Distributed Programming Patterns
Workshop, OT2000, Oxford

Session Organiser

- Kevlin Henney, Curbralan Limited, kevlin@curbralan.com

Session Participants

- John Wilshaw, ECSoft, john.wilshaw@ecsoft.co.uk
- Sven Erik Knop, Versant, sven.knop@versant.com
- Alan Wills, Trireme, alan@trireme.com
- John Daniels, Syntropy, john@syntropy.co.uk
- David Leibs, Neometron, leibs@neometron.com
- Leejanne Meijboom, Vitalogic, l.meijboom@vitalogic.nl
- Annwen Owen, UCNW, Annwena@yahoo.com
- Robin Harwood, Independent, rharwood@bcs.org.uk
- Benedict Heal, Independent, bheal@cix.co.uk
- John Nolan, Connextra, john@connextra.com
- Ann Corduroy, Aspect, ann.corduroy@aspect.com
- Steve Freeman, M3P, steve@m3p.co.uk
- Tung Mac, Connextra, tung@connextra.com
- David Vines, IBM, dvines@uk.ibm.com
- Monique Limbos, CE, Monique.Limbos@cee.eu.int (bounces)
- Kirby McInnis, Castek, kmcinnis@castek.com
- Immo Huneke, Logica, HunekeI@logica.com
- Ewan Milne, emilne@lucent.com

Introduction

Motivation

Development in a distributed environment introduces a number of challenges not present in sequential single process programming. The objectives of this workshop were to mine patterns from successful practices for programming distributed systems, and link them together.

Design

Design can be defined as a creational and intentional act: the conception and construction of a structure on purpose, for a purpose. In this, it must take into account the context: solution structure is sensitive to details of purpose and context, and context can challenge and invalidate assumptions. As such, context-free design is meaningless.

Distribution presents a context for design:

- Operation invocations are no longer trivial: communication can dominate computation, and partial failure is almost inevitable.
- Concurrency is implicit.

In turn, concurrency itself presents a context for design:

- Synchronisation is required to ensure consistent and coherent state.
- Property style programming is inappropriate, e.g. MIDL properties, OMG IDL attributes, *set* and *get* operation pairs.

Distributed deployment is a further context for consideration:

- Heterogeneous platforms are the norm rather than the exception.
- Absence of static linking raises versioning issues.
- Simple naming may not uniquely identify components.

Patterns

A pattern documents a reusable solution to a problem within a given context, capturing the facets defining the design space:

- *Context* defines the design situation giving rise to the problem.
- *Conflicting forces* are the issues that must be taken into account in arriving at the solution.
- *Configuration* defines the elements of the solution that balance the forces.

The pattern form used in the workshop varied a little from group to group, but was based around the following form:

Name

- The memorable name given to the pattern

Problem

- A short description of the problem being addressed by the pattern

Solution

- A short description of the solution recommended by the pattern

Example

- An example demonstrating the pattern in action

Forces

- The circumstances that make this pattern relevant and the principles that lie behind the solution

Consequences

- The resulting context of applying the pattern

Relationships

- How the pattern relates to or interacts with others

Relationships between patterns can take many forms:

- Patterns may have other names, e.g. SURROGATE and PROXY.
- Patterns may be variations or more specific versions of other patterns, e.g. SMART POINTER and PROXY.
- Use of one pattern may lead to the application of another, e.g. LAYERS and PROXY.

A pattern language connects patterns by application, so that the solution structure of one pattern may be described in terms of other patterns and the consequences of one pattern may generate forces that must be resolved by another.

Workshop

The hands-on element of the workshop was 2½ hours long and participants were divided into groups of 4 or 5.

Following an initial brainstorming session to capture names and descriptions of practices found useful in the development of distributed systems, an initial crop of proto-patterns was harvested based on discarding well-known patterns, duplicate/synonymous concepts, and uninteresting ones!

Problem/solution thumbnails were then documented to add some substance behind the names, followed by identification of consequences and forces. Participants moved between groups and established relationships between proto-patterns they had already documented or seen and others.

The Patterns

The output from the workshop, in the form of posters for each pattern and documented relationships, has been captured as-is in this document. Above basic reformatting, no interpretation has been applied to them and the patterns are arranged loosely by group in the order found at the end of the session.

If you have any comments or suggestions for refinement please feel free to get in touch, so that the patterns can evolve and benefit from reflection and the experience of others.

STORE AND FORWARD

Problem

- Point to point communication is not guaranteed.

Solution

- Store the payload at each node, until it can be sent to the next node.

Consequences

Pros

- Resilience.
- Flexible.
- More manageable message routing (network traffic balancing).

Cons

- Unpredictable delivery.
- Resources required for delivery.
- More points of failure (harder to track a problem).

Relationships

- Uses MIRROR RESULTS for reliability.
- Similar to BATCH MESSAGES.
- Used in implementing the relationship between PUBLISH and SUBSCRIBE.

MIRROR RESULTS

Problem

- Need to maintain state when components fail.

Solution

- Duplicate the state of the running system so you can recover immediately.

Consequences

Pros

- Reliable.
- Limited impact on application design.

Cons

- Expensive.
- Hard to get right (particularly with fail over).

Relationships

- Used by HOT STANDBY (undocumented), which incorporates it.
- Grouped with HOT STANDBY.

SUBSCRIBE

Problem

- The client wants particular information but doesn't want to know where it comes from.

Solution

- The client subscribes to an information source.

Consequences

Pros

- Decoupling.
- Runtime flexibility.
- Resilience.
- Multiple sources easier.

Cons

- Another query language (for subscription).
- Client is not aware of server problems (can be good, i.e. a pro).
- Complex to administer.

Relationships

- Used with PUBLISH.

PUBLISH

Problem

- Servers need to notify multiple clients, which takes a lot of processing.

Solution

- The server sends it once to a bus, which notifies all the clients.

Consequences

Pros

- Reduced bandwidth requirements.
- Decoupling server and clients.
- Simplifies application design.

Cons

- Need more components (message bus) (3rd party, too?).
- Need to learn new stuff (e.g. Tibco) (OTOH, *CV++* seen to be a pro).
- Harder to administer.

Relationships

- Used with SUBSCRIBE.

DO IT ONCE

Problem

- Distributed systems are hard, complicated and expensive. Networks are getting better.

Solution

- Put all your processing in a single well-supported place.

Consequences

Pros

- Simpler.
- Well-understood, easier to construct.
- Easier to manage.

Cons

- Single point of failure.
- Speed of light.
- Harder to customise by location (organisational).

Relationships

- On its own.

BATCH MESSAGES

Problem

- Lots of little requests over the wire will be very slow.

Solution

- Group your requests and send them together.

Consequences

Pros

- Better bandwidth use.
- Easier to optimise.
- More processor efficient.

Cons

- More complex.
- Non-deterministic delivery times.
- Must design messaging to be able to be batched.

Relationships

- Similar to STORE AND FORWARD.
- Used in implementing the relationship between PUBLISH and SUBSCRIBE.

DOCUMENT EXCHANGE

Problem

- We want to support a business process that requires information to flow between different "scope of control".

Forces

- Lack of a single implementation standard.

Solution

- Package information into a "document" which is passed between scopes using a trivial protocol.

Consequences

- The scopes must agree on the structure of the document.
- Easier to implement the interchange protocol.
- Requirement for shared semantics.

Relationships

- Can use PUBLISH/SUBSCRIBE.
- Is a mega-FACADE.

PUBLISH/SUBSCRIBE

Problem

- Eliminate explicit binding between the producer of information and the consumer(s).

Forces

- New consumers without affecting the producer.

Solution

- Name the information items and use a third party to hold, manage and implement the coupling.

Consequences

- Possible introduction of single point of failure.
- Publisher doesn't know about subscribers.
- "Write-Only" publishing.
- Decisions about granularity of publishing.
- Lots of issues to be resolved (e.g. guaranteed delivery).

Relationships

- Builds on MESSAGE QUEUE.
- Is a version of OBSERVER.
- Can use HITCHHIKER.
- Can use BUS SERVICE.

GATEWAY

Problem

- Need to incorporate legacy systems in a distributed environment.

Forces

- COBOL's unwillingness to go away

Solution

- Provide a gateway that acts as an adapter layer.

Consequences

- Mapping decisions become localised.
- Load overhead of "extra" layer.

Relationships

- Specialises ADAPTER
- May adapt SYNCH/ASYNCH

SESSIONS

Problem

- Need to keep context of "user" interaction but not pass all state on every call.

Forces

- Desire to reduce amount of information on "the wire".

Solution

- Keep state at the server and implement a mechanism to allow client to identify and use session context, e.g. alarm concentrator.

Consequences

- Waste server resources if housekeeping is poor.
- Possible and undesirable mingling of session and "business" state.
- Makes load balancing harder.
- Network traffic is reduced.
- Now need to be aware of ordering constraints at the server.

Relationships

- Surround by undesirability field.
- BUS SERVICE tends to lead to SESSIONS.
- HITCHHIKER tends to lead to SESSIONS.

HITCHHIKER

Problem

- There is a need to reduce communication overheads for remote method calls.

Forces

- Mix of "urgent" and non-urgent communication.
- Network load and performance.
- Message ordering not important.

Solution

- For non-urgent remote invocations can register with a "hitchhiking agency".

Consequences

- Management overhead.
- API complexity increases.
- Design decision: queue message for non-urgent delivery or register a callback.

Relationships

- Often leads to SAFETY DEPOSIT BOX.
- Alternative to BUS SERVICE.

TRANSACTION TIMEOUT

Problem

- When I invoke a remote operation, how can I be certain about the server's state?

Forces

- Don't want to use heavyweight transaction locking.
- Need to use unidirectional messaging.
- System designer needs to be able to specify a policy for recovery from client failure during transaction.

Solution

- Within a known time interval of the invocation, the server takes a previously agreed default action.

Consequences

- Time interval during which server state unknown.
- Forces client to take action before timeout if the default action by server is not desired.

Relationships

- Simplified by BIRDS OF A FEATHER.

OBJECT MIGRATION

Problem

- An object needs to invoke many operations on remote services.

Forces

- Efficiency of method invocation.
- Efficiency of object transfer.
- Easier load balancing.

Solution

- It may be more efficient to relocate itself to be "near" the servants.

Consequences

- Needs a mechanism for "atomic" transfer of object state.
- Difficult to do across heterogeneous systems:
  - Virtual Machine (Java, Smalltalk) → version compatibility.
  - Compiled classes already present on target server → version compatibility.
- Tradeoff:
  - Migrate attributes only.
  - Factorise into smaller classes (agents) and migrate just some of them.
- Identity problem – handle before/after "cloning".
- Load management solution needed (manager, query interface).

Relationships

- Simplified/obviated by BIRDS OF A FEATHER.
- Dynamic version of BIRDS OF A FEATHER.

BUS SERVICE

Problem

- Need to reduce communication overhead for remote method calls.

Forces

- Need to maintain message ordering from each client's perspective.
- Network load and performance.
- Most messages non-urgent, if not all.
- Not suitable for synchronous RPC.

Solution

- Group parameters/calls and send together in single message.

Consequences

- Purity of design is obscured – client's view very different from server's.
- Concurrency and precedence between different clients not guaranteed.

Relationships

- Alternative to HITCHHIKER.
- Often leads to SAFETY DEPOSIT BOX.

REMOTE GOLDFISH / LOCAL ELEPHANT

Problem

- Low quality of service.

Forces

- Cost of transmitting state is less than cost of maintaining state in server.
- Client does not rely on being served by the same server every time.
- Load balancing across server farms: dynamic, flexible.

Solution

- Remove reliance on locality of service to facilitate load balancing and resilience by making objects/components context-free and maintaining the context on the client.

Consequences

- No need to maintain session.
- The state must be transmitted with each request (à la HTTP, CICS).
- Reliance on client to preserve state data → security issues ("Do you accept this cookie?").
- Load balancing across server farms: dynamic, flexible.

Relationships

- Alternative to SAFETY DEPOSIT BOX.

META-DATA

Problem

- Object interfaces are prone to change, so we wish to decouple servers from clients to minimise deployment impact of change.

Forces

- Flexibility, poor specs, moving target.
- Need to delay design decisions.

Solution

- Implement flexibility through meta-data, e.g. data dictionary, interface repository, XML, etc.

Consequences

- Increases the workload on the client,
- Increases the workload on the network.
- May be forced to cache meta-data → versioning.

Relationships

- Overhead reduced by BIRDS OF A FEATHER.

SAFETY DEPOSIT BOX

Problem

- The amount of state too expensive to be communicated each request.

Forces

- Thinner client.
- Low bandwidth.
- Short in resources.

Solution

- Maintain state on server and locate it by maintaining session data at client.

Consequences

- Need to recover from client failure (keep alive).
- Client identity problem.
- Need to maintain session.
- Makes load balancing / fail over process more difficult.

Relationship

- Alternative to REMOTE GOLDFISH / LOCAL ELEPHANT.

BIRDS OF A FEATHER

Problem

- Need to reduce usage of expensive communication routes amongst related objects.

Forces

- Network load/performance where different legs have widely differing performance.
- Tightly interacting/coupled objects.
- Adequate capacity to accommodate object population (app server).
- Want to simplify client API.

Solution

- Gather closely related objects/components into "less" distributed subsystems (e.g. the same system!).

Consequences

- Loss of flexibility:
  - At design time.
  - For load balancing.
- Better encapsulation: larger grained classes/components.
- Reduced complexity: single point of contact for clients.

Relationships

- Often leads to META-DATA.
- FACADE is a related pattern.

N-TIER ARCHITECTURE

Problem

- Single point of access (e.g. a mainframe).
- Convoluted access, data and business logic.

Forces

- Maintainability of software is difficult.

Solution

- Introduce multiple points of access.
- Separate out GUI, Application and Database.

Consequences

Pros

- Share data/resources.
- Flexible architecture for adding logic.

Cons

- Increased network traffic.
- Concurrency potential.

Relationships

- Classified as architecture-level pattern.
- Excludes FEDERATION.

FEDERATION

Problem

- Single point of failure.
- Too big for one box.
- Access via a single point for many users.

Forces

- Use if:
  - Lots of existing services that can be coupled.
  - Need for robust, fault tolerant system.
- Don't use if:
  - Systems use wildly different representations.
  - Coupling doesn't add value.

Solution

- Loosely coupled independently useful systems.

Consequences

Pros

- Make subcomponents work independently.
- Reliability of comms is less critical (alternate paths).

Cons

- Difficult to predict performance.
- Have inter-system coupling/dependencies.
- Balance reliability with bandwidth/replication.

Relationships

- Classified as architecture-level pattern.
- Excludes N-TIER ARCHITECTURE.

CAREFUL SYNCHRONISATION

Problem

- Keep two related objects up to date but not occupy too much bandwidth.

Forces

- Two or more values have to be in sync, but bandwidth etc doesn't allow immediate sync (i.e. multiple writes, few reads).
- Disconnected working.

Solution

1. Analyse:
   - How often reads/writes occur.
   - Whether small asynchronies matter (with respect to business problem).
   - When asynchronies matter.
2. Schedule synchronisation when / as often as necessary:
   - Block read access when necessary until sync done.

Consequences

Pros

- Can choose how well synchronised objects are.
- Makes optimal use of resources.

Cons

- Requires planning work.
- May get sync schedule wrong.
- Requires computing cycles.
- Apps aware that values may be de-synchronised.

Relationships

- Classified as object-level pattern.

SESSION TOKEN

Problem

- Server must maintain sessions but communication channel is stateless.

Forces

- To minimise complexity of session management software on the server.

Solution

- Client maintains session identity and server uses token to locate client PROXY, e.g. shopping basket identifier cookie.
- Useful when client cannot keep whole state or is unreliable.

Consequences

- One token per session, e.g. single shopping cart per session.
- More responsibility placed on the client (state management).
- Increased need for client reliability.
- Minimises amount of required bandwidth.
- Security issues arise (client could forge session ID).

Relationships

- Classified as either implementation-level or architecture-level pattern.

SERIALISATION

Problem

- Creating a replica of an object in a different place, but channel/store doesn't understand objects per se.

Forces

- When to encode ops as well as attributes.
- Deep/shallow copy?
  - How to represent inter-object references.
  - How to avoid looping around circular references.
- Sending abstract versus concrete state.

Solution

1. Encode object('s attributes) as a load of bytes/text/XML/other-fashionable-stuff.
2. Squirt down comms channel or into store.
3. Decode at other end/on retrieval.

Consequences

Pros

- Deal with subset of object state/behaviour.
- Use byte-oriented tools/services (email, hexedit, zip, etc).

Cons

- Must have some common schema/understanding.
- More human-readable → more bandwidth.

Relationships

- Also known as MARSHALLING.
- Classified as object-level pattern.

MOBILE OBJECTS

Problem

- Too much communication with remote object.

Forces

- Reduce remote communication.
- Cost of messages is greater than cost of relocating.

Solution

- Move it towards client.

Consequences

- Need OID independent of location.
- Need some location-finding pattern.
- Need efficient marshalling.
- Allow more space (memory) for incoming objects.
- Behaviour must be mobile.

Relationships

- Classified as object-level pattern.
- SERIALISATION is how MOBILE OBJECTS work.