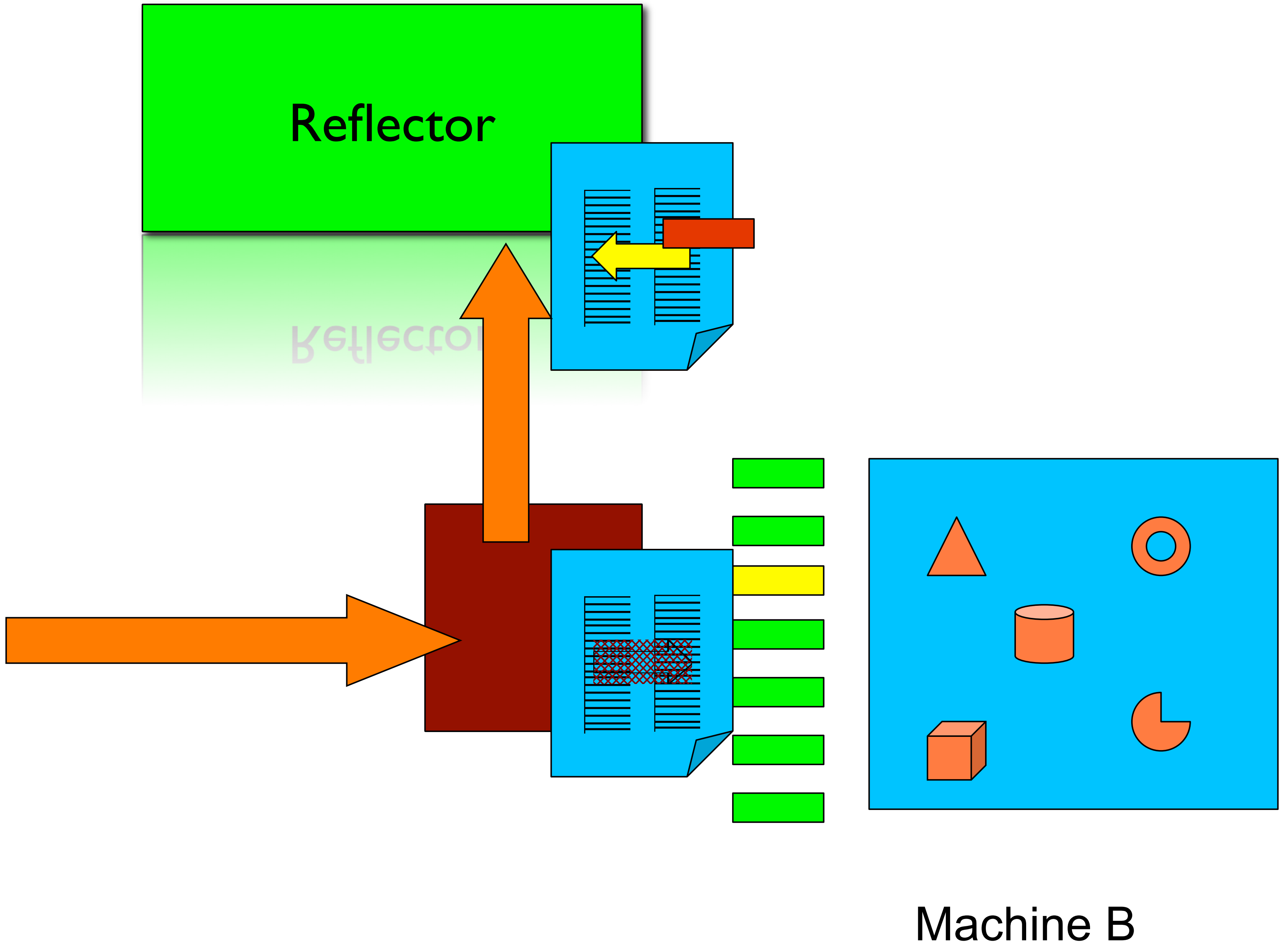
Controller looks up object/message pair in facet dictionary.



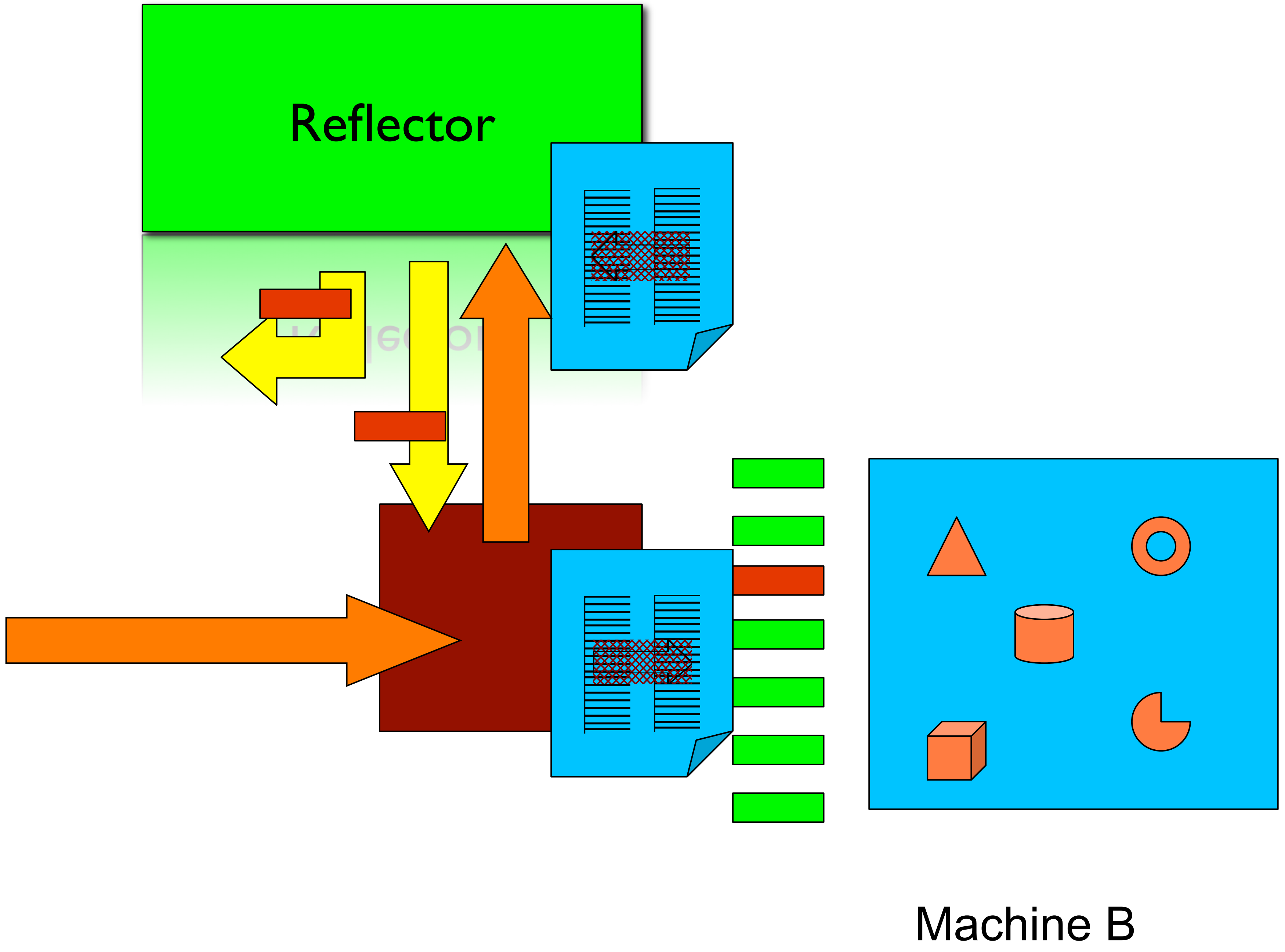
Facet is used to invoke replicated message, sent to reflector



Reflector performs reverse look-up to find original message.



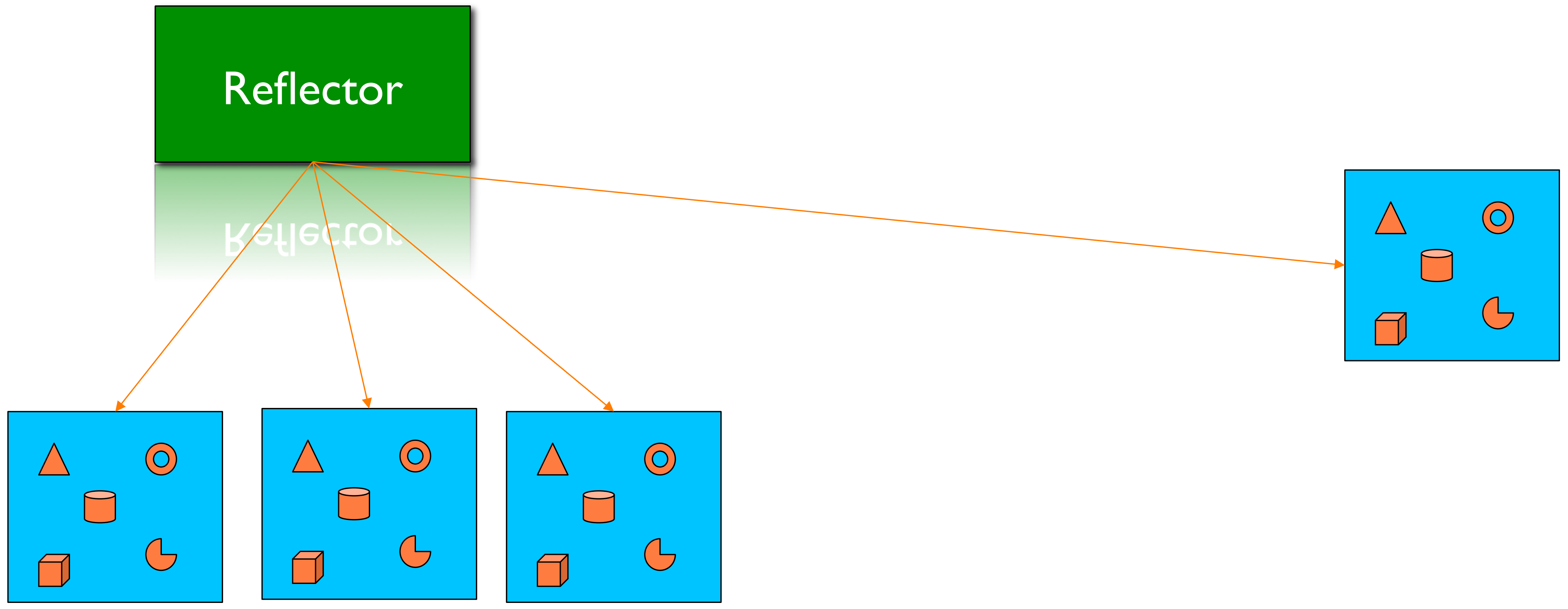
Actual message is sent to all controllers.



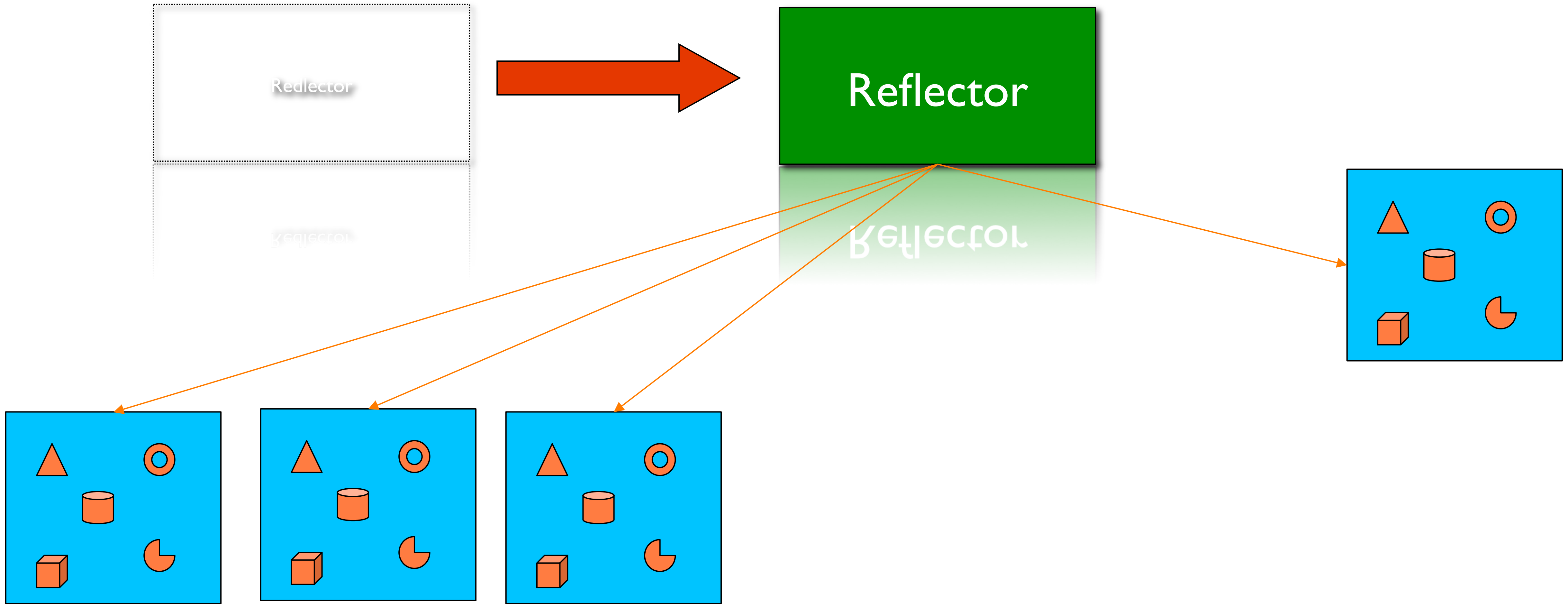
Nice side effects

- 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 a bobble (though the high latency participant has a poor experience).
- Reflectors can be independent of Bobble/controller pairs, hence can be positioned on minimal latency or centralized balanced latency servers.
- Reflectors can even be moved around if necessary to improve latency for specific users or groups.

Latency does not effect accuracy – only usability.



Reflector can be moved to more latency centric location.



End