The Mobile Object Workbench

Advanced Topics

Richard Hayton

APM Ltd

Talk Overview

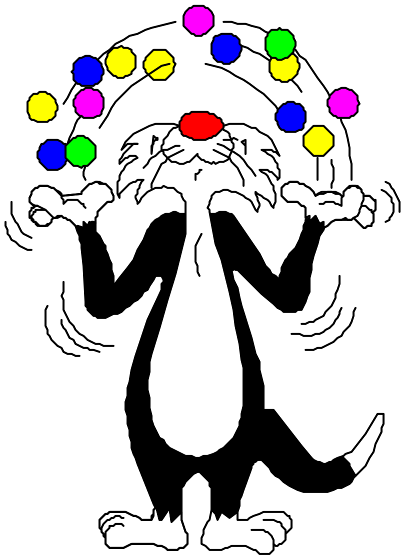
- The mobile object workbench
  - what is it and why?
- Adding advanced facilities to the MOW.
  - True code mobility
    - mobile classes not mobile class names
    - version management
  - Scale
    - federated distributed name service
  - Security
    - distrust between hosts, malicious code

Mobile Object Workbench: Concepts

- Mange units of code - “clusters”
  - Units may be applications, applets, agents, libraries….
- Within clusters
  - “standard” Java mechanisms
  - uniform trust and management
- Between clusters
  - Selectively transparent communications
    - Looks like simple method invocation
    - perform remote access, security checks, transactions etc….

Programming the MOW

- Standard distributed programming
  - Standard Java programming
  - Remote Method Invocation
  - Traders / Name Services
  - Standard API to local services
- Each distributed component is a cluster
  - Clusters can move autonomously
  - Some minor limitations on the code within a cluster
  - Interconnections between clusters ‘stretch’.



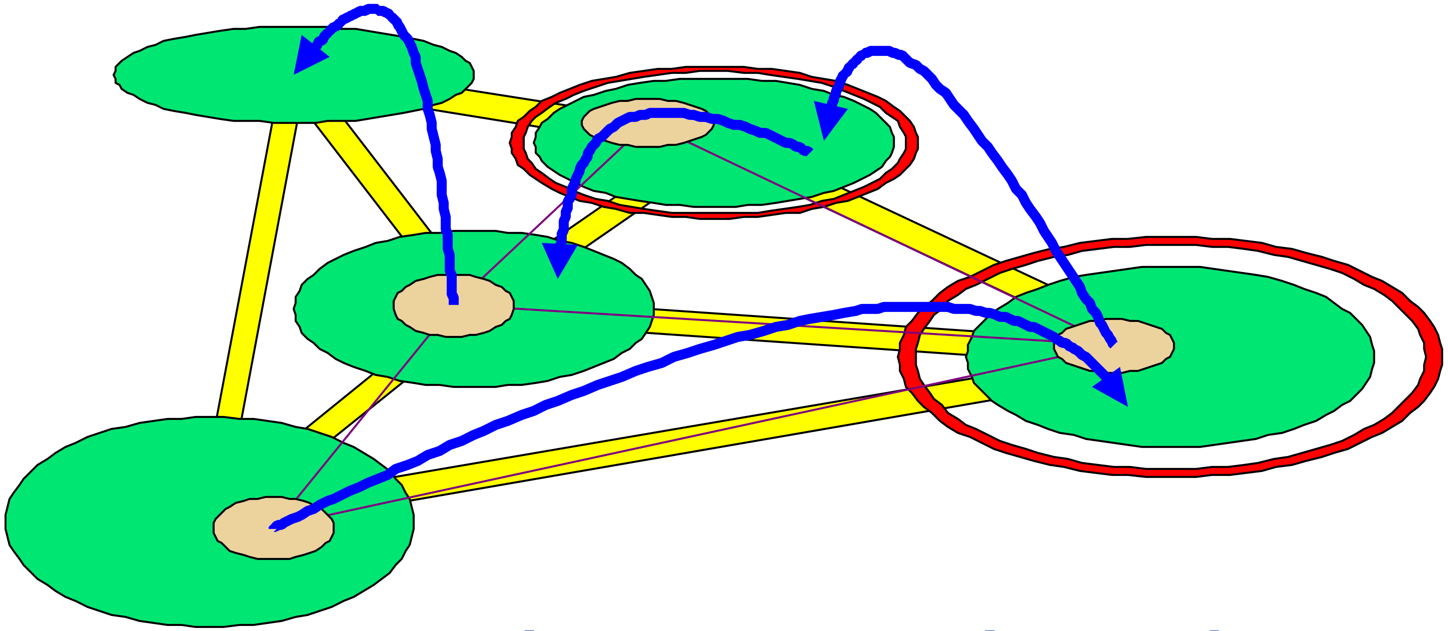
Mobile Object Workbench
Advanced Topics

Code Mobility

Richard Hayton

APM Ltd

Cluster Mobility



Clusters may move between hosts
References between clusters continue to work.

Cluster Mobility = Code Mobility ?

- To move a cluster
  - take a snapshot of the cluster's state
  - copy the snapshot to the new location
  - fix up references
- Basically an RPC
  - `newlocation.recieveCluster(Cluster c)`
- But….
  - the cluster we are transferring will be a *subclass* of Cluster
  - the destination host must be able to resolve this class.

Resolving a 'foreign' class

- Locating the class
  - Where can we load the class from?
- Trusting the class
  - Who wrote the code? Has it been modified?
- Naming the class
  - Have we already loaded a class with the same name?
  - Are there more than one version of the class out there?

These issues apply equally to Agents and Applets

Resolving a foreign class - standard Applet solution

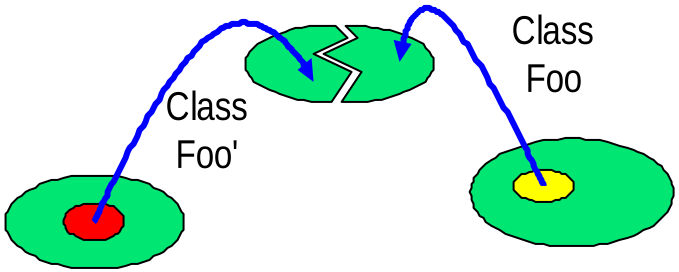
- Locating the class
  - load from the remote web server containing the applet
- Trusting the class
  - digital signatures (JDK 1.2)
- Naming the class
  - don’t care
    - each applet is a self contained name space
    - use one class loader per applet or per web page

Resolving a foreign class -why are Agents different?

- Locating the class
  - Agent comes from an ordinary host not a web server
    - may be less powerful, less well connected
- Naming the class
  - Agents talk to each other
    - They must share definitions of the classes they use to communicate
  - Agents may be related
    - There is scope for sharing classes between agent instances

Naive Approach

- Load all classes from the previous host
  - they are guaranteed to be the correct version
- Use a completely separate ClassLoader/JVM for each mobile object



- Trust the programmer to ensure that communicating clusters are using the same version of a class

Analysis

- Simple & Works
  - if the programmer gets it right
- Network load
  - inefficient - always load classes from previous host.
  - is caching of classes possible?
- Memory usage
  - A host may load the same class many times
  - each host must store classes in two forms, ‘raw’ and ‘loaded’
- Performance
  - cannot optimise same machine communication.

Improving Performance

- 1. Network Class Repository
  - networked resource - e.g. one per ISP or per LAN
  - cache of 'foreign' classes
  - also caches digital signature checks
- 2. Shared Class Loaders
  - reduce memory usage in a single JVM
  - decrease start-up time for a newly moved/created cluster
  - allow optimised same JVM cluster-cluster communication

Network Class Repository

- Maps class identifier to class data
- How to we identify classes?
  - cannot use classname
    - - there may be many version of each class
  - cannot impose a new global naming scheme
    - package naming is supposed to achieve this already
    - prefer something more transparent / automatic
  - use secureHash(classname+classdata)
    - uniquely identifies a class version
    - provides robust global naming

Naming/Storage granularity

- per-class is too small
  - large overhead for secureHash
  - lookup time is O(log(no classes))
  - size of class identifier is large (inefficient RMI)
  - high network overhead
- Use ‘bundles’ - e.g. JARs
  - smaller overhead for secureHash
  - lookup time is O(log(no JARs)) for first class O(log(no classes in JAR)) for rest
  - size of class identifier is small

Size of Class Identifier in RMI (simple)

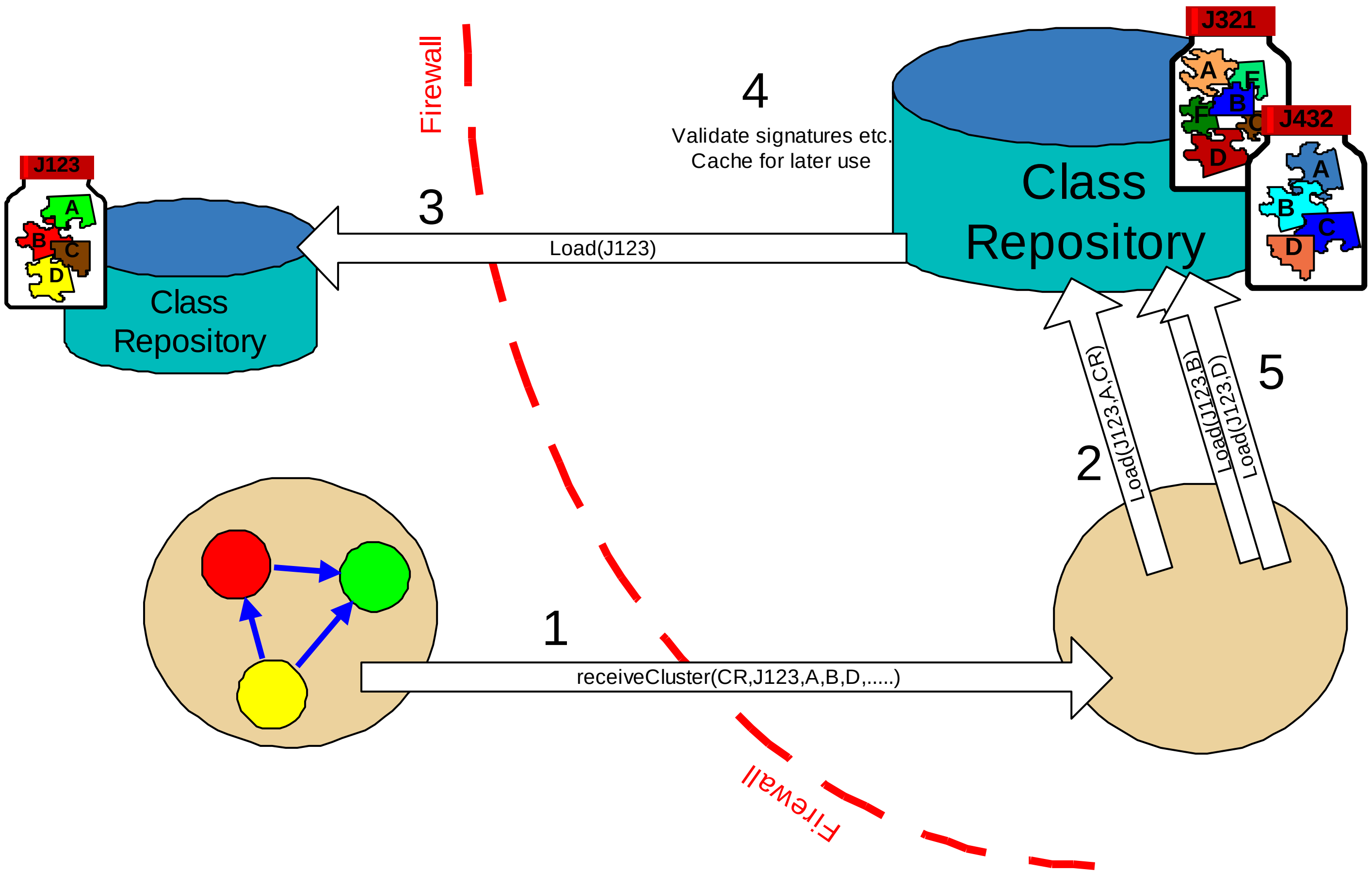
- First reference to a class
  - flag + classname+package name ~40bytes
- Subsequent reference to a class
  - class no. 2 bytes
- Assuming 20 classes, 5 references to each
  - (20*40)+(100-20)*2 = 960bytes

- Does not deal with multiple classes of the same name
- Does not deal with multiple versions of a class

Size of Class Identifier in RMI (Bundles)

- First reference to a class in a newly referenced bundle
  - bundle ID + index within bundle 24bytes+2bytes
- First reference to a class in previously referenced bundle
  - class index 2bytes
- Subsequent reference to a class
  - class index 2bytes
- Assuming 2 bundles, 20 classes, 5 references to each
  - 2*24 + 100*2 = 248 bytes

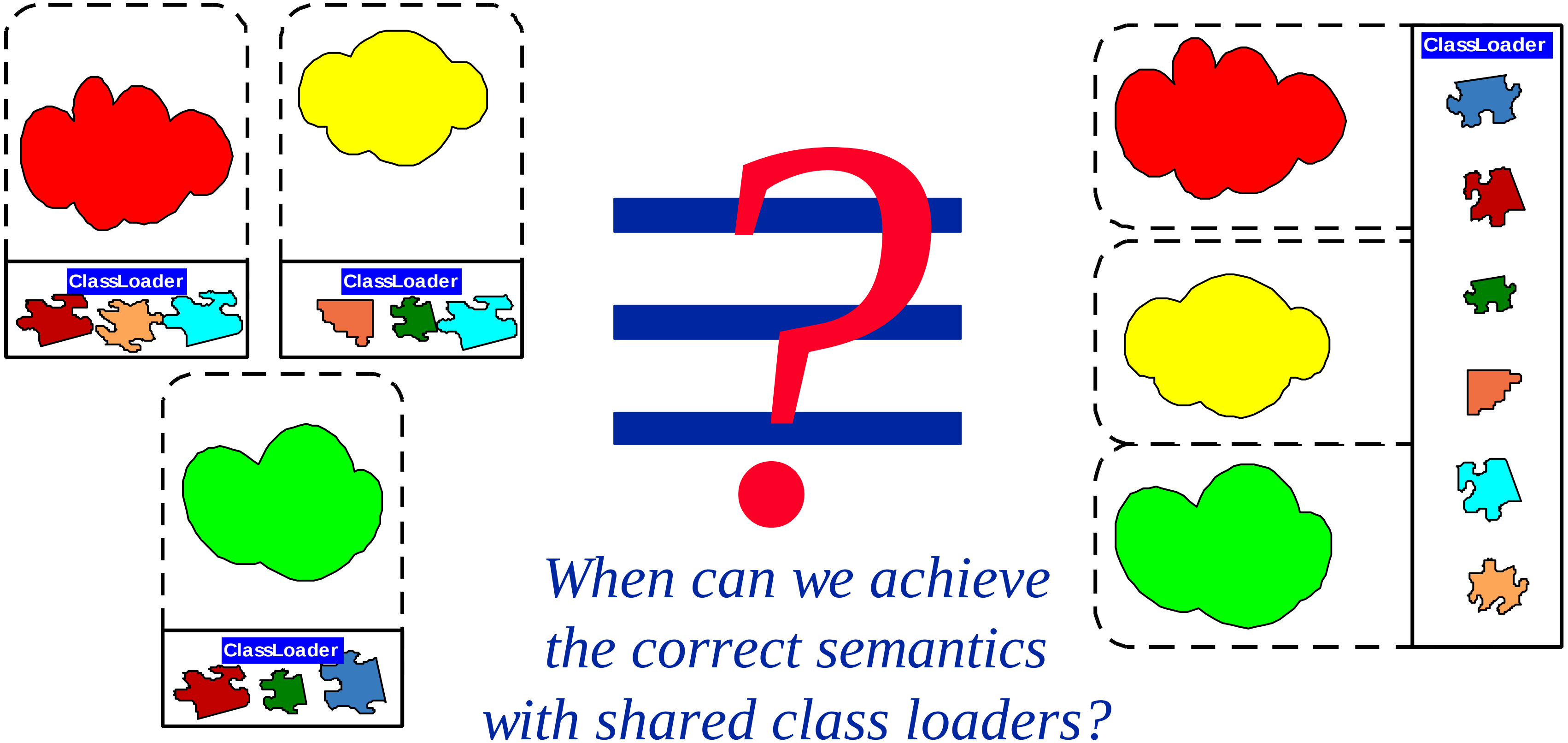
Resolving a set of remote classes



Class Loaders

- Conceptually one per mobile object
  - each requires its own name space / versions of classes
- Share classloaders for efficiency
  - only need load a class once
  - less memory usage / start-up time
  - optimised communication
- Problem?
  - when may we (transparently) share classloaders?

Sharing Class Loaders



When can we achieve
the correct semantics
with shared class loaders?

Which class loader may a class live in?

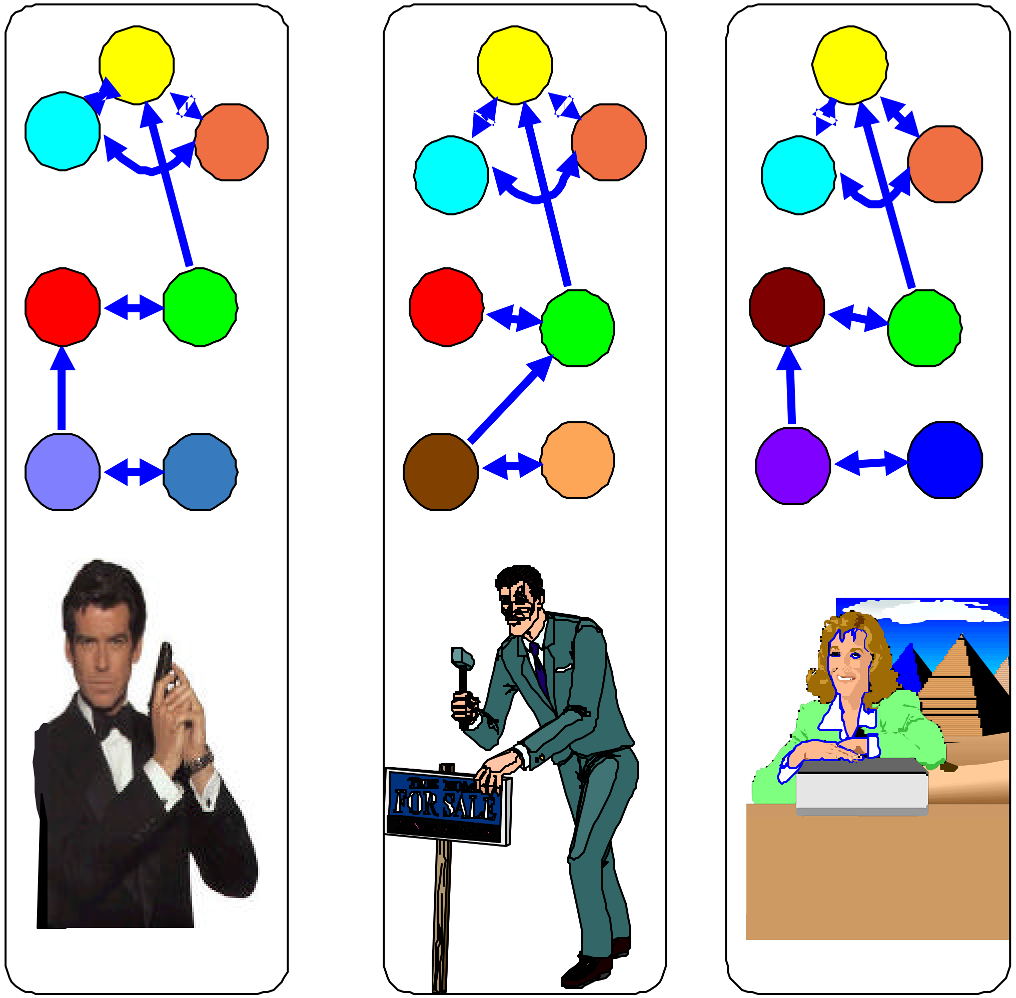
- Rules:
  - All classes in a package must be loaded by the same classloader
  - If two packages refer to each other, they might as well be in the same class loader.
  - If package A references package B, but B does not reference A, then A may be in a *child* class loader to B.
  - Each class loader can only load one class with each name.
  - Classes with static data must be treated carefully
- Providing the rules are met
  - any class may be in any classloader

Individual Class Loaders

Each agent has its own class loader

Some classes are loaded several times.

Two versions of one class

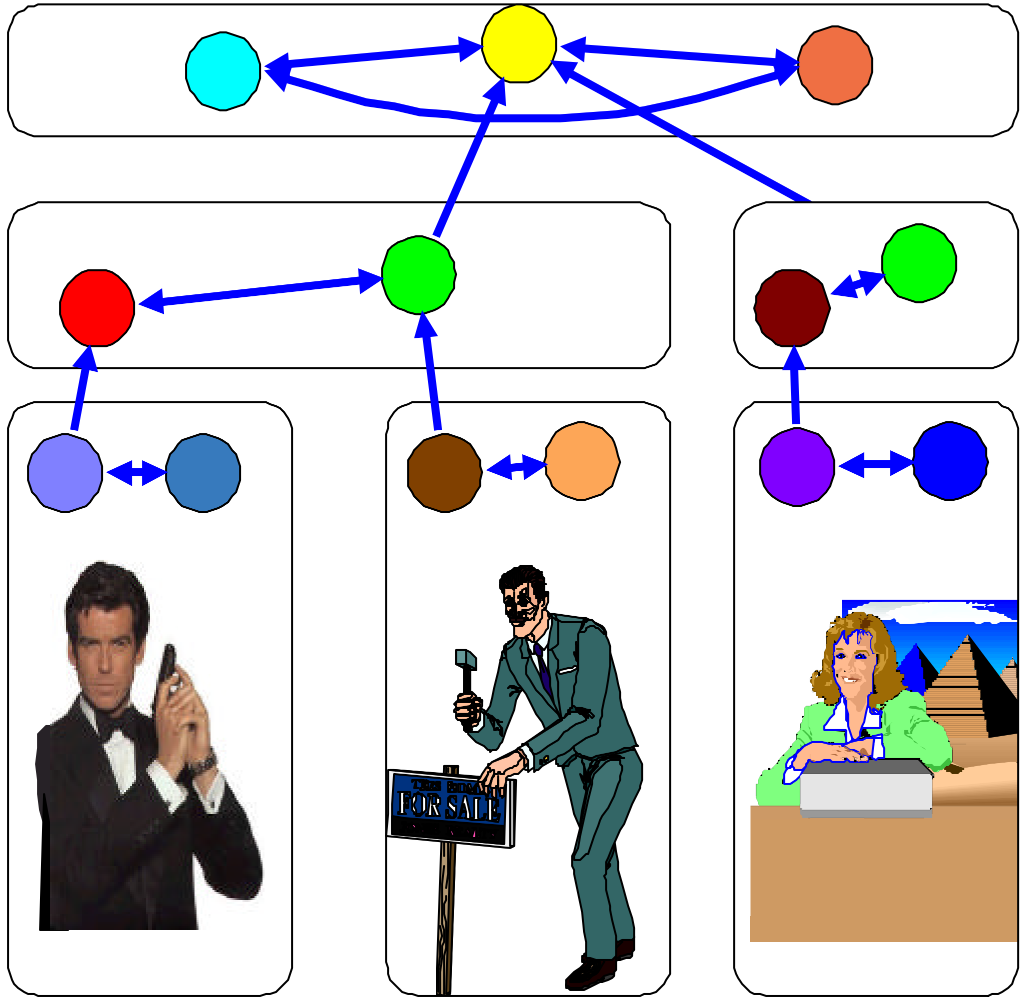


Sharing Class Loaders

Most classes are only loaded once.

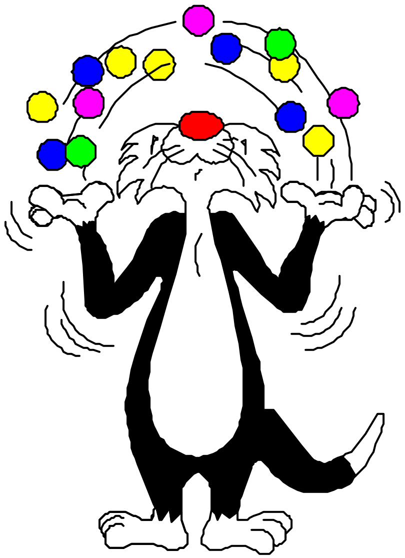
Still support for multiple versions

We can analyse packages for cross references off line (In Repository)



Summary

- Code may be authored in many places
  - cannot rely on classnames to uniquely identify classes
  - we can use secure hashes as a better identifier
  - Class repositories act like web caches
- We may need to load several versions of a class
  - 'True' versions, due to ageing code
  - Different classes with the same name.
- We must support multiple class loaders
  - analysis allows us to reduce the number needed.



Mobile Object Workbench
Advanced Topics

Scaleable Object Relocation

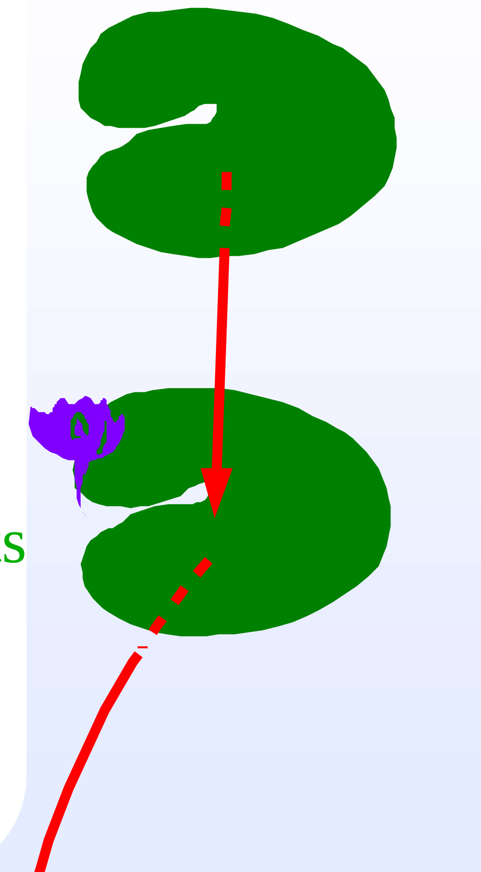
Richard Hayton
APM Ltd

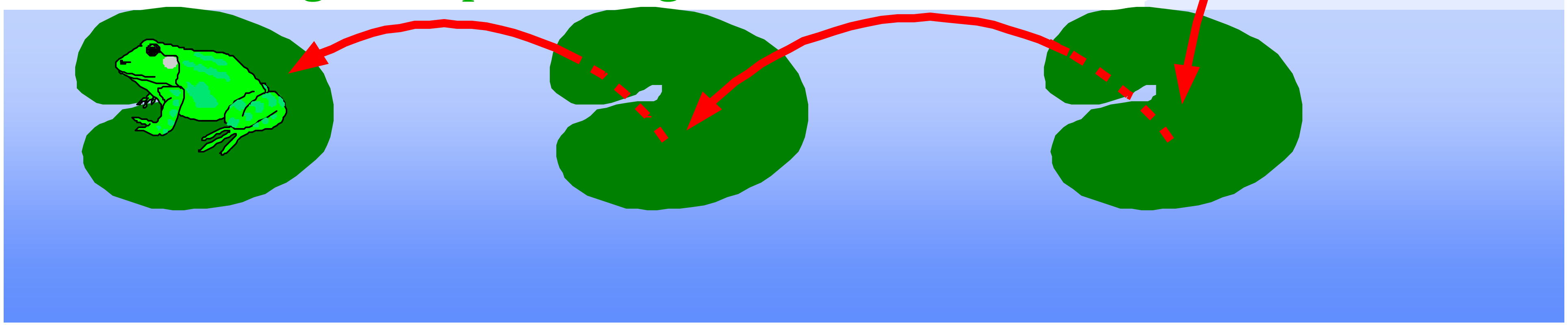
Distributed Relocation Service

- Locating an object that has moved
  - even if some hosts have failed
- Managing many millions of objects
  - created at many hosts, all over the world
- Dealing with deceit
  - claims by a host that it has an object it does not
  - malicious reuse of ‘unique’ names
  - one host or object masquerading as another

Locating a moved cluster

- Usual approach is Tombstones…..**but!**
  - Cost of resolution can be high
  - Very susceptible to host failure
  - Hosts accumulate ‘garbage’
  - Optimisations are susceptible to malicious hosts
- Other issues when considering alternatives
  - Cost of object creation and movement
  - Background processing





New Name Resolution Scheme

- Designed for a large scale environment with poor reliability and mutual distrust
  - i.e. for FollowMe in a WWW environment

- Implemented as a set of "stages"
  - each is a refinement on the previous stage

- Current status
  - stage one is implemented

Approach

- Assume objects don’t move
  - low object creation cost
  - only use auxiliary mechanisms when a move occurs
- Allow for any host to fail
  - objects are not permanently tied to their first location
  - reduce dependence on ‘client’ hosts
- Optimise from experience
  - allow the cost of relocation to be spread independently from movement.

Stage One: Directory Based

- On cluster creation:
  - choose a directory d but don't use it yet
  - Name the cluster (d, current address)

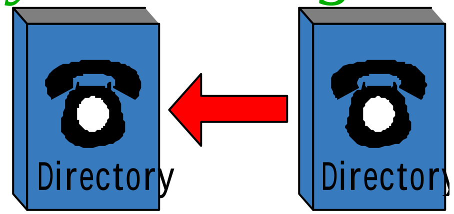
- On move
  - update directory d with old address ⇨ new address
- On lookup
  - try the previous address, if it fails contact d

Analysis

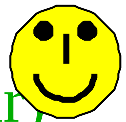
- Security/Integrity
  - High trust in directory
  - Clusters can choose an appropriate directory 
  - Hosts cannot fool others into thinking they have a cluster
- Move/Lookup Cost 
  - At most two additional calls
  - One may be to a distant host if the directory is ill placed 
- Reliability
  - Require access to 1 host out of 1 possible host 

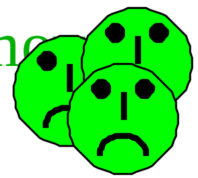
Stage Two: Reducing Move/Lookup Cost

When the system decides that a directory is no longer suitable for a particular cluster:



- Pick a more suitable directory d2
  - Update the cluster's name to (d2, current address)
  - Update the old directory d with (current address ⇨ d2)
    - Tombstoning directories
- Analysis
  - Lookup/Move: 2 calls (directory normally near)
  - Reliability: n+1 hosts out of n+1 after n directory mo

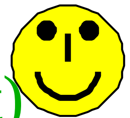




Stage Three: Improving Reliability

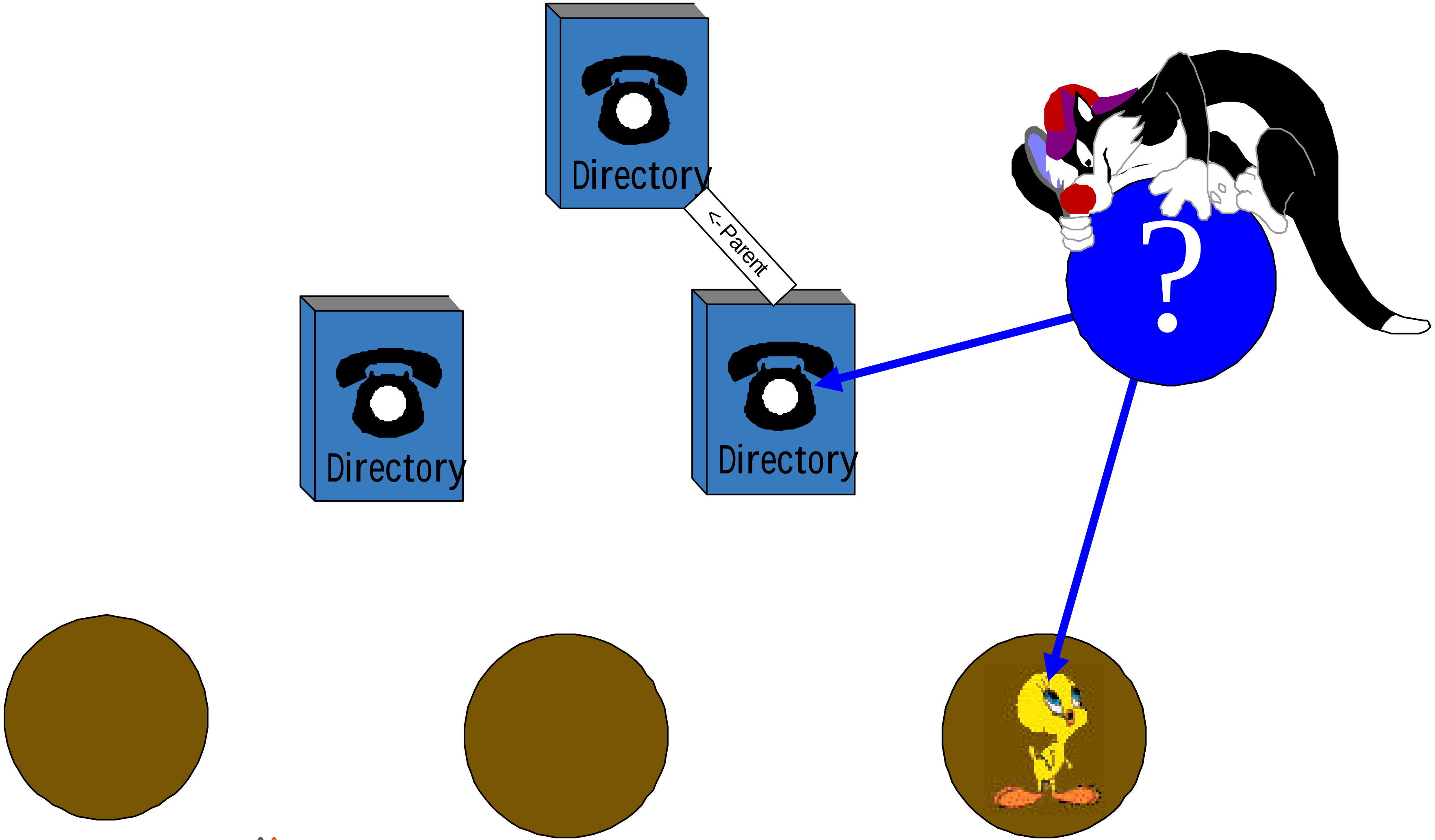
- Each directory is given a well known parent
- A directory may copy any entry to its parent
- If a directory is uncontactable, the parent is asked

- Analysis of reliability:
  - n hosts out of 2n (each tombstone or its parent)

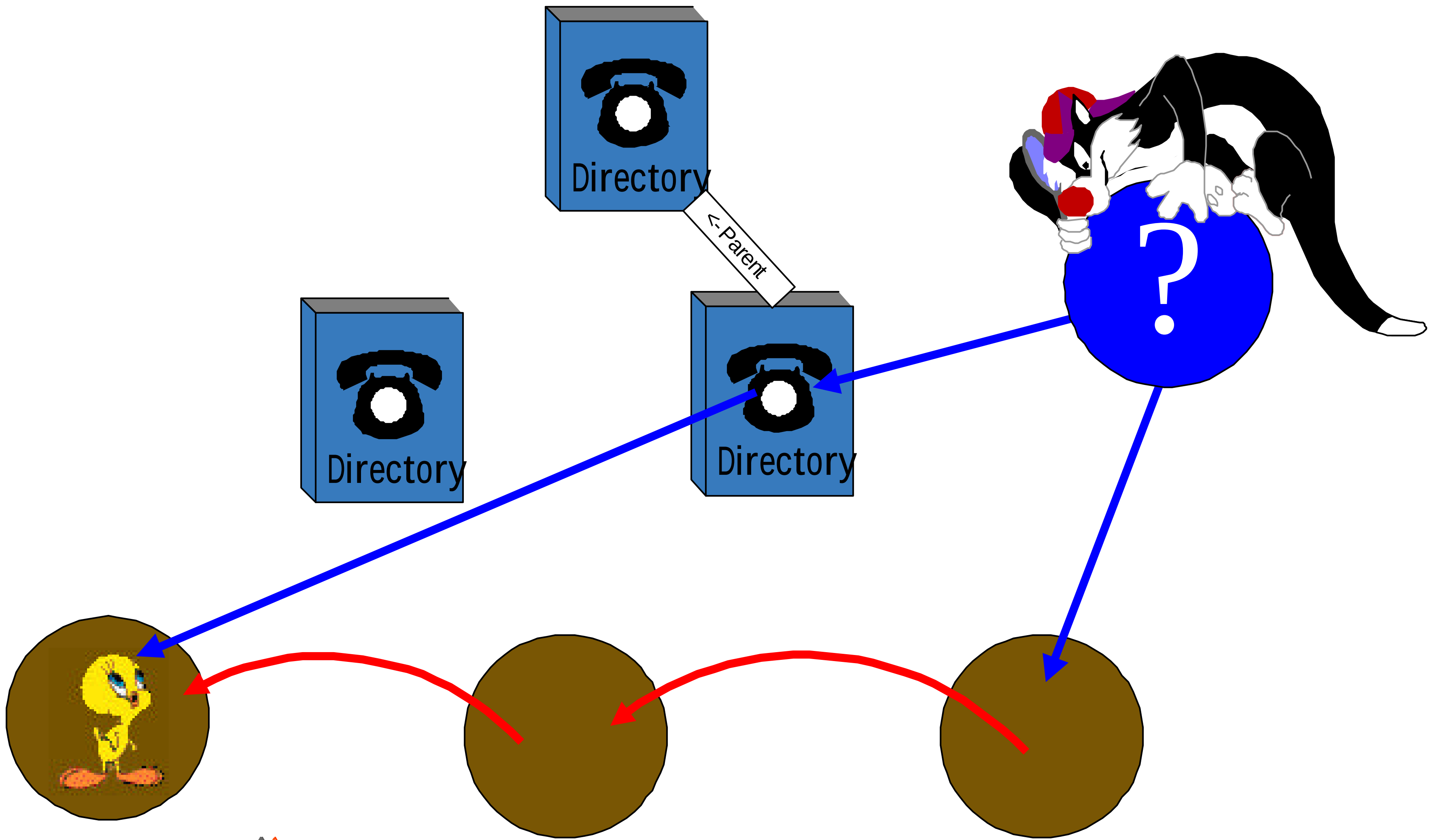


- Analysis of background cost
  - Low - *if we only copy to parent when we create tombstones*

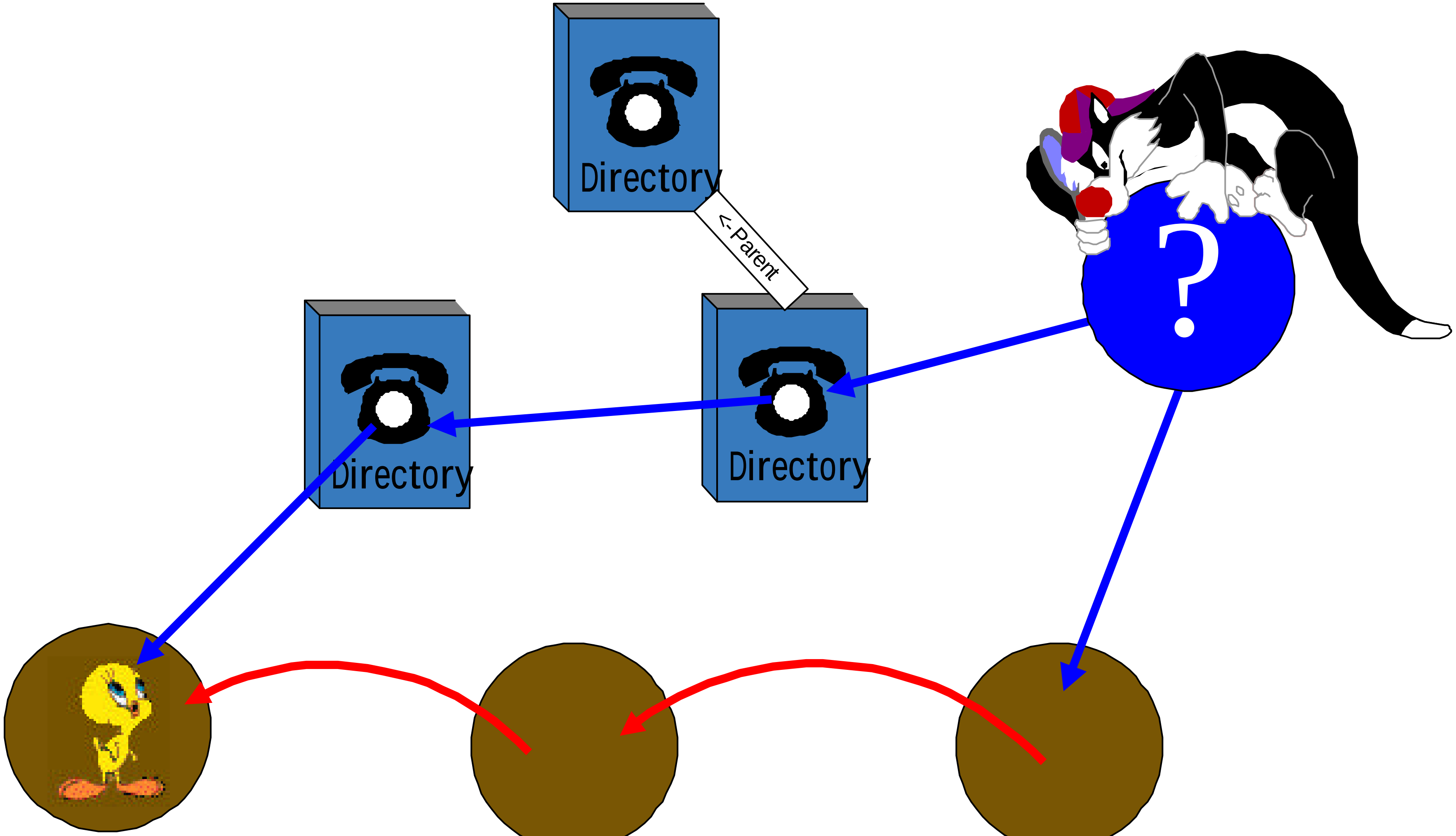
Catch the Birdie…...



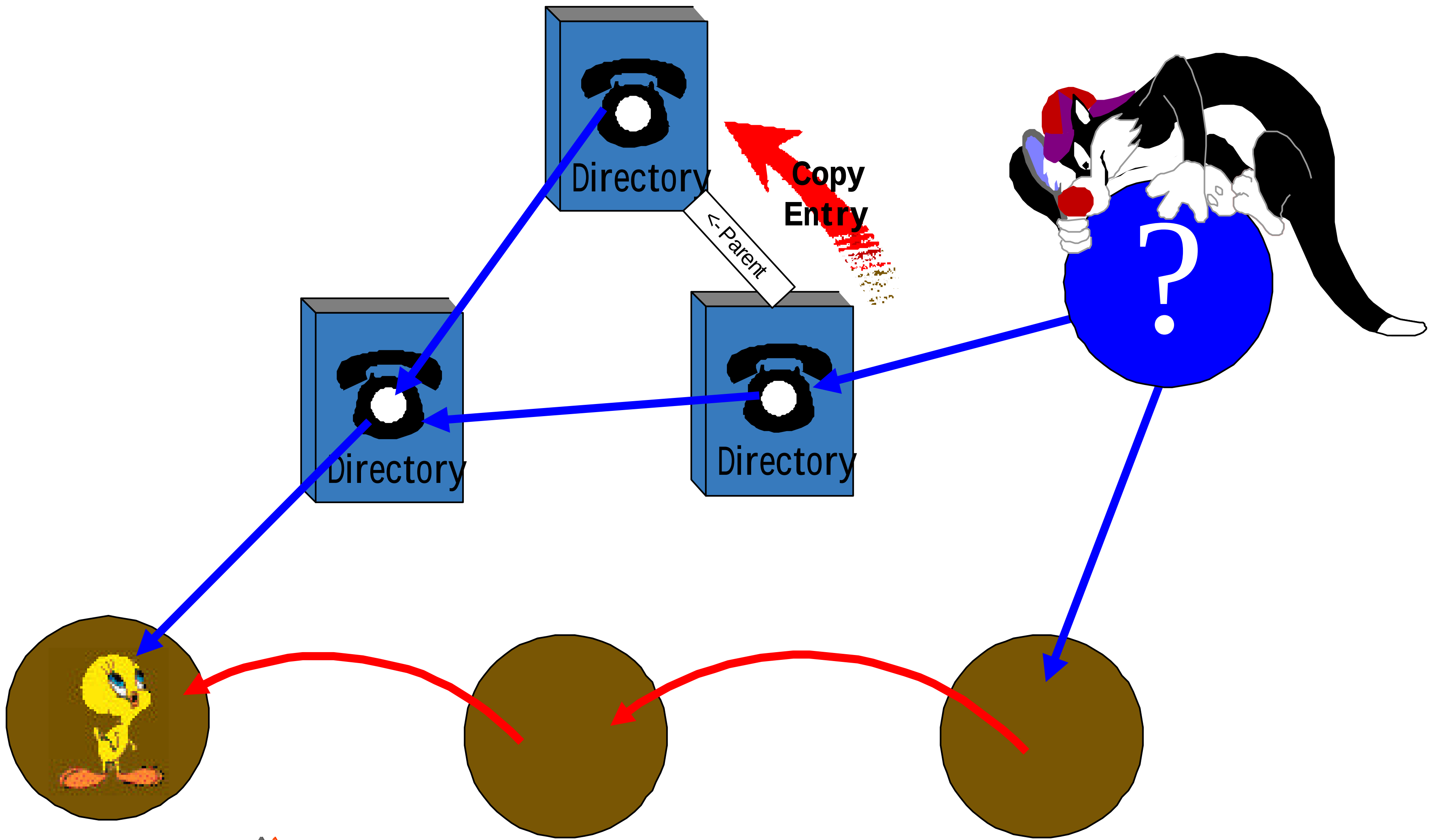
Catch the Birdie......



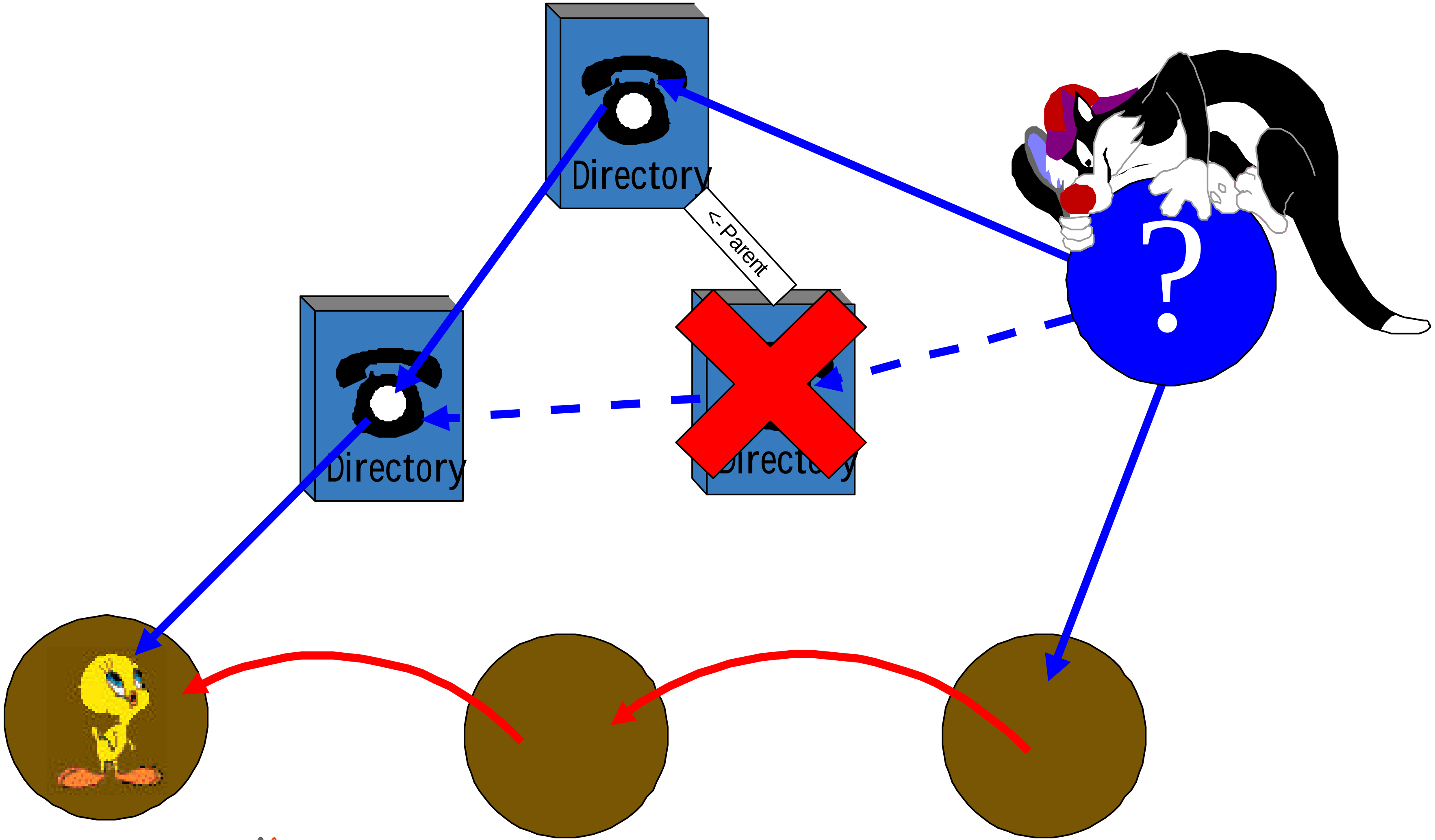
Catch the Birdie…...



Catch the Birdie……



Catch the Birdie.…..



Stage Four: Reduce Garbage Accumulation

In the current scheme a directory can never forget an object that has not been deleted, even if it is 'long gone'

- Solution:
  - A directory may copy an entry to its parent, and delete the local reference
  - When a client requests a lookup of an unknown name, the directory bounces the request to its parent
  - NB. There must be a short chain of parents or invalid names will take a long time to return definite failure on lookup
- Stage Five: mobile places……

Deployment of Directories

Level 0 directories
in mobile hosts (e.g. portable computers)

- Level 1 directories
  - On servers. Approx. 1 per LAN
  - have parents at level 2
- Level 2 directories
  - Backup servers. Approx. 1 per LAN
  - no parents

Managing many moves

- Directory stores `old name ⇨ new name`
- If an object moves many times there will be many entries
  - `address1 ⇨ address2`
  - `address2 ⇨ address3`
  - ...
- We may not delete any of these entries, as other objects may hold any of the addresses.

Optimisation

- Split the address into two parts.
  - address = (ID, current address)
- Then we need only store the latest address
  - (ID, address1) ⇨ (ID, address2)
  - (ID, address2) ⇨ (ID, address3)
  - ...

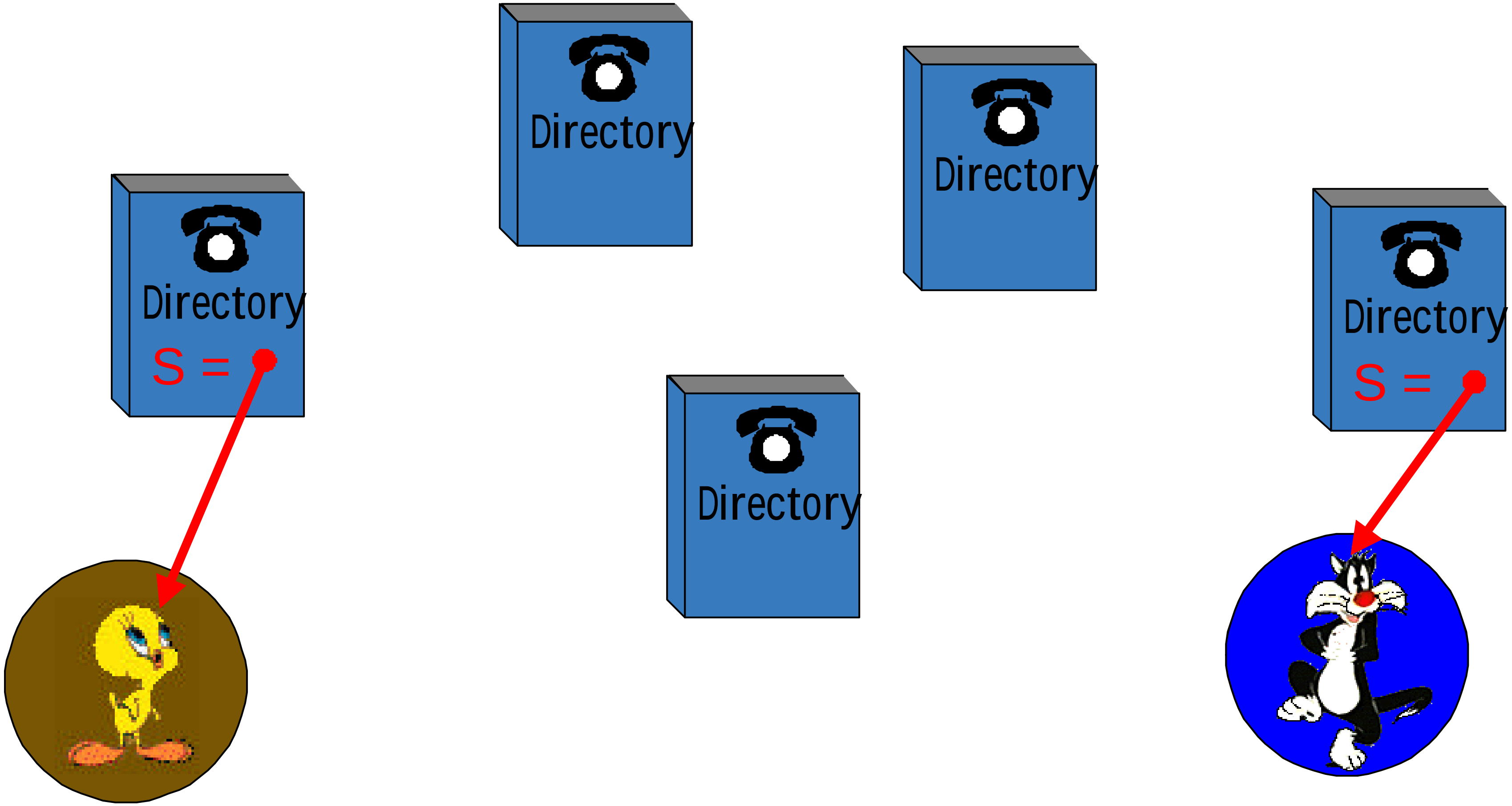
  becomes

  - ID ⇨ address3
- This is a common, and simple, optimisation
  - However, it introduces a security loophole

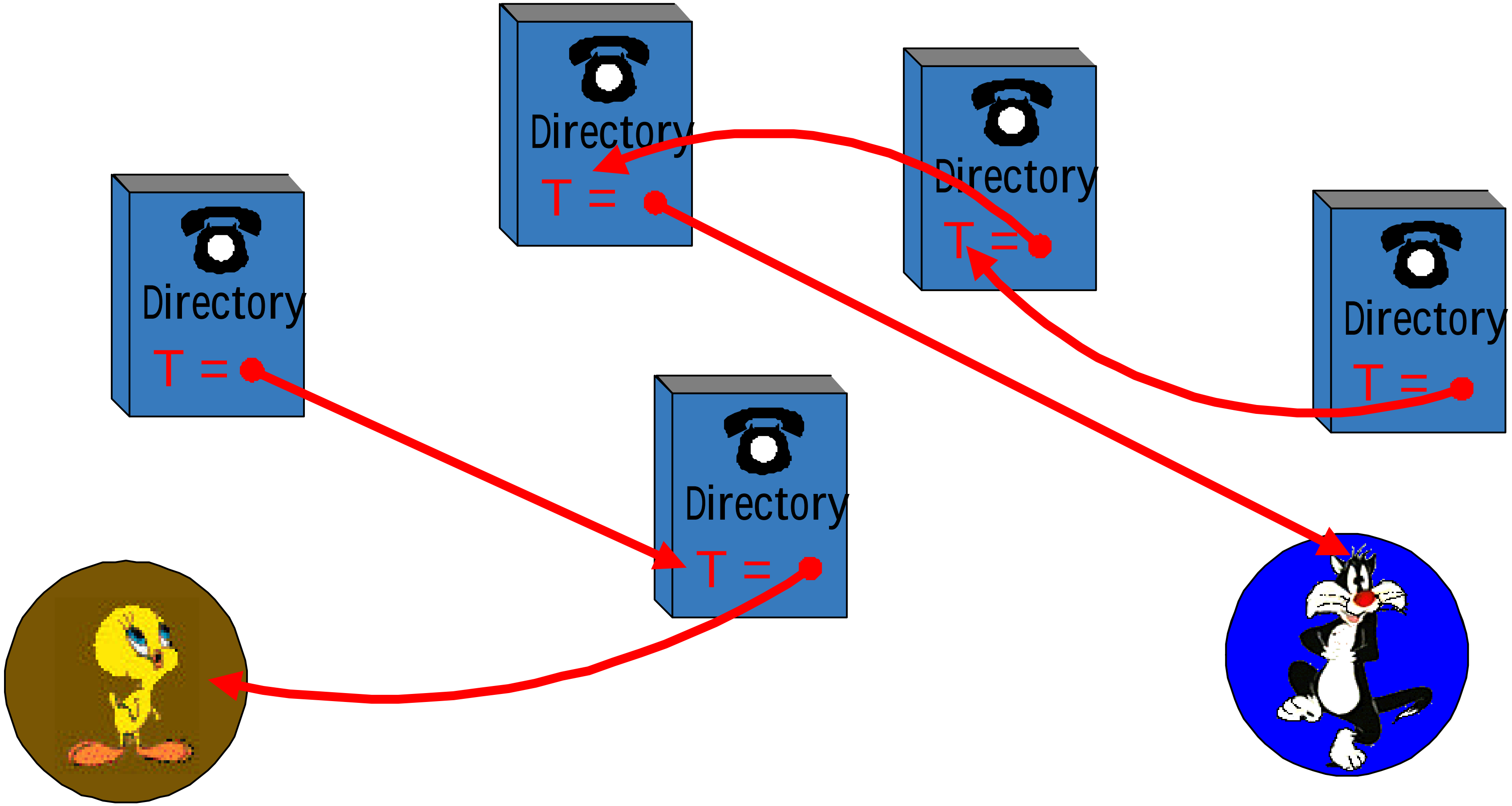
Attack on Optimised Relocation Service

- Attack
  - Someone else can reuse a ID
  - A directory cannot tell if a ID is original or forged
  - A directory storing a forged ID cannot store the original ID as well (it cannot tell them apart)
- Solution
  - Used the optimised approach until we get a clash
  - assign a new identifier whenever it is needed
  - Does not prevent attack, but reduces it to a resource attack

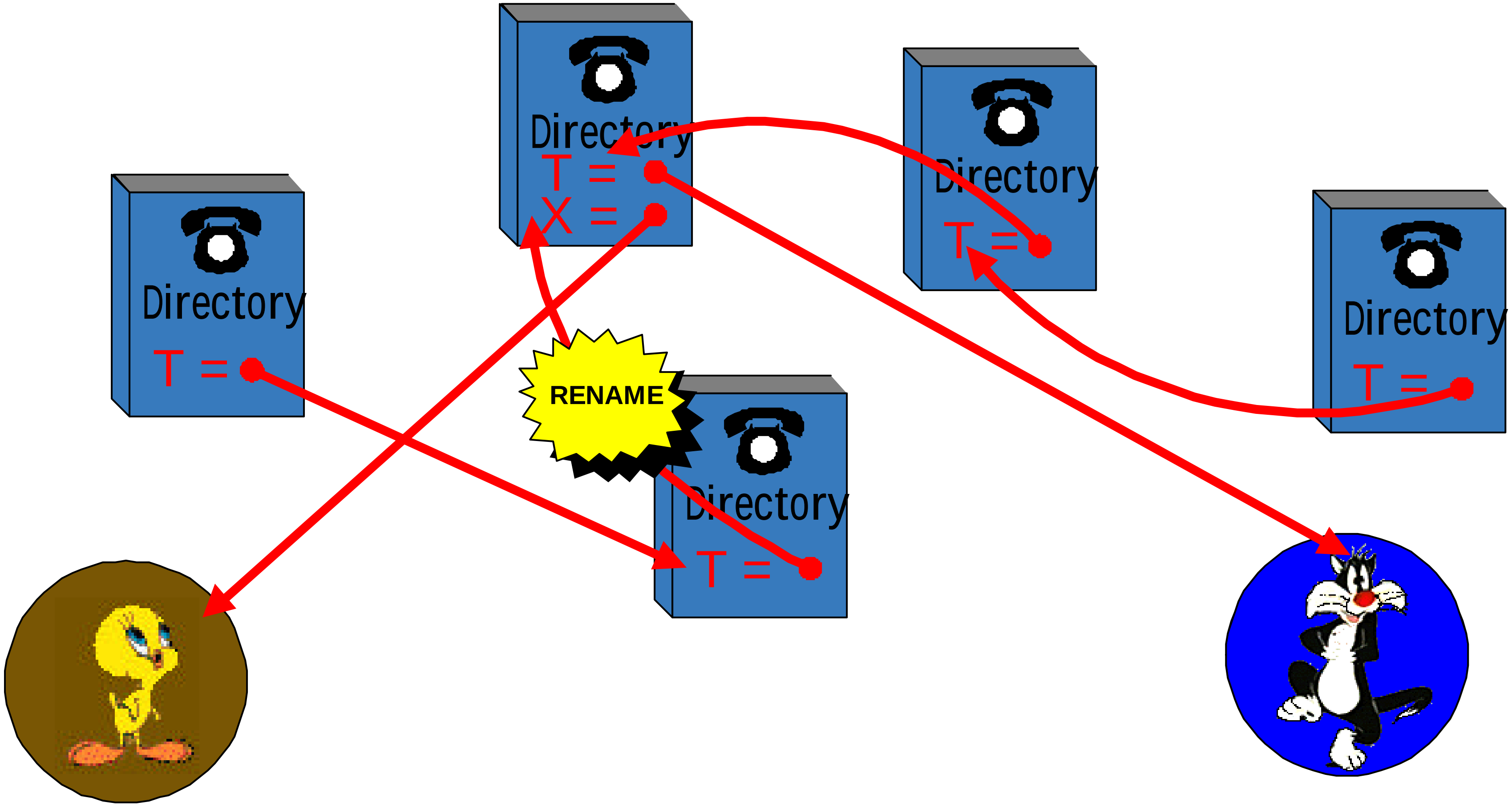
Two clusters with the same identifier



No problems if they don't share directories



Change IDs to avoid clashes



Other attacks

- A host can masquerade as a one containing object A and then tell the relocation service that A has moved.
  - We must use authentication when talking to relocation service
  - A simple scheme is sufficient
    - we only care that two messages have the same origin
    - keys can be created dynamically - no management overhead
  - However, there may be authentication services for other purposes
    - make use of whatever is available

Summary

- Managing a world-wide name space is tricky
  - Security has to be an issue
  - network partition is inevitable
  - cannot predict optimal deployment in advance
  - centralised co-ordination doesn't work
- We have a solution
  - our architecture meets the above needs
  - it does not require global consensus
  - it allows a mixture of (in)secure / (un)robust / (un)replicated services