Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

APM

Utoepea and Merlin Review

Dave Otway

Abstract

This is the report of an external review of the Utoepea and Merlin projects for the Chemical Abstracts Service.

APM.1483.01 **Approved** 12th May 1995

Technical Report

Distribution:
Supersedes:
Superseded by:

Contents

1 Overview

Will the current design work?	-- YES
Can the team implement it?	-- YES
So what's wrong then?	-- it could be easier to debug and maintain -- it could be more adaptable -- it could go faster -- it could could be better positioned to exploit commercial off-the-shelf software
How can it be improved?	-- make the organization more flexible

2 A Distributed Systems View

2.1 Market trends

There is a general trend away from mainframe/terminal systems to client/server systems. This is accompanied by a shift from asynchronous messaging to multi-threaded remote procedure calls.

Asynchronous messaging was the standard way of connecting a (small) number of mainframes together, but it does not scale well to large numbers of nodes. The main problem is the uncertainty about who is responsible for a message when a node fails. Building reliable systems with this style of programming requires each object in the system to keep state for each message that it has processed. This state must be kept until either a reply, or a notification that another object has taken over responsibility for the message, is received.

The message state information increases linearly with the number of message types and the volume of messages. But the possible number of state transitions for each object increases exponentially with the number of message types in the system. The large finite state machines that result can be partitioned into ones of more manageable size by decomposing the system into smaller objects, but this exacerbates the communication problem described in the next paragraph.

The communication overhead is governed by the number of objects in the system. Although the communication is insensitive to the number of nodes in the system (because local and remote communications are equally inefficient) the remote communications overhead is paid on every object interaction regardless of whether the objects are local to or remote from each other. [The round trip time for a pair of null request and reply messages between two single threaded active objects is of the order of a few milliseconds (pretty much regardless of their relative locations) while a null procedure call on the same thread between two co-located passive objects is less than 10 microseconds.]

For single threaded objects, the concurrency of the system is governed by the number of objects. Tuning this major performance parameter involves a redesign of the system. While for multi-threaded systems the applications concurrency can be optimized simply by adjusting the number of threads.

For the above reasons the “client/server”, “object-oriented” distributed systems market is moving away from asynchronous messaging to multi-threaded remote procedure calls (RPC).

A direct consequence of this is that the suppliers are investing heavily in improving the reliability and performance of their thread and RPC implementations. These implementations are also being moved into the kernel for even greater efficiency.

The more CAS can align its designs with the “industry standard” way of doing things then the more it will be able to use commercial off-the-shelf solutions, with the consequent saving of development and maintenance costs.

2.2 Shifting requirements

The proposed utoepea and merlin designs are (mostly correctly) solving the problems of the previous generation of systems. These systems successfully used asynchronous messaging on a small number of reliable nodes, but had serious problems with concurrency bugs.

But the new systems will be radically different in two important aspects:

1. there will be more numerous but less reliable nodes
   [one of the benefits of distribution is that the system can use cheaper components and survive (most of) their failures]
2. an object oriented design and implementation is being used

In such a system, reliable asynchronous messaging will run into scaling problems, while strict encapsulation will severely restrict the scope of possible concurrency conflicts. [An object can be shrunk to fit each specific problem.] I think that the fear that the current programming team can’t cope with concurrency is misplaced. Programmers that can successfully debug a large scale asynchronous message system are certainly intelligent enough to handle small scale concurrency problems, they just need to be retrained.

Also, I don’t believe you can successfully hide all concurrency problems from the applications programmers, and any that escape will inevitably be the really nasty ones. This will result in those programmers that have been deliberately isolated from the effects of concurrency having to deal with its hardest problems.

Both the problems and benefits of distributed processing stem from concurrency. If the programmers are trained to think about the effects of concurrency in every part of the system then they will quickly recognize and deal with any problems. They will also be able to benefit from concurrency by exploiting application knowledge to choose more optimistic mechanisms.

[For example, if the chances of a conflict on making a coordinated set of updates was statistically insignificant then an alternative to locking all the records while the update is computed and performed would be to timestamp or sequence each record when it is updated. The records could then be read and the updates computed without any synchronization controls. The records would only need to be locked for the duration of the timestamp or sequence check and the actual overwriting. In the rare case that any of the timestamps or sequence numbers have changed during the computation, the set of updates is aborted and restarted.]

In summary,

- if programmers can’t cope with concurrency they won’t be able to cope with large scale asynchronous messaging either
- it is less risky to retrain them to use a programming style that is more appropriate to the problem than to continue with the old one the new style will result in more reliable and efficient programs

2.3 Over specialization

The current utoepea design is optimized to solve one particular class of problem, that of implementing asynchronous bidirectional communications protocols.

This class of problem is typical of only half of the merlin system (access in and access out) and is very untypical of distributed applications in general.

A glance at the communication patterns for the merlin executive and database engine shows that the request and reply messages follow the same flow as nested procedure calls. Also, there is only a minimal requirement for concurrent access to data.

Changing the implementation style for these applications to multi-threaded local procedure calls between passive objects would reduce the inter-object communications delays, remove unnecessary queuing, simplify and speed up the processing of local state (by moving it from context objects into local variables on the stack) and make it easier to tune the level of concurrency.

The executive's session manager would be a good place to control the overall concurrency. It could start a new thread for each operation up to an optimum level of concurrency and then queue additional operations until a thread becomes free.

The only complications are with state that persists across user operations (i.e. session state) and concurrent operations in the same session (i.e. a user abort). The session manager would still need a context object.

User abort operations would be done by a concurrent thread setting a flag associated with the session being aborted at one of a (small) number of strategic abort points. These abort points would be placed in the major loops of an operation. The session flag would be cleared before the loop was entered and checked on each iteration. The session manager should keep one thread for exclusive use by abort operations to avoid them being queued in busy periods.

2.4 No metrics

The requirements don't contain any numbers, either for the old system or the new. If you don't know what the numbers are, they will probably be bad.

2.5 Specific recommendations

Measure the performance of the old system.

Measure the performance of the those bits of the new system that have been built.

Repeat and plot performance measurements every month. Figure out why they have got worse and how to improve them.

Don't change access in or access out.

Make all objects except session manager in the executive and database engine passive. Move all context information (except for sessions) onto the stack. Use procedure calls to communicate between objects.

Make references to passive objects a smart pointer to their (interface) class instance, not a port.

Change session manager to use a different thread for each operation. It should generate threads up to (a measured) optimum number and then queue extra requests. Returning threads should execute the first queued request or wait for the first new request to arrive.

Keep a dedicated thread for aborts.

Identify strategic and clean abort points for all long running operations and program the synchronization with the abort operation as outlined above.

Define the interfaces between access in - executive, executive - database engine and database engine - access out in CORBA IDL.

Use a CORBA stub compiler to generate the C++ header files for those interfaces. Use these header files to build the system; using local procedure calls.

Place the four applications in separate processes on the same node and measure the performance.

Place the four applications on separate nodes and measure the performance.

Evaluate the best configuration and whether to partition the applications any further.

Re-evaluate the whole design in the light of experience.

3 An Organizational View

I wasn't specifically asked to comment on management issues, but I feel that they are affecting the software, so in my humble opinion these are where I think the problems lie.

3.1 Rigidity

I don't know whether this is a result of a deliberate management policy or a communication failure, but the development team is designing software as if the solution to every problem has to be made to fit the existing organizational structure and responsibilities. They do not think that they are allowed to consider the possibility that the organization could be changed to make it fit the problem.

My impression of the prevailing culture is that people think that they can do only those things that they have specifically been given responsibility for.

The development systems appear to be managed as if they were operational systems in use by the general public.

An organization can't become more responsive to the requirements of its customers and capabilities of its competitors without being flexible itself.

The development team needs more freedom to experiment with and compare different hardware and software products. They have got to keep abreast of the market (see below). To do this they need to deploy products, that may never become operational, with minimal effort and at short notice. It is important to minimize the cost of evaluating the products that are discarded as these will be much more numerous than the ones that are eventually used.

They also need to control the timing of changes to their computing environment to fit in with their own timescales rather than those of a department with much larger and very different concerns.

The development team also needs control of all of its budgets so that it can trade off investments in hardware, software and effort.

Although the decisions on what operational hardware and software products to buy and what organizational changes to make will be taken by senior management, it should be the responsibility of the development team to explore the available options and keep open any that might be useful in the future. They should be encouraged to discus requirements and possible solutions with as wide a range of people as possible (customers, suppliers, marketing, sales, operations). Rigid lines of communication up and down the management tree are only necessary when decisions need to be taken.

For example, they should keep open the option of switching to a cheaper hardware supplier in future years by ensuring that the systems they develop run on a reasonable spread of machines. They should also try to ensure that

any infrastructure or libraries are suitable for as wide a range of future applications as possible.

Keeping options open and designing for re-use are ways of managing risk, it balances a known small expenditure now against the risk of incurring a large one in the future. Future re-use and flexibility don't come for free or by accident. They have to be paid for and managed. This must be done separately from the current work, otherwise short term pressures will prevail.

3.2 Isolation

Twenty years ago CAS had an application on the leading edge and had to develop its own solutions. But the rest of the world hasn't stood still, they have caught up in most areas and forged ahead in others. CAS no longer has a special application, just a few constraints that are more severe than normal.

There is a lot of things out in the market that could be of use. If a product doesn't quite do the job then try and encourage, demand or pay the supplier to extend the product to meet CAS's requirements. This generally means forming a relationship that extends beyond the sales critter.

Development staff should to be actively encouraged to interact with other organizations and learn from their experience. They should go to conferences, read the literature (rather than the media), join special interest groups and user organizations, evaluate new products, talk to their peers in other user organizations and form links with local universities. Use the internet, there is a lot of good information out there for free, once you've learned where to look.

A policy of no recruitment is understandable in the circumstances but also increases the isolation of the remaining staff. A regular supply of new experience and up to date training would benefit the team.

The internal walls need to be breached as well, operational staff will have learned valuable lessons and should be consulted at all stages. They should also be given a sign-off on new systems, if it doesn't meet their requirements it shouldn't go live.

3.3 Morale

The development team is a very competent bunch, but they are very frustrated. This could quickly be turned into enthusiasm by making the organization more flexible.

3.4 Specific recommendations

Encourage the team to talk to anybody that they think may help them improve the system or organization. Management permission should only be required to actually implement changes, not to discus them.

Make it explicit that the development team have a responsibility to look over the horizon and anticipate problems and opportunities. They should not limit themselves to just considering the current project.

Eject people from the building on a regular basis. Send them to conferences, workshops, user organizations, training courses, local universities and customers. Its cheaper to learn from other peoples mistakes.

Make each team member responsible for keeping up to date in a specific subject or product of (possible) relevance to CAS.

Form better relationships with suppliers. Customers have to invest time and effort in their suppliers if they want to get the best out of them.

Budget and evaluate re-usable software separately from projects. Re-evaluate it after every project and retrofit any changes to all applications to prevent divergence and reduce maintenance.

Broaden the development environment by providing a (small) variety of competing platforms to some of the development staff. It is easier to develop on several platforms in parallel than to do serial ports afterwards. You will find the problems at an early stage and learn to avoid them. Use a single source tree and compile into a separate shadow tree for each different platform. Keep all configuration specific code in one place. Build and test the current source tree on all the platforms at least once a month (this is what weekends are for -- why let all those mips go to waste).

Claw back the development share of hardware and software central budgets. You can then choose the most appropriate products, make changes faster and trade off between investments in hardware, software and people.

Detach the development systems from operations. Make the development team take over the responsibility for their own systems. Absolve operations staff of any responsibility for development systems. Connect the development network to the operational network via a packet filter under the control of the operations staff.

Make each development team member responsible for the administration of their own workstation. Appoint one team member to coordinate and maintain a base set of software on a central server. This will reduce wasteful duplication without restricting flexibility.

Involve operations staff in setting requirements, design reviews and quality control. Give them a sign-off on systems going live.