Emergent Software Systems

Summer School

Barry Porter & Roberto Rodrigues Filho
School of Computing and Communications
Lancaster University

Funded by The Royal Society Newton Fund

Distributed Emergent Systems

- Applying our concept to complex distributed systems
- Starting from an **objective** and **reward**, how do we assemble and learn the best collection of behaviours?
- How do we convert all of the decision making in distributed systems into a simple action / reward model?
  - While guaranteeing that the system will always be "functional"

Distributed Emergent Systems

- We project our local component model directly into a distributed system, and we introduce the idea of "Class A" and "Class B" distributed interactions

- Given a set of available machines M, this allows us to decide on **placement** of all needed sub-systems; the **replication** factor of those systems; and the internal **composition** of each sub-system
  - *All guided from one reward function*

Self-Distributing Systems

Class A

Self-Distributing Systems

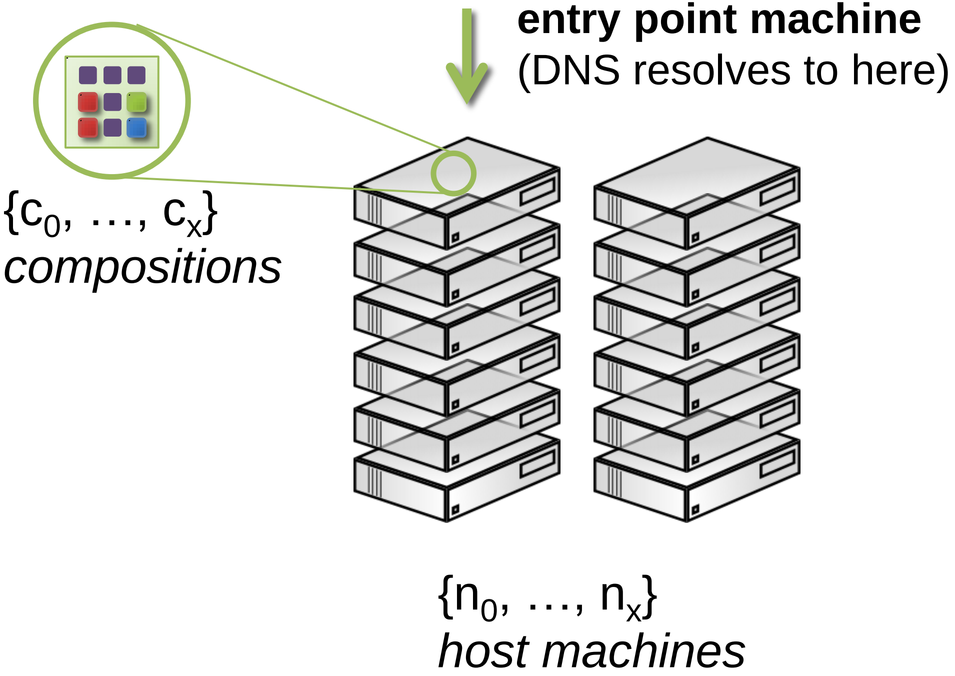
- Let’s imagine that we’re building an entire datacentre ecosystem
- We have the necessary building blocks for a web server, database, and memcached system
- We’ll start by considering only the web server system

Self-Distributing Systems

- We have a set of available machines which can host components
- We have an entry point machine to which ingress traffic (HTTP requests from users) arrives to the datacentre
- We have a set of compositions which can form the functionality of a web server (receive HTTP request, respond with resource)

Self-Distributing Systems

- To illustrate:



Self-Distributing Systems

- Dana gives us two useful capabilities here
- We can hot-swap a component at runtime to a different implementation
- We can see which interfaces have state (declared as transfer state)

Self-Distributing Systems

- This leads to the idea that we could *relocate a component* to a different host machine at runtime

- To do this we would create a proxy version of the interface which forwards function calls to a remote version, marshalling / unmarshalling parameters / return values

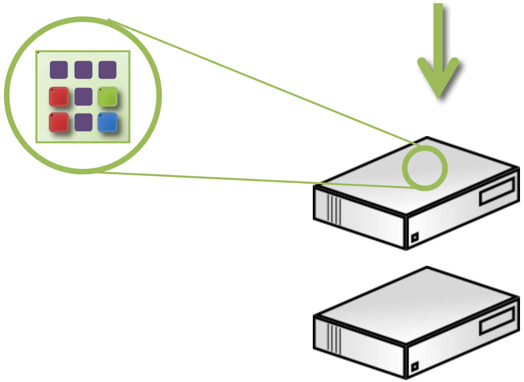
Self-Distributing Systems

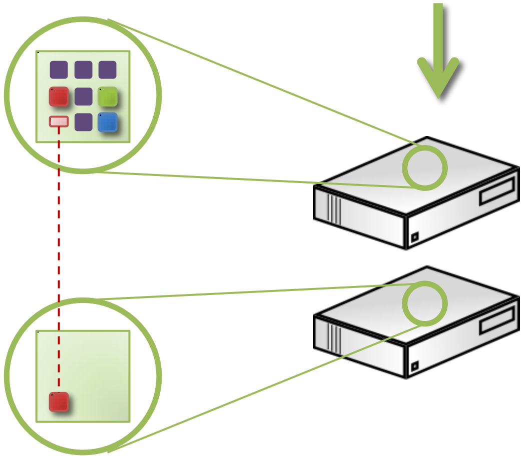
- By relocating a component we might move it closer to the source of data it uses, or we might benefit from extra computation power of another host machine

- But we can do more: if an interface has no associated state, we can *replicate a component* across multiple hosts

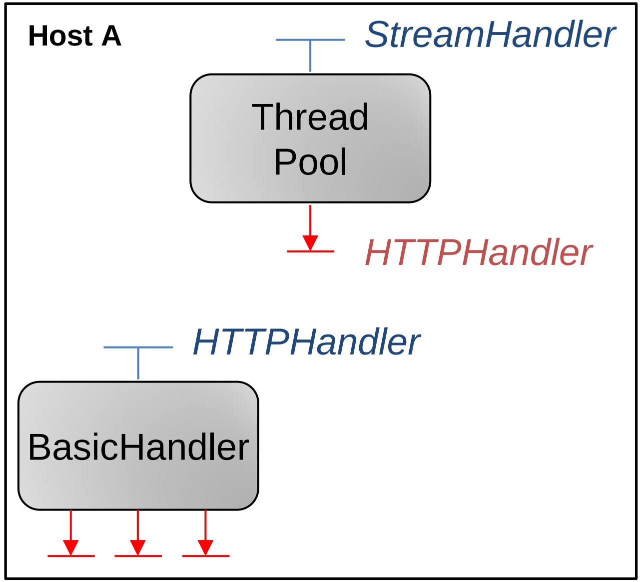
- Even if an interface does have state, we can still replicate it if we have a custom-built plug-in to manage the state

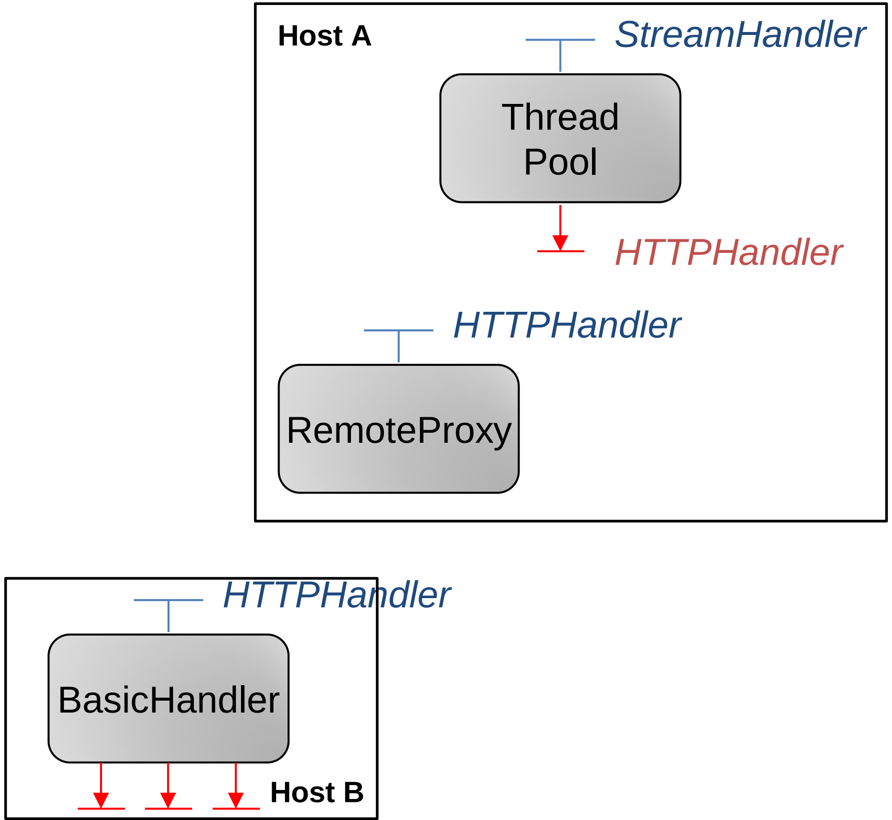
Self-Distributing Systems



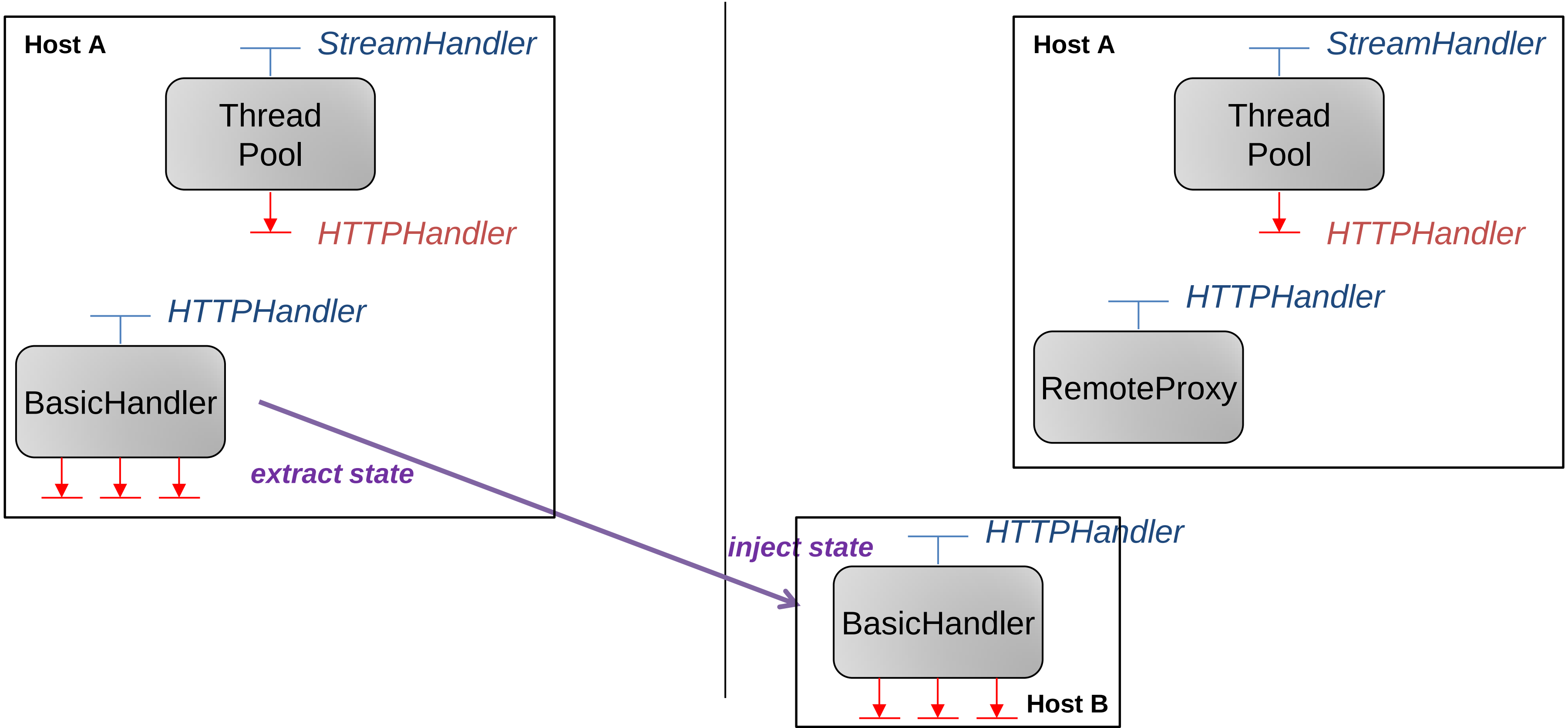


Self-Distributing Systems

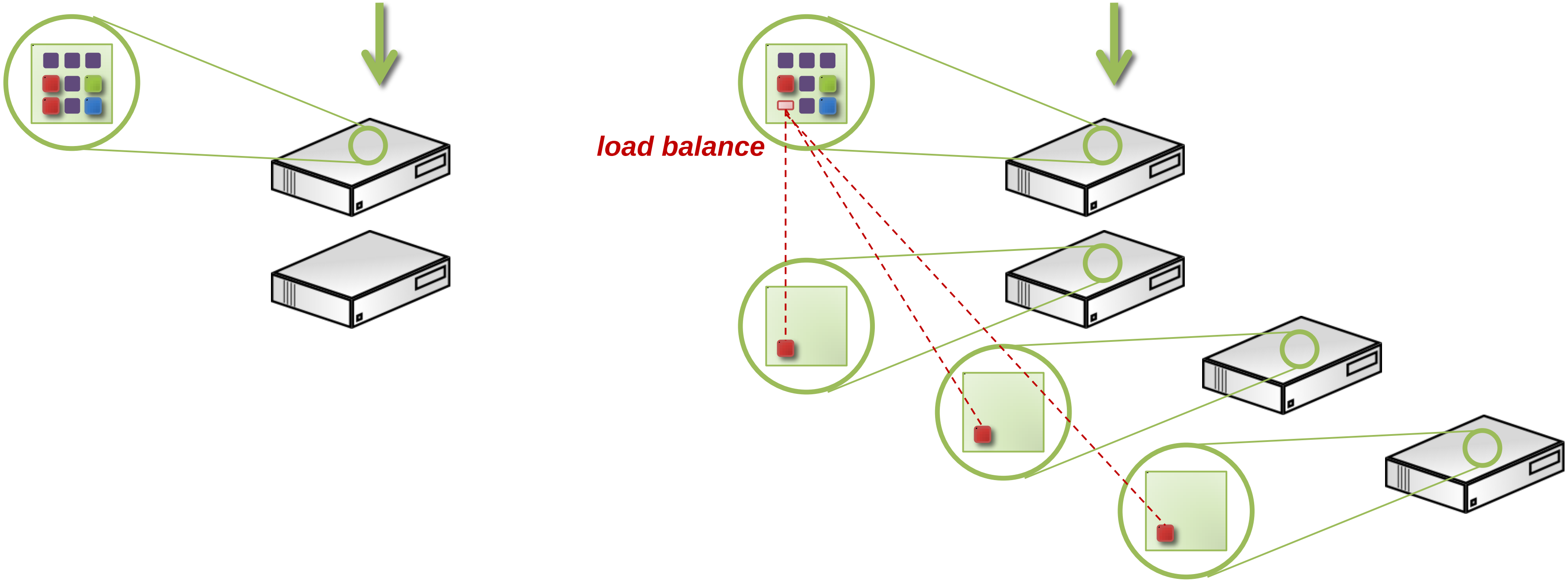




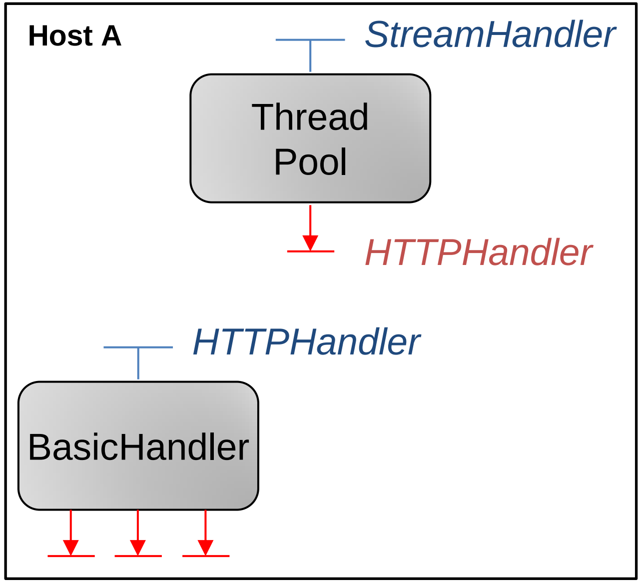
Self-Distributing Systems

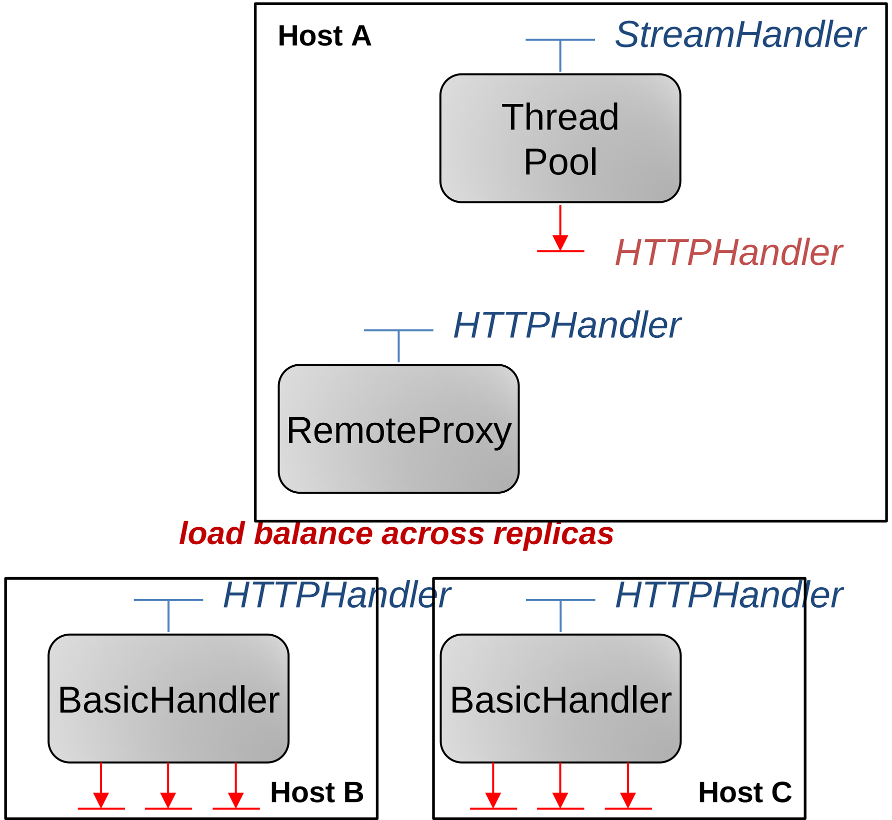


Self-Distributing Systems

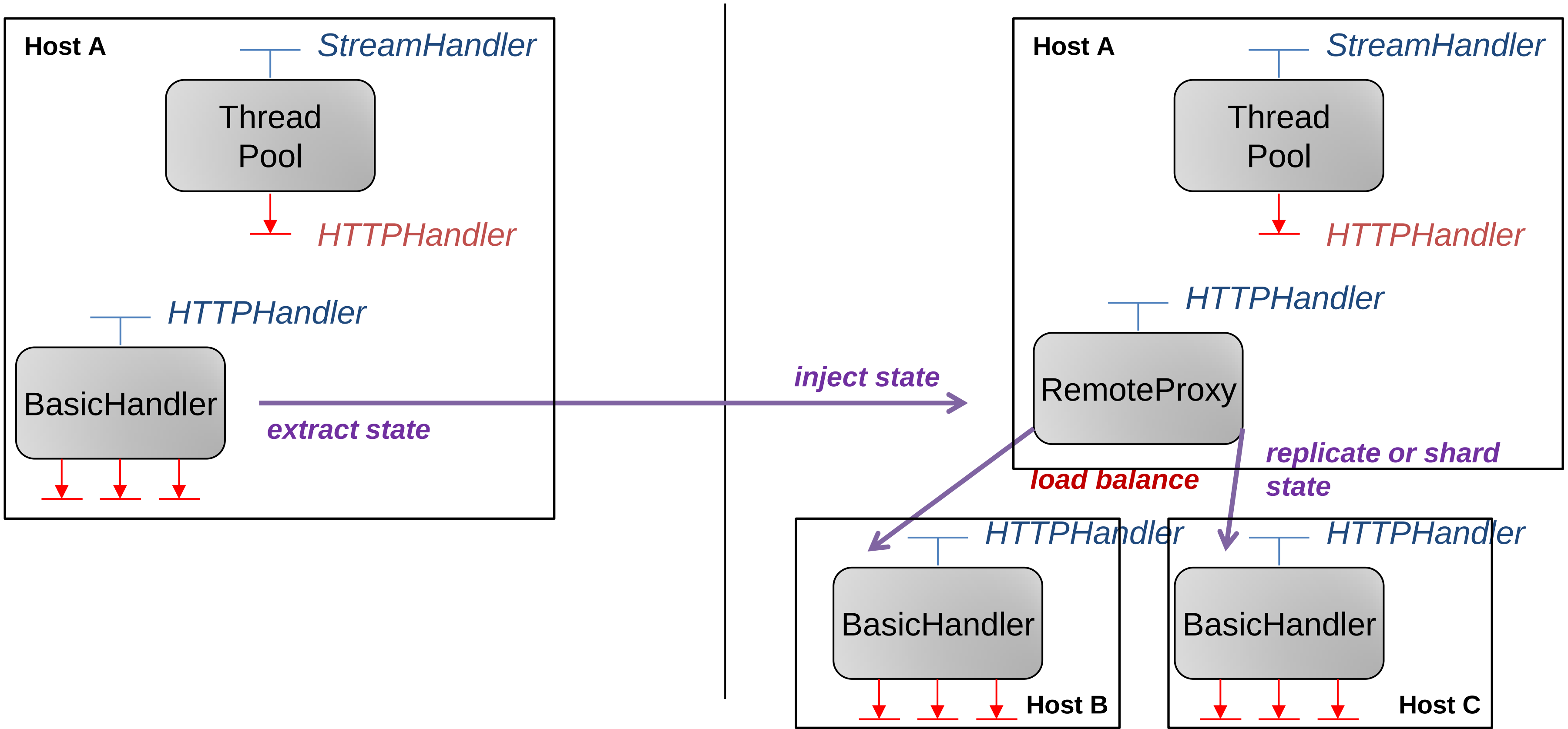


Self-Distributing Systems



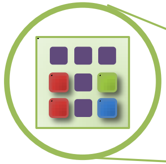


Self-Distributing Systems



Self-Distributing Systems

- What does our action list look like?



$\{c_0, \ldots, c_x\}$
compositions

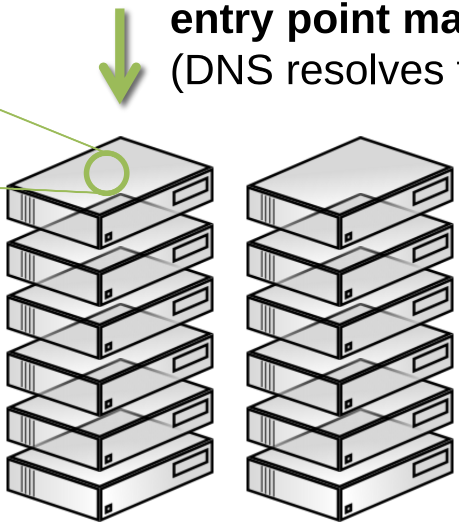
[WebServer.o, BasicHandler.o, ThreadPool.o | 0.ReqHandler:2, …]

[WebServer.o, CacheHandler.o, ThreadPool.o | 0.ReqHandler:2, …]

[WebServer.o, RemoteHandler.o {192.16.30.49}, ThreadPool.o | …]

[WebServer.o, RemoteHandler.o {192.16.30.49, 184.2.45.93}, …]

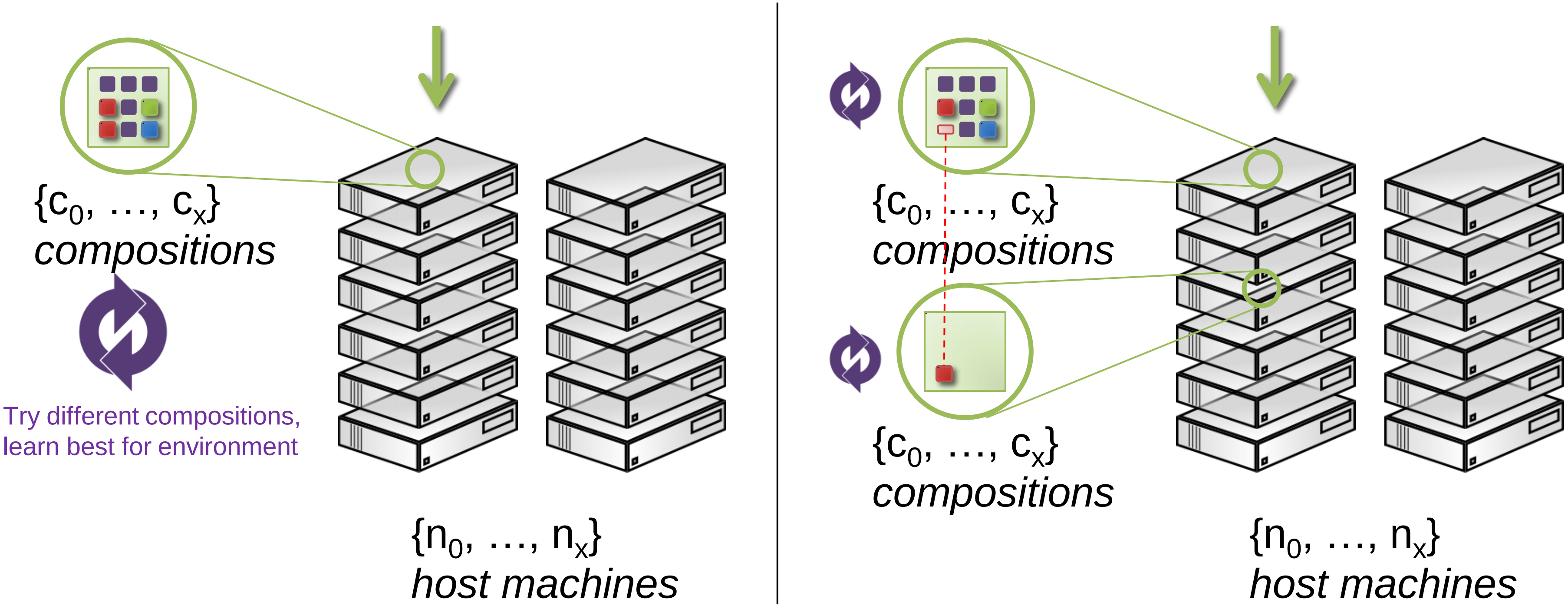
entry point machine
(DNS resolves to here)



$\{n_0, \ldots, n_x\}$
host machines

Self-Distributing Systems

- How does learning work?



Self-Distributing Systems

- What we're doing here is distributing code that was *never designed* for distribution

- What if a remote host fails under this distribution model?

- A seminal paper by Waldo *et al* presents a concise analysis of the problems with distributed objects

A note on distributed computing, Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. Tech. rep., Mountain View, CA, USA, 1994

Self-Distributing Systems

- **Latency**
  - If distributing objects introduces unexpectedly high latency compared to a local object interaction, this can impact system behaviour in unexpected ways
- **Memory access**
  - Some programming paradigms have shared writable memory between objects, which is very hard to synchronise across a distributed system

Self-Distributing Systems

- **Distribution causes partial failures**
  - In normal conditions, a local system is either *fully working* or *not working at all*, so there is no need for the programmer to deal with “partial failure” where one particular object stops functioning
  - In distributed systems, partial failures are the norm, as network links experience dropouts and host machines fail
  - Waldo *et al* propose two possible solutions to this situation for distributed object systems

Self-Distributing Systems

- **Distribution causes partial failures**
  - Program everything as if all interactions are always local, and ignore remote errors caused by network or host problems; this can lead to catastrophic system conditions as remote errors are supressed without any available handling code

  - Program everything as if all interactions are always distributed (even if they're local), so that there is explicit fault handling code for every function call; this introduces huge volumes of failure handling code in every object and also brings associated runtime costs

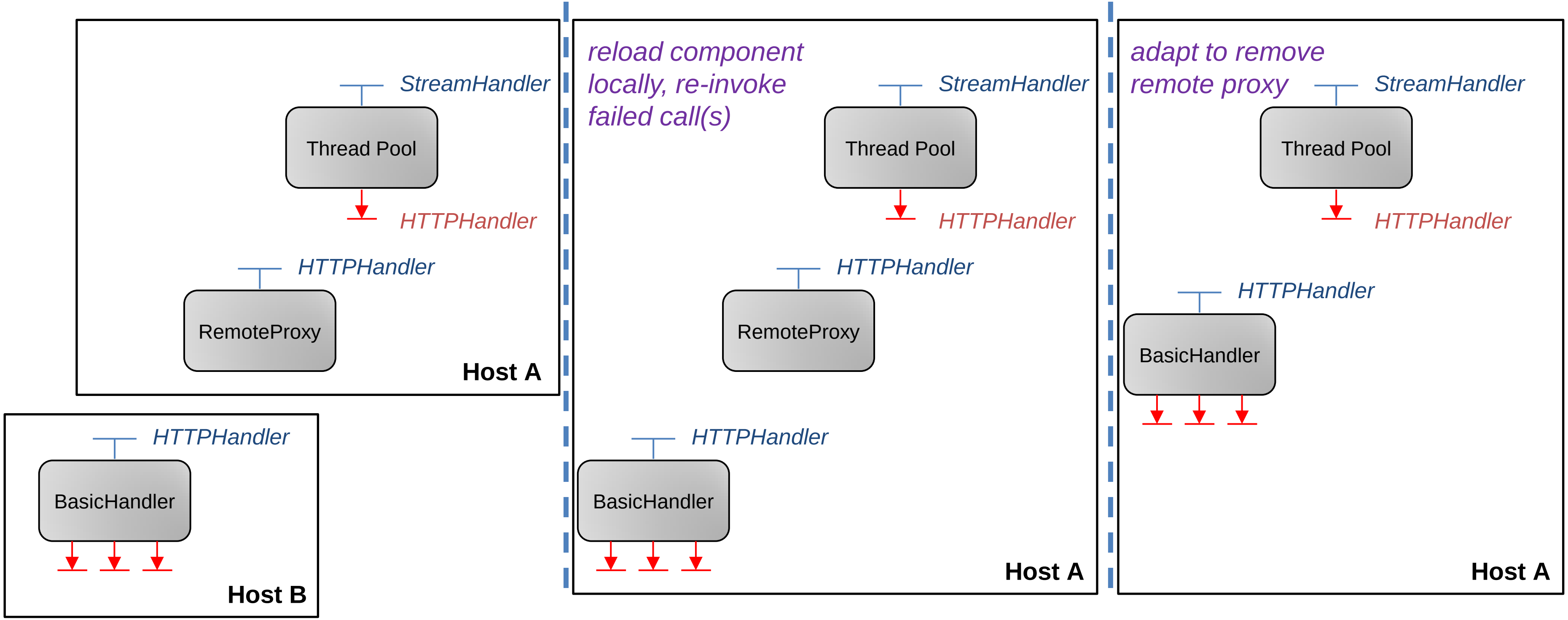
Self-Distributing Systems

- **Distribution causes partial failures**
  - We take a third path: we program everything as if all interactions are always local, so there's no programmer overhead

  - **But** we only ever distribute an object if we can guarantee that we can *automatically* and *seamlessly* recover from remote errors
    - When distributing a component we therefore introduce failure handling infrastructure which can recover automatically from errors without the system ever noticing that an error occurred
    - This is **not** the same as "masking" remote errors, because we're making guarantees that system integrity is assured with auto-recovery

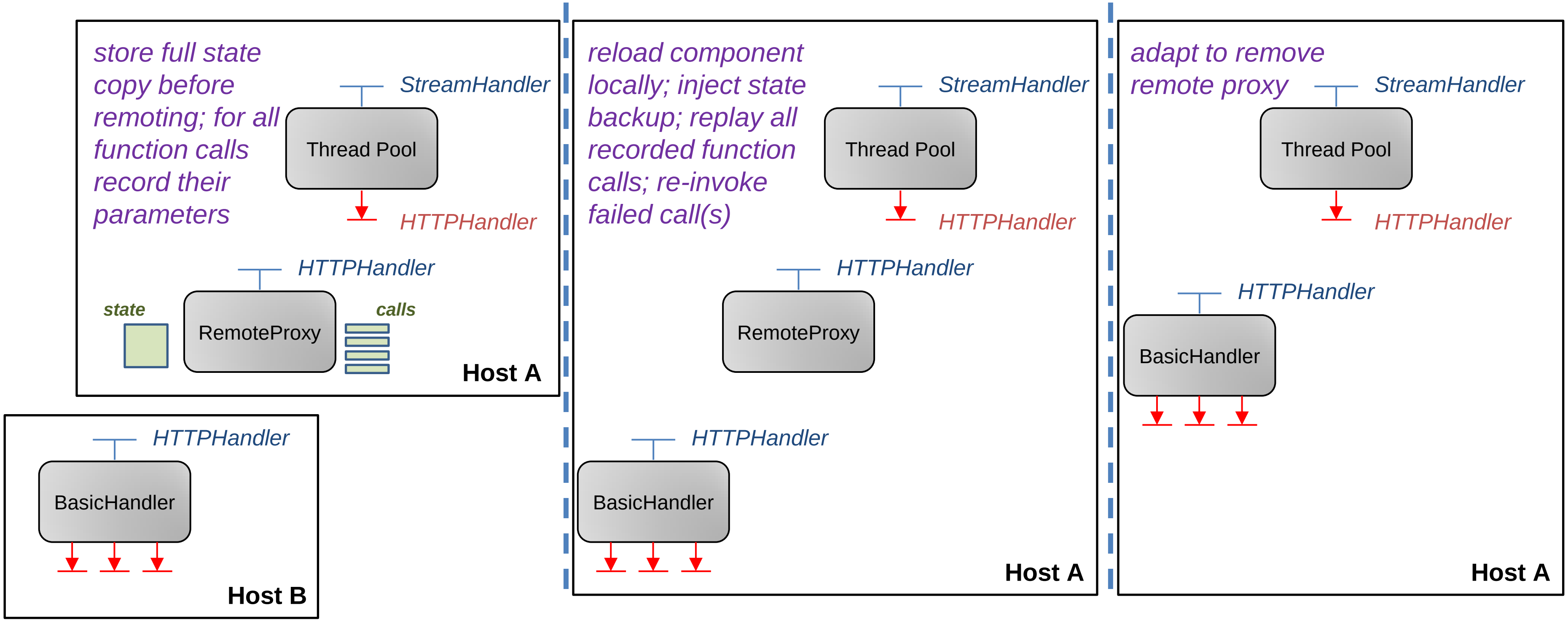
Self-Distributing Systems

- Mechanics of fault tolerance for automated distribution



Self-Distributing Systems

- Mechanics of fault tolerance for automated distribution



Self-Distributing Systems

Class B

Self-Distributing Systems

- We’ve seen how automated distribution of apparently local code requires an automated and seamless approach to ensuring fault-tolerance of the overall system
- For interfaces with large amounts of state, and with frequent interactions, supporting this requirement this becomes very expensive

Self-Distributing Systems

- To counter this issue, we introduce the idea of *explicitly distributed* interfaces, which we call “Class B” proxies
- These interfaces are designed to communicate with a remote service, and have explicit error handling built into the interface definition for remote failures
- Components using a Class B interface will be designed to deal with remote failures reported by the interface

Self-Distributing Systems

- Because failure handling logic is explicit for Class B interfaces, we do not need to employ seamless recovery and rather can leave decision logic to the application

- It’s much more common for things like databases to be represented in this way

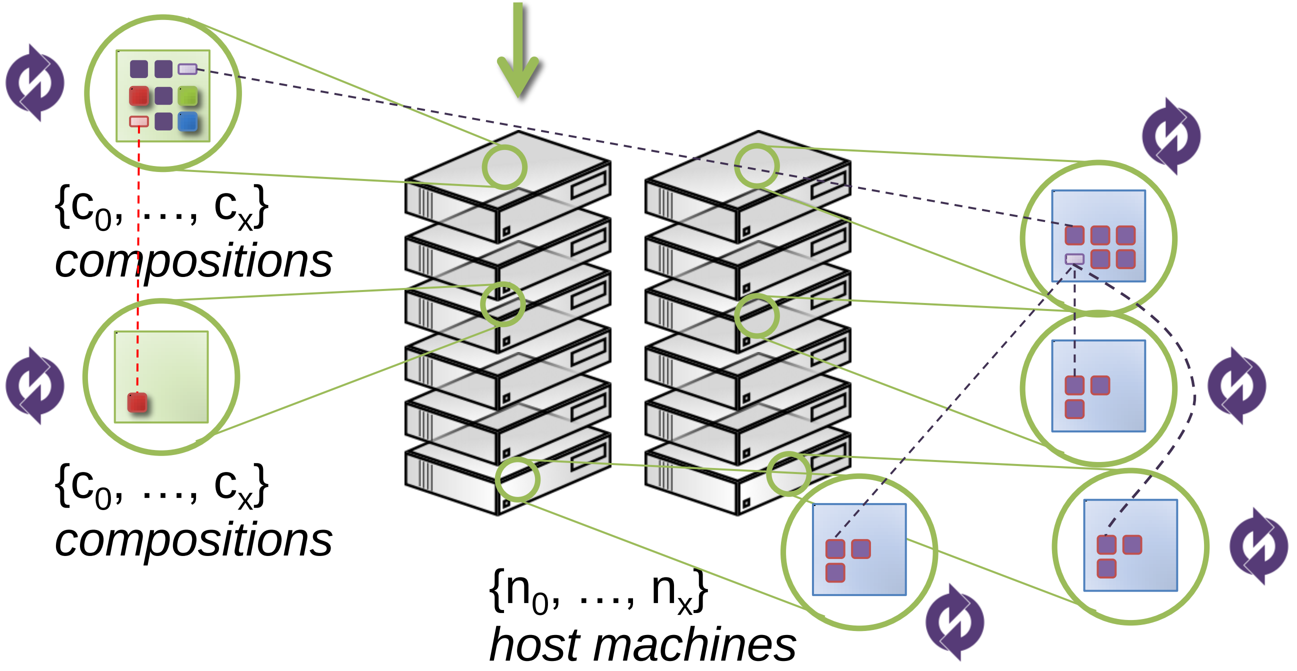
- Besides this detail, the action / reward matrix looks identical to the Class A case

Self-Distributing Systems

Each learning action is one composition, but a composition may include remoting or replication of particular components to particular hosts

This allows us to learn **placement**, co-location, or **replication factors** of all elements of a system, from a simple action list

When we move a component to a remote host, we start a new learning agent at that host to learn about the sub-composition



Summary

- We can take advantage of very strong encapsulation to allow local components to be distributed over a network
  - This applies generically to any component, including buttons on a user interface or machine learning implementations
- Doing this requires automated fault tolerance so that failures are seamlessly recovered
- The result is the ability to learn where to place each element of system logic and how much replication to apply, all from a simple action/reward matrix