LocALE: a Location-Aware Lifecycle Environment for Ubiquitous Computing

Diego López de Ipiña
Laboratory for Communications Engineering,
Cambridge University Engineering Department,
Cambridge, United Kingdom
dl231@eng.cam.ac.uk

Sai-Lai Lo
AT&T Laboratories Cambridge,
24a Trumpington Street,
Cambridge, United Kingdom
S.Lo@uk.research.att.com

Abstract

The LocALE (Location-Aware Lifecycle Environment) framework provides a simple management interface for controlling the lifecycle of CORBA distributed objects. It supports mechanisms for the remote construction, movement, removal and recovery of heterogeneous software objects in a location domain, i.e. a group of hosts on a network within a given physical area. Client applications use LocALE to intelligently control their required services' location and relocation in the network. LocALE offers load-balancing, automatic activation, and fault-tolerance facilities for the services whose lifecycles it controls. It provides the middleware necessary for the efficient implementation of location-aware mobile applications in richly equipped network environments. LocALE's infrastructure has been tested with the development of several follow-me applications that dynamically move with their users as they change location. For illustration, two of these follow-me LocALE-enabled applications are described.

1. Introduction

Computing devices are becoming increasingly ubiquitous; computationally capable devices are present in almost every room of a modern office, and are rapidly invading our homes. Moreover, those environments are also becoming aware; sensor networks such as the Active Badge [14] system allow applications to respond to the location of users.

In order to provide a location-aware environment, any particular software service should be available to the user wherever she may be, and follow her as she moves. The goal is to allow users to move freely throughout a building or greater environment without undue degradation of the computing and communications resources available to them. To enable this the following are necessary:

1. Rich and up-to-date information about the environment, gathered from a range of environmental sensors, resource monitors and static data repositories. In particular, the locations of people and computing resources, and the capabilities of these resources.
2. An infrastructure to make possible the transparent migration of objects associated to a user from one address space to another.

The first requirement is being addressed by work on Sentient Computing [7], a research field that uses sensor and resource status data to maintain a model of the world shared between users and applications. The SPIRIT project [5] creates such a model combining the information collected from resource activity monitors (CPU, disk, network) on each LAN node and a network of location sensors. These indoor location systems provide the position of tagged entities in a building; the Active Badge and 3D-iD [15] systems are representative examples.

Several research efforts have already tried to provide a solution to the second issue. However, they are either constrained to enable migration of components written in a specific portable programming language such as Java [2, 3] or Python, or present a partial solution, only enabling some functionality of the components, such as the GUI [12], to move. This paper describes LocALE, a framework for the management of heterogeneous (multi-purpose and multi-lingual), objects' lifecycle that provides a more general solution to this problem.

2. Component Mobility in CORBA

The OMG's CORBA (Common Object Request Broker Architecture) standard [10] defines a framework for developing object-oriented distributed applications. The Object Request Broker (ORB) is the middleware that establishes client-server relationships between objects. Using an ORB, a client can transparently invoke remote methods on a server object, regardless of where the object is located, the language in which it is programmed, its underlying operating system, or any other system aspects that are not part of an object interface. These heterogeneity features make CORBA ideal for the implementation of multi-purpose platform-independent components. For this reason the LocALE framework is CORBA-based.

The CORBA object model relies heavily on the semantics of object references. The standard object reference format, called the Interoperable Object Reference (IOR) [10], contains an identifier of the most derived type of an object and one or several *profiles* that permit the location of an object to be determined. Among other data, these profiles may contain an IP address, a TCP port number, and an object key that uniquely identifies the target object. Clients can only invoke an operation on an object if they hold a reference to the object. Unfortunately, IORs' location-specific information, i.e. the IP address and port number of the server at which the object resides, means that when a CORBA server implementation moves, the client(s) can no longer maintain the bind to the server.

Adding mobility support to CORBA supposes that any client holding a reference to an object will maintain that binding for the entire lifetime of the object. This means that client invocations must be forwarded seamlessly when the location of the targeted object changes. CORBA handles this situation by defining within GIOP (General Inter-ORB Protocol) [10] the `LOCATION_FORWARD` reply message. This message indicates to the receiving client ORB that it must retry a previously issued request using the enclosed IOR.

3. LocALE Framework

The LocALE framework proposes a simple mechanism for managing the lifecycle of distributed CORBA objects residing in a ubiquitous computing network. The emphasis of LocALE's design is placed on providing a suitable interface for third party *object-location controllers*. These controllers, aware of personnel location and computing resources location, load and capabilities, can intelligently direct components' locations and lifecycles. LocALE's goal is to offer the object-lifecycle handling infrastructure required. It addresses the following functional requirements:

- *Permit the remote instantiation of CORBA services wherever wanted*.
  Traditionally, in a distributed environment, the factory design pattern is used to avoid having to start servers manually every time a client requires a service. A factory is an object that offers one or more operations to create other objects. Upon client invocation, the factory creates a new object and returns a reference to the client. Nevertheless, the problem with conventional factories is that clients are, normally, limited to creating new objects within the address space of previously started factories. It is desirable for a client to be able to control the location (host) where a new service is instantiated, independently of whether that location already contains a factory capable of creating objects of the desired kind or not.
- *Enable heterogeneous object migration*.
  Computational objects (part of an application) may have to migrate to optimise speed and latency, or simply to get physically closer to a user. Communication endpoints may consequently change location over time. An infrastructure providing transparent object migration to clients without affecting the references they hold is required.

3.1. LocALE Migration Support Rationale

This section presents an overview of the design choices that LocALE has adopted to provide migration support for CORBA components.

Migration of software objects must occur in two phases: (1) transfer state and, optionally, implementation of a running object to a new location and (2) make all clients of the object transparently use the new object. With regard to the first aspect, a migratory system must decide whether the system should support (a) code and state migration, or (b) simply state migration. The use of portable interpreted programming languages such as Java or Python with built-in object serialisation and bytecode portability features facilitates object state and code migration. However, LocALE aims to control the lifecycle of heterogeneous objects, not constrained to a single programming language or platform. Thus, LocALE does not support object implementation transfer upon object migration. Our view is that objects should be able to migrate to different implementations of the same functionality. For example, an object implemented in C++ running on a Windows NT host should be able to migrate to a Java version of it running on Linux. This signifies that wherever an object moves there must be access somehow, e.g. locally or through a shared file system, to a valid object implementation.

There are two basic methods by which state transfer may be performed. One option is to take a snapshot of a running object and transfer that state. Java's object serialisation permits this to some extent. The main problem of this approach is that it prevents objects moving to different implementations. A second option is for objects to transfer their own state when they are instructed to migrate. One or several methods of a newly created object are used to set its state, according to the state of the source object. LocALE adopts this approach as it suits heterogeneous object migration and allows a more general interpretation of what can be considered to be 'state'.

The second issue to be addressed when designing an object migration system is how clients of an object can track the object as it moves. CORBA provides support for this by the GIOP's `LOCATION_FORWARD` message previously mentioned. There are two obvious ways in which this mechanism can be used to track object locations: (1) chaining of forwarding agents and (2) use of a home location forwarding agent.

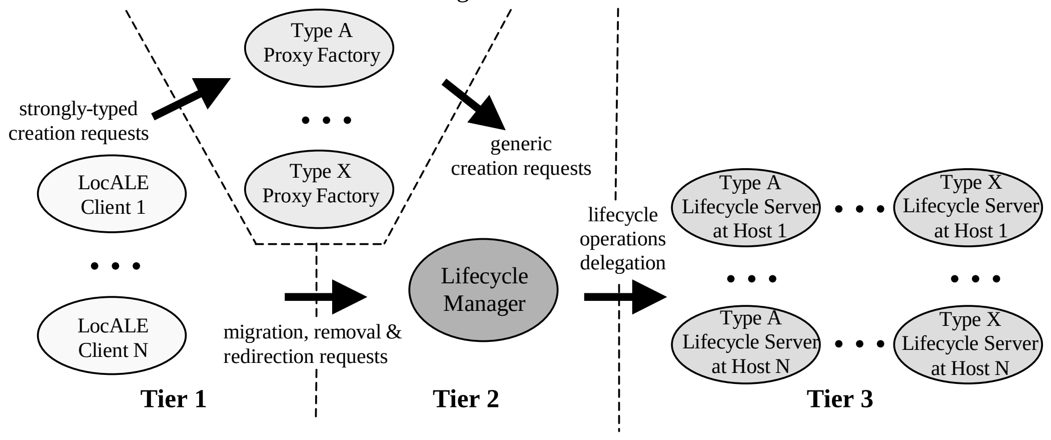
With the first approach, every time an object migrates, it leaves behind an agent, which forwards clients to the object's new location. The result is a trail of forwarding agents at every location the object has ever occupied. This brings about intolerance to faults, and difficulties with garbage collection.

The second approach nominates for each object a *home location* that is guaranteed to know the current location of the object. The home location either provides the object itself, or forwards the client to it. Clients remember the home location, and consult it whenever the object ceases to exist in the location they were using. Whenever an object migrates, it informs the home location of its new location. This is the object tracking mechanism employed with LocALE since it introduces less latency and better resource cleanup properties than the other approach, although it still presents a single point of failure problem.

3.2. LocALE Architecture

The design of LocALE presents a 3½-tier architecture, shown in Figure 1, composed of client applications and the following 3 types of components:

- *The Lifecycle Manager* (LCManager).
  This component provides generic lifecycle control interfaces and mediates the lifecycle operations over any CORBA object residing in a location domain. In this context the term *location domain* refers to a group of machines on a LAN that are located within a given physical area, such as a building, a floor or a room. Every object creation, movement or deletion request is routed through this object. This permits the LCManager to cache the current location of every object in a domain and thus act as a *forwarding agent* that redirects client requests after object references held by clients are broken due to either object movement or failure. In the latter case, the LCManager first tries to recover the failed object, and, in case of success, returns the new object reference.
- *Lifecycle Server*(s) (LCServer).
  These components act as surrogates of their local domain LCManager. Lifecycle operations invoked on an LCManager are delegated to a suitable LCServer. These servers contain, within their address space, LocALE-enabled objects subject to lifecycle control. LCServers subsume standard strongly typed factories' functionality by not only providing a type specific creation method with hard-coded types, but also assisting their local objects with their migration to other LCServers. They can be started either manually or by the local LCManager after a creation or migration request arrives at a location where the required LCServer type does not exist. In either case, they register with the manager and pass metadata containing their physical location (host), IOR and information specific to the type of objects they handle.
- *Type-specific Proxy Factories*.
  These components are placed in between clients and LCManagers. Their purpose is to prevent client applications from dealing with the generic object creation interface offered by an LCManager. Using specific-purpose interfaces makes client code far shorter and simpler to understand. With the generic version if a mismatch in the type of a constructor argument occurred, the client would only receive an error at run-time, whilst with the specific purpose



Type A Proxy Factory
strongly-typed creation requests
• • •
generic creation requests
Type X Proxy Factory
LocALE Client 1
• • •
LocALE Client N
migration, removal & redirection requests
Lifecycle Manager
lifecycle operations delegation
Type A Lifecycle Server at Host 1
Type X Lifecycle Server at Host 1
Type A Lifecycle Server at Host N
Type X Lifecycle Server at Host N
Tier 1
Tier 2
Tier 3

Figure 1. LocALE 3½-tier architecture

version it would be at compile time. Type-specific Proxy Factories offer type specific object-creation methods with the same argument types and semantics as the LCServers they represent, mapping type specific requests to the generic format demanded by LCManagers. They are activated either manually or automatically by the local manager, after which they register, passing their IOR and type details.

LocALE's architecture guarantees clients' compile-time type safety with respect to the lifecycle control of their required services. Clients find, through the LCManager, object references to type-specific proxy factories and issue through them object creation requests. Object migration and deletion requests are directly invoked on the LCManager because they are independent of object types. The manager then delegates incoming lifecycle operations to the appropriate LCServers.

3.3. Location-constrained Lifecycle Control

One of the main advantages of CORBA is that it provides location transparency to applications, i.e. it makes method invocation as simple on remote objects as on local objects. LocALE leverages CORBA's location transparency, but it asserts that being able to control *where* services used by clients are located, and also to set the constraints under which those services can later be re-located may bring important benefits. For example, load balanced applications may wish to initiate new service instances in the hosts of a LAN with lowest processing load. Follow-me applications may want to move objects tied to a user's physical location to the nearest host with the required capabilities. To provide CORBA services' *location-awareness* to clients, LocALE changes distributed object construction semantics compared with conventional factory objects, in two ways:

1. It enforces client code not only to specify the arguments of the constructor of an object, but also the *location* where the object should be created and some *constraints* to control its life evolution.
2. Object creation requests are always routed through the local domain's LCManager that processes them and finds or creates suitable LCServers where such requests are delegated. Thus, the manager can cache the object references returned by surrogate LCServers and convey to clients specially manufactured IORs tied to objects for their entire lifetime.

The following two attributes are appended to the type specific creation interface provided by proxy factories and the generic interface offered by LCManagers:

- *Location* attribute: specifies *"where"* a service should be instantiated (or moved to). Its possible formats are shown in Table 1.
- *LifeCycle contraints* (`LCconstraints`) attribute: determines whether an object to be created should be considered as recoverable and/or movable within a location scope. A recoverable object is either a stateless or a persistent object whose state can be restored after failure. Table 2 lists all existing `LCcontraints` and their usage dependencies with `Location` attribute specifications.

Location independent services are instantiated using `hostDN("ANY")` or `roomID("ANY")` constructions. These specifications are of special interest when there are same type Lifecycle Servers running on separate hosts of a location domain. Upon object creation or movement, as a crude form of load balancing, the LCManager randomises the list of available Lifecycle Servers to select one to which to delegate the request. Certain services, however, need to be created in the same location as a user. For instance, an object with a user interface should be created either in the host closest to a mobile user, e.g. `hostDN("guinness")`, or otherwise in any host within the same room, e.g. `roomID("Room1")`. Finally, some services should be created in any machine within a given location that provides a special resource (e.g. sound, video capture, etc.). The construction `hostGroup` is used in this case. It is up to the client application or other services to select a suitable list of hosts.

Table 1. Location attribute specs

`hostDN("guinness.eng.cam.ac.uk")`
`hostDN("ANY")`
`roomID("Room 1")`
`roomID("ANY")`
`hostGroup(hostDN_1, …, hostDN_n)`

LocALE assumes that objects are, by default, static and non-recoverable. Once an object is created, it remains in the same address space until it is removed or fails. However, LocALE provides mechanisms to make objects `MOVABLE`, providing they know how to transfer their own state, and/or `RECOVERABLE`, providing they can restore their

Table 2. Lifecycle constraints and usage rules

RECOVERABLE	∀ Location specs
RECOVERABLE_WITHIN_ROOM	iff (Location = roomID(x) ∨ Location = hostDN(x)) ∧ ¬(x="ANY")
RECOVERABLE_WITHIN_HOST	iff (Location = hostDN(x)) ∧ ¬(x="ANY")
MOVABLE	∀ Location specs
MOVABLE_WITHIN_ROOM	iff (Location = roomID(X) ∨ Location = hostDN(x)) ∧ ¬(x="ANY")

state after failure. The LCManager recreates failed objects using the same constructor argument values as when the object was first constructed. An object can be simultaneously movable and recoverable, as long as both constraints apply to the same location scope, i.e. `MOVABLE & RECOVERABLE` is valid but not `MOVABLE_WITHIN_ROOM & RECOVERABLE`.

The capability of receiving location constraints on object creation converts the LCManager into a minimal Trader [8] server. The LCManager matches client object creation specifications against the advertised object Lifecycle Server types and locations. On a service instantiation, its type, initial location and life evolution constraints are conveyed to the local LCManager. This component searches among the registered Lifecycle Server types for a suitable LCServer where the requested lifecycle operation can be delegated. LCManager's trading capabilities could be extended by enforcing LCServers to specify the computing resources demanded by their objects (e.g. need for sound capability). Thus, if a client issued a lifecycle operation over a given object type the LCManager would match both location constraints and object type specific resource demands. This would make the use of the `hostGroup` location format unnecessary.

3.4. LocALE Lifecycle Manager

The LCManager keeps, in an internal registry (see Table 3), information about the objects whose lifecycle it controls, the proxy factories used for strongly typed object creation and the LCServers where object creation, migration and removal actually takes place. This information, serialised to disk to permit the LCManager component to recover from transient failures, is used to provide the following functionality:

1. Load-balanced object lifecycle control.
2. Object reference forwarding to clients on method invocation to objects whose location has changed.
3. Automatic activation of LCServers when demanded at a location where they are not available.
4. Deactivation of automatically started LCServers.
5. Activation of type-specific proxy factories when there is no instance running in a location domain.
6. Deactivation of automatically started proxy factories.
7. Fault tolerance of stateless or persistent LocALE objects by recreating them upon a failed client request.

Swizzling Mechanism for Object Reference Management

The LCManager *swizzles* object references returned by LCServer creation methods into object references pointing to it. The actual object reference is cached in the `Object` table (see Table 3). The swizzled IORs returned to clients are tied to their corresponding objects for their entire lifetime no matter how many times they migrate or are recovered. As result of this procedure, every first request invocation on an object reference is not addressed to the intended object but to the LCManager. The manager then forwards the client the actual object reference through a `LOCATION_FORWARD` reply. The client ORB transparently retries the operation on the new object reference, and will use this reference while it remains valid. When an object moves or fails, a client request will fail and its ORB will fall back to the original swizzled reference. In that case, the LCManager either returns a moved object new reference, tries to recover a failed object and return its reference, or, otherwise, raises an error exception.

The cost of this mechanism is that client requests are transparently delayed due to the additional round-trips introduced after a lifecycle or failure event occurs. The advantage is that the LCManager presents object fault-tolerance and mobility support facilities without breaking client held references.

Internally, the LCManager's C++ implementation makes use of two POA[1] (Portable Object Adapter) [10] components, one containing the servant implementing the remote interfaces offered by this server and the other one, with a Servant Manager [13] object registered with it, that implements the client request redirection procedure described above. Client requests on swizzled IORs (created using the second POA) arrive at this POA which up-calls a method of its associated Servant Manager implementing the swizzling mechanism. This method uses a POA defined standard mechanism to generate `LOCATION_FORWARD` reply messages, namely the `ForwardRequest` exception.

Table 3. Lifecycle Manager Internal State

LCServer(<u>LCServID</u>, *kindID*, *hostDN*, LCServRef, instanceCounter, maxNumObjs, manuallyInit)
Object(<u>objID</u>, *kindID*, *LCServID*, objRef, locationSpec, paramValues, LCconstraints)
ProxyFactory(<u>proxyFactID</u>, proxyFactRef, manuallyInit)
HostLCServers(<u>hostDN</u>, set<*LCServID*>, *roomID*, OS)
RoomHosts(<u>roomID</u>, set<*hostDN*>)
KindLCServers(<u>kindID</u>, set<*LCServID*>, objCreationMethod, paramTypes, start-up-process)
KindProxyFactories(<u>kindID</u>, set<*proxyFactID*>, start-up-process)

3.5. LocALE Lifecycle Servers

Every Lifecycle Server defines a method to create objects of a specific type and, in addition, implements the

[1] A POA manages CORBA objects within a process and allows applications control the mapping of CORBA objects to servants (a programming language entity incarnating a CORBA object).

LCServer interface (see Figure 2). The LCManager uses the move_object method in this interface to trigger the migration of an object under the control of an LCServer to a clone of that object in another LCServer. Every object managed by a Lifecycle Server is derived, in turn, from the LocALE_Object (see Figure 2) interface. Upon a call to its transfer_state method, an object conveys its state to the given clone. Object type specific method(s) are invoked to adapt the state of the target object to the source one.

```
interface LCServer {
 void move_object(in LocALE_Object objRef,
                  in LocALE_Object cloneObjRef)
  raises(UnKnownObj, WrongObjType, InvalidClone);
 void destroy(); // Method to destroy LCServer
};

interface LocALE_Object {
 void destroy();
 void transfer_state(in LocALE_Object cloneRef)
             raises (InvalidClone);
 readonly attribute HostDN currentLocation;
};
```

Figure 2. LocALE LCServer and object interfaces

In order to support mobility, objects created within an LCServer must be activated simultaneously with two POAs (see Figure 3). The RootPOA (or any other POA) through which requests from external clients arrive to the LCServer's servant and servants of the objects whose lifecycle it controls and a second POA (TransferStatePOA) through which state_transfer operations on the controlled objects are issued by the LCServer. The reason being that on object migration, while the LCServer triggers the state transfer between the source and target object and waits for its completion, it must guarantee that requests on the migrating object are queued for later processing. Every POA contains an

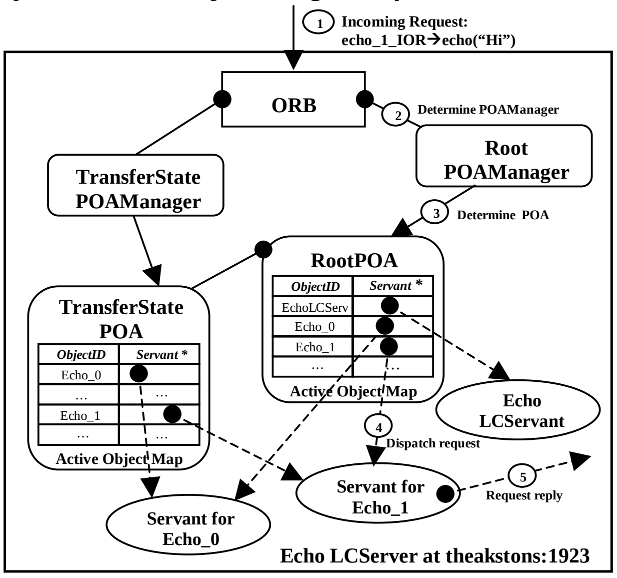


Figure 3. Request dispatching in an LCServer

associated POA Manager [13] object that controls the flow of requests into it. Thus, client requests on the migrating object are blocked by calling the hold_requests operation on the RootPOAManager. The LCServer uses a second IOR for the object, obtained from its TransferStatePOA, to invoke the state_transfer operation. Once the object migration is finished, the RootPOAManager is reactivated. Queued invocations on that POA Manager then result in OBJECT_NOT_EXIST exceptions. Client ORBs react to these exceptions by re-routing the requests, assisted by LCManager's redirection capabilities, to the new object location.

4. LocALE Object Lifecycle Management

This section provides some C++ code illustrating how clients can issue lifecycle control operations (i.e. remote construction, movement and removal) on an object using LocALE's API. Figure 4 shows the interactions involved within LocALE's architecture upon object migration.

```
// Create a new recoverable and movable echo
// object in host dogbolter and invoke echo()
LocALE_ProxyFactory_var pFact = LCMan->
  find_proxy_factory("IDL:Echo/EchoObject:1.0");
Echo::EchoProxyFactory_var echoPFact =
  Echo::EchoProxyFactory::_narrow(pFact);
Echo::EchoObject_var echo1Ref = echoPFact->
  create_echo_object("Echo_1",
            LOCATION(HOST("dogbolter")),
            LC_CONSTRAINT(RECREATABLE¦MOVABLE));
echo1Ref->echo("Hi");

// Move previously created object to host
// theakstons and invoke echo() on it
LCMan->move_object(echo1Ref,
                   LOCATION(HOST("theakstons")));
echo1Ref->echo("Hi again");

// Remove the echo object moved to theakstons
LCMan->remove_object(echo1Ref);
```

5. Applications

This section describes two LocALE-enabled applications built to demonstrate LocALE's potential as an enabling infrastructure for follow-me applications. These applications are composed of multiple distributed objects. When a user moves, only those objects tied to the user's physical location follow him. All inter-object communication is undertaken through CORBA RPCs.

5.1. Follow-me Music

The largest application constructed with this system supplies mobile users with music from the nearest set of speakers, wherever they move in our lab. The source of music is a static MP3 jukebox server that is controlled using the buttons of an Active Badge. A personal software agent, associated with each lab person, listens for that person's movement events coming from an Active Badge Location server [14]. This agent, acting as a component migration controller, moves (using LocALE migration capabilities) the audio player and MP3 decoder components of the application to the host nearest to the user supplying the appropriate facilities. As the user moves around the location-aware environment, the music is played back continuously from the nearest set of speakers. The state of the system and time index into the current song persists as the components migrate.

5.2. Follow-me Email Notification

Another personal agent has been built that acts as an IMAP mail client and an event sink for location events coming from an Active Badge Location server. This agent caches the last location where a user was seen, and when an email for that person is received the agent remotely instantiates, through LocALE, a text-to-speech object at the host (with sound capabilities) closest to the user. A verbal notification with the sender and subject details of the message is then produced from the nearest speaker. The body of the email message may also be read on the user's request, by clicking one of the Active Badge buttons. After a timeout expires, the agent removes, assisted by LocALE, the text-to-speech object.

6. Related Work

The CORBA Object Life Cycle Service [9] defines conventions that allow clients to perform lifecycle operations on objects in different locations. It provides IDL interfaces that permit clients to control the lifecycle of objects without knowing about their types or semantics. This makes it suitable only for applications that require a weak type model. LocALE subsumes most of this service functionality, still guaranteeing client code compile-time type safety through its proposed 3½-tier architecture.

Jumping Beans [1] is a toolkit that lets CORBA server objects move transparently to the clients. The approach is based on a centralised Jumping Beans server. When a CORBA server object moves, it first moves to the Jumping Beans server and then to its final destination. This project resembles LocALE in the use of a central entity that controls object migration. In contrast to LocALE, it does not address heterogeneous component migration. It only provides mobility support for Java implemented objects but can hence support implementation code transfer. LocALE differs from Jumping Beans in its capability to perform location constrained object lifecycle control.

DAWS [4] proposes a decentralised heterogeneous component migration framework where, when an object migrates from the location where it was created (*home location*) to another location, it leaves behind a *forwarding agent*, which emits `LOCATION_FORWARD` messages pointing to the new location. This design presents good scalability properties but makes object resource cleanup more difficult than with LocALE's centralised model. DAWS is ORB-dependent, in contrast to LocALE, because its implementation modified a CORBA 2.0 ORB lacking POA support. DAWS does not support the load-balanced remote service instantiation, Lifecycle Server automatic activation and deactivation, and object recoverability features owed to LocALE's centralised design.

A LocALE LCManager is *per se* a CORBA Implementation Repository (IMR) [6], because it provides facilities to (1) track CORBA objects' current location, (2) forward a client request for an object to the correct process, and (3) automatically activate servers on demand. But in addition, LCManagers also control object lifecycle operations. Existing IMRs only cache location and activation information of server object adapters (POAs)

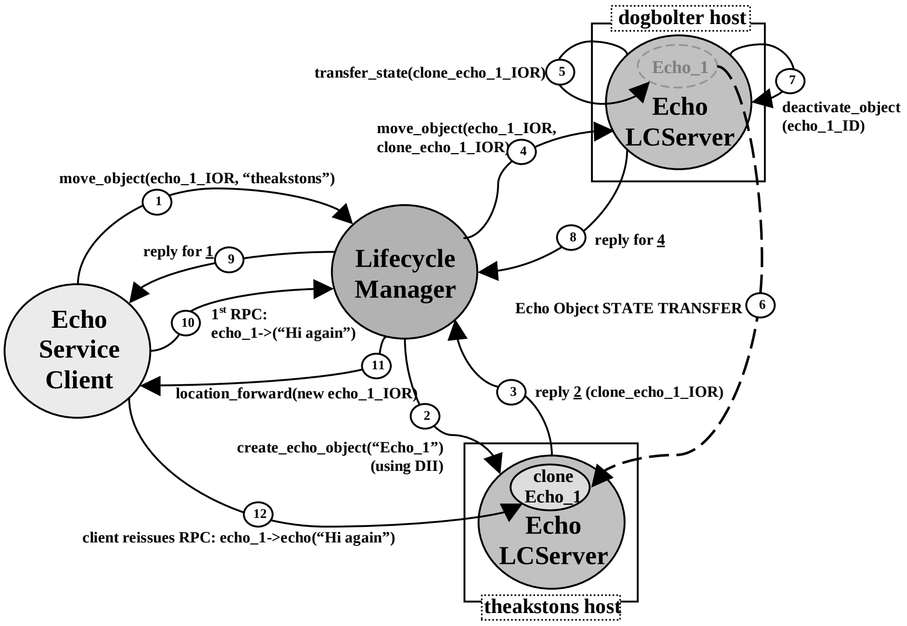

dogbolter host
Echo_1
Echo LCServer
transfer_state(clone_echo_1_IOR) 5
7 deactivate_object (echo_1_ID)
move_object(echo_1_IOR, clone_echo_1_IOR) 4
move_object(echo_1_IOR, "theakstons") 1
Lifecycle Manager
8 reply for 4
reply for 1 9
Echo Object STATE TRANSFER 6
Echo Service Client
10 1st RPC: echo_1->("Hi again")
11 location_forward(new echo_1_IOR)
3 reply 2 (clone_echo_1_IOR)
2 create_echo_object("Echo_1") (using DII)
clone Echo_1
Echo LCServer
12 client reissues RPC: echo_1->echo("Hi again")
theakstons host

Figure 4. Object Migration in LocALE

where objects are activated. This offers good scalability because the same adapter information is used to provide indirect binding support to all objects contained within it. Nevertheless, it only enables migration, at once, of all objects registered with a particular object adapter. IMRs are ORB-specific because they can only manufacture forwarding IORs for objects created with their same ORB.

Mobile Agent Systems, such as Voyager [3] or Mole [2], are focused on enabling the transparent migration of agents, including their code, data and threads, from place to place. Mobile agents are autonomous and detached from clients, whereas LocALE clients direct objects' lifecycle. LocALE is designed to be used within a LAN; mobile agents are typically used on the Internet where security and reliability are much more important.

7. Conclusion and Future Work

This paper has described the design of LocALE, a framework for the control of CORBA objects' lifecycle in a fixed ubiquitous network. LocALE permits client applications to constrain the physical location of newly created objects, and to define the location restrictions under which these may later be migrated or recovered. Object lifecycle requests issued by clients are routed through a central Lifecycle Manager that, after matching objects location and lifecycle constraints, delegates requests to suitable LCServers. Likewise, the LCManager redirects client requests after an object's location changes. LocALE has shown its capability as an enabling infrastructure for location-aware mobile computing by the successful implementation of two follow-me applications. Future work will address its applicability in the development of load-balanced distributed systems.

In due course work on LocALE will tackle the two main limitations of its design: (1) the Lifecycle Manager constitutes a single point of failure and bottleneck within a location domain and (2) an undesirable overhead is incurred on the first RPC over a LocALE object.

The first problem may be addressed by replicating the LCManager and using, for instance, group communication mechanisms to synchronise their internal state. Multi-profile IORs embedding all object references of the federated LCManagers could then be manufactured upon object creation. Thus, clients would transparently reissue a request on the object reference of another replicated server when one member of the federation failed [11]. Likewise, the load balancing features of the system could also be improved by selecting, in a random fashion, one of the multiple LCManager IORs available.

The overhead incurred on the first RPC over an object (due to LocALE's IOR swizzling mechanism) may be solved by including the actual object reference as the primary profile within a multi-profile IOR [11].

Acknowledgements

The authors are grateful to Tim Edmonds, Rob Headon, David Riddoch and Jamie Walch for their advice and suggestions on this work. Thanks are also due to the Basque Government Department of Education and AT&T Laboratories Cambridge for their sponsorship of the authors.

References

[1] Ad Astra Engineering. "Jumping Beans White Paper". October 1999, http://www.JumpingBeans.com

[2] Baumann J., Hohl F., Rothermel K., Schwehm M., and Straßer M. "Mole 3.0: A Middleware for Java-Based Mobile Software Agents", Proceedings of Middleware'98, September 1998, pp. 355-370

[3] Glass G. "ObjectSpace Voyager Core Package Technical Overview", ObjectSpace, White Paper, 1999

[4] Grisby D. P. "A Distributed Adaptive Window System", PhD thesis, Computer Laboratory, University of Cambridge, 1999

[5] Harter A., Hopper A, Steggles P., Ward A. and Webster P. "The Anatomy of Context-Aware Application", Proceedings of MOBICOM'99, Seattle, August 1999

[6] Henning M. "Binding, migration and scalability in CORBA", Communications of the ACM, vol.41, no.10, October 1998, pp. 62-71

[7] Hopper. A. "Sentient Computing", The Royal Society Clifford Paterson Lecture, ", AT&T Laboratories Cambridge, Technical Report 1999.12, 1999

[8] Object Management Group, "Trading Object Service Specification", May 2000

[9] Object Management Group, "Life Cycle Service Specification", June 2000

[10] Object Management Group, "The Common Object Request Broker Architecture: Architecture and Specification", October 1999

[11] Object Management Group, "Fault Tolerant CORBA", Revised Joint Fault Tolerance Submission, December 1999

[12] Richardson T, Stafford-Fraser Q., Wood K.R. and Hopper A. "Virtual Network Computing", IEEE Internet Computing, Jan/Feb 1998, vol.2, no.1, pp 33-38

[13] Schmidt D. and Vinoski S. "C++ Servant Managers for the Portable Object Adapter", SIGS C++ Report, September 1998

[14] Want R., Hopper A., Falcão A. and Gibbons J. "The Active Badge Location System", ACM Transactions on Information Systems, January 1992, vol.10, no.1, pp. 91-102

[15] Werb J. and Lanzl C. "Designing a positioning system for finding things and people indoors", IEEE Spectrum, September 1998, pp.71-78