# The Database Machine: Old Story, New Slant?

**Julie A. McCann**

Department of Computing
Imperial College London, UK
jamm@doc.ic.ac.uk

## Abstract

Current database management system technology is not well equipped to provide adequate support for what has been deemed the 3rd wave of computing -- Ubiquitous Computing. Such applications require systems that are sufficiently lightweight and customisable to provide high performance while consuming minimal power, yet extensible enough to adapt to a constantly changing environment. Current DBMS architectures inherently do not provide this level of customisation or adaptability. Therefore we suggest an alternative where database systems shake off their relatively static monolithic structure and become open sets of *fine-grained* components providing a collection of key information provision services and moreover have the ability to adapt. This paper explores the motivation for componentisation and how modern operating systems research can influence the DBMS architecture. If components are the answer, then are we announcing the end of database management systems as we currently know them, or are we just describing a database machine for the 21st century?

## 1 Introduction

Traditional database management workhorses such as transaction based computing in finance and number crunching in scientific applications will certainly be around for some time. However with computing devices becoming more ubiquitous and pervasive due to increased power in small devices, mobility, and use of the web, we are beginning to see new challenges regarding the management of data. Since data in such environments are predicted to be less centralised and themselves pervasive, the concept of a 'Database Management System' is essentially defunct. That is, over the years we have come to view the database as large structured sets of persistent data. It is this view that has essentially kept the architecture of the management system static since the seventies. Data contained in pervasive computing systems cannot be viewed as a single *base* (physical or virtual), therefore a radically new way of looking at data management is required.

Pervasive data systems can be viewed like an extremely distributed database except that datasets are much more fine-grained, versioning is more important, and data representation is highly heterogeneous. The types of computers supporting these are typically anything from a set of sensors, PDAs, mobile phones and webpads etc. to servers. What is required of such services are things like the ability to *trust* the data (something that is assumed more readily in traditional DBMS), the ability to cope with slightly out-of-date data, and the ability of a query's answer to change as requirements change dynamically at run time. At an architectural level the system must be able to cope with units failing – perhaps mid way through answering a query (and being replaced with minimal maintenance or the whole processing 'jumping' to another device to continue/finish). Running in embedded and smaller capacity devices means that it must be as lightweight as possible and able to adapt its own functionality at run-time (i.e. be self-reconfigurable or self-healing thus reducing systems' administration and maintenance). Most importantly, since the architecture becomes more adhoc and thus complex, strong software engineering standards must be encouraged to avoid systems misbehaving.

Currently a DBMS controls the organisation, storage and retrieval of data whilst regulating the security and integrity of the database, it accepts requests for data from the application programs and instructs the operating system (OS) to transfer the appropriate data. As a resource manager the OS and DBMS function in a very similar way but with differing levels of abstraction. The broad functionality of the modern DBMS has meant that typical DBMS architectures have become bloated



**Proceedings of the 2003 CIDR Conference**

compromising the DBMSs ability to meet changing application demands, but moreover it disables its ability to self-regulate and dynamically change which potentially renders it unusable by many modern and future applications.

An established solution to providing self-configuring systems, that are lightweight and highly flexible, is that of component-based architectures. Most component-based research has focused on relatively thick-grained components, at the level of middleware and COTS. The DBMS community, who have recognised for some time that finer-grained componentisation of the DBMS is required [23], have thus far only examined thick-grain components [12][1]. However, this paper presents a fine-grained component architecture. The reader will readily see that the DBMS no longer exists in this architecture as the boundaries between a DBMS and the Operating System (OS) have become blurred – in fact, there is no DBMS or OS in this architecture just components and hardware and some 'intelligence'. Essentially we believe that to have a truly fine-grained DBMS we must have fine-grained components throughout (i.e. including the OS). By dissolving the software into its elementary units, then using these as building blocks that are configured/re-configured based on the 'intelligence' we should result in resource management that was truly adaptive whilst remaining lightweight.

The next section discusses component-based computing from both the OS and DBMS architecture points of view. An introduction to adaptive systems and how they relate to component-based systems for self-reconfiguration follows this. We then introduce our Adaptive Data Management architecture and discuss it briefly using simple scenarios based on a ubiquitous computing environment to illustrate how adaptivity can be embedded in components. Finally we conclude and, using our experiences, try to speculate what the future issues might be.

### 1.1 Component-based Systems

There is no universally agreed definition of the term *component*. In this paper we define a component in terms of its use; where objects are fundamentally a programmers' tool, components are concrete entities consisting of implementation and interfaces. The boundaries between components are concrete and are present in a *running system*. Therefore, we term component-based OS or DBMS as those in which the Kernel is decomposed into building blocks or components such as schedulers, memory/buffer managers and security managers etc.

We have known since the industrial revolution that componentisation reduces production costs and the time-to-market. However for a long time many of our current fundamental applications such as OS and DBMS, have been frozen executables primarily assembled for a single processor where the configuration of the application was exclusively under the control of the developer. However the monolithic nature of these systems was primarily due to performance restrictions and not lack of foresight of the software architect.

The past ten years has heralded the emergence of software engineering frameworks and languages designed to support *component-based* application design. The advantages are two-fold: the assembly of applications from pre-fabricated software components is timely, and developers can purchase interoperable software components off the shelf. Further, facilitated primarily by object technology, we can see that today's emerging component-based systems are no longer two-tier client-server models tied to a single platform. Instead they are multi-tier, distributed and dynamically reconfigured across a heterogeneous computing base.

Until the 1980s almost all OS were inflexible in nature with all systems services incorporated into a *monolithic* kernel in which costly system calls were implemented by trapping from user mode. As all services were precompiled into the kernel it was impossible to load them on demand. Furthermore, reconfiguration, such as connecting a new device, required recompiling the entire kernel. In the mid 1980s, *microkernels* were introduced as a result higher demand from distributed systems users. Microkernels deployed object oriented software. They were relatively lean and efficient, realizing only necessary OS functions, by moving all services (that is, Networked File System, Shared Memory etc.) to the user-level. This resulted in an increased level of flexibility. Nevertheless, they were not adaptable and the microkernel was not lightweight or efficient.

Many researchers considered the major factor limiting both system flexibility and performance to be the fixed high-level of abstraction [8, 9]. It is for this reason that OS research began to search for better abstractions for finer granularity in system services [8]. These systems are termed *extensible kernels*. Elimination of unnecessary abstraction, pushing the interface nearer to the hardware, ensured a significant performance improvement [9]. However they lacked the ability to tailor the OS to the application and be re-configured at runtime to adapt OS processing on demand. Consequently research looked at decomposing very core of the OS to be into its logical components.

Nevertheless, modern OS design community has been relatively slow in the uptake of component-based engineering mainly for exactly the same reason monolithic systems prevailed for so long – performance[2].

[1] The author's initial chapter discusses the requirements for truly componentised DBMS yet subsequent chapters in the book outline middleware level componentization.

[2] Similarly, one revolution that made early versions of Unix interesting was that it was not coded in Assembler but in a high-level language, C, to gain the software engineering benefits that high-level languages

The two most notable research efforts into component-based operating systems is SawMill [18], based on the L4 $\mu$-kernel and Pebble [10].

### 1.2 DBMS Componentisation

Work on evolving the architecture the DBMS architecture has followed a very similar path to that of the OS but has to some extent lagged behind. For example some research has been carried out on adapting lightweight DBMS (LWDB) [4, 27, 16]. Often, these systems omit features found in traditional DBMS, and include more specialised features which maximise performance. Such systems are very limited to particular applications and most were custom built and remained monolithic.

Research has also considered extensible systems to tailor a DBMS to a particular application, [2, 13, 7, 3]. These, though very much in the right direction, were essentially customisable monolithic DBMSs, their architecture being layered resulting in performance overhead problems, which would not suit the types of applications, we wish to focus on. However focus on performance not the only issue. To provide the amount of flexibility and runtime adaptability for modern applications, component-based DBMS are required. That is, decoupling DBMS services allows the functionality required at a given time to be swapped in on demand. This is impossible in DBMS architectures that are monolithic regardless of how lightweight and extensible they are.

Some work is closer to our thinking, for example [28] looked at a tool that took Smallbase [16] and decomposed it into components, the primary objective being reuse identification rather than architectural flexibility or dynamism. Since they restricted themselves by using C, any form of adaptivity and re-configuration is impossible. Nevertheless, the components were relatively fine-grained –resource manager level, and the system did demonstrate reasonable performance. Further, [6] presents a similar architecture to the one we present here, with very much the same reasoning. That is, we agree that the '*feature-overload'* of modern DBMS architectures will cause future problems. They too suggest that the DBMS processing be broken down into specific functions such as a select-project-join processor (SPJ) and that the system should be self-tuning. However their solution radically differs in that our suggested components are targeted at a finer-grain and at lower level operations (such as get-page). Further we are not tied to a single data model (relational) nor are we suggesting strictly limited data types (quite the opposite). Finally the *Universal Glue* i.e. their equivalent of our 'intelligence' is targeted at the middleware level; a much higher level of abstraction compared with our policy style glue which we in Section 3.

provide. This was hitherto resisted as it was thought that the object code for the OS compiled from a high-level language would perform too poorly.

## 2 Self-Adaptive Systems

An adaptive system is one that can modify its behaviour based on stimulus produced either from within its system or from external sources. Pervasive computing requires that the units and their service have a degree of autonomy and therefore adaptivity would have to be quite self-contained. The more autonomy the unit requires, the more complex the processing where the system may have to evolve new behaviours. This is where the major tradeoffs lie – the more 'intelligent' the system is the more complex the task to decide how to adapt is. This is in terms of the processing and information storage of monitoring feedback, the change optimisation and the cost of the reconfiguration itself

Two major areas of computing have examined this subject from different angles. Artificial intelligence has focused on the rules and mechanisms to allow open-adaptive systems to evolve new behaviours, focusing on very narrow application areas [25]. Alternatively Software Engineering has considered component-based architecture configuration languages and adaptation policy management [22]. Thus far, closed-adaptive systems have only been considered. Either way, an adaptive OS and DBMS architecture must not preclude the type of adaptive system it supports. This means that the system must be able to provide facilities for dynamic reconfiguration, the ability to store adaptivity rules, and the ability to react to those rules in a way that does not compromise performance. There is no point in a system reacting to a problem so slowly that system fails before it can do anything about it. Furthermore componentisation itself must not produce excessive overheads.

Typically adaptive systems have been designed in a non component-based fashion. This is to allow the system itself to change or evolve functionality. However these systems will become difficult to maintain and engineer. Furthermore, without componentisation lightweightedness is compromised (i.e. the parts of the system not required at a given time remain in the architecture bloating the system). Therefore we advocate trading-off adaptive flexibility for a more component-based architecture as a good compromise. However if the componentisation is fine-grained the added flexibility can be maintained to a degree e.g. a function consists of a number of small building blocks, which can be composed and recomposed to adapt the functionality at runtime.

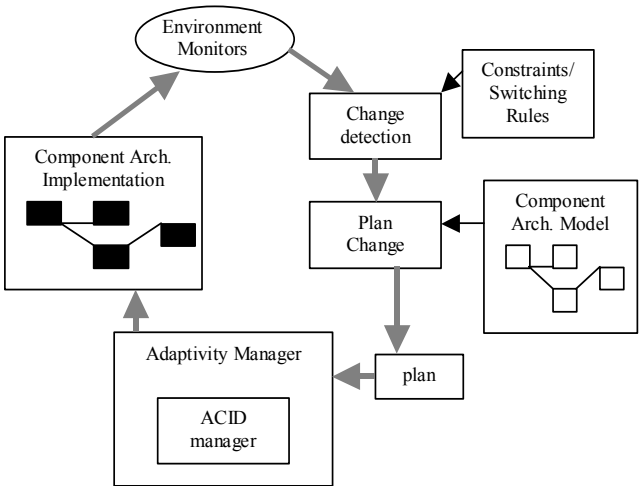


**Figure 1 Adaptation Framework**

This past two years has begun to see some research in adaptivity for DBMS. The nature of Internet applications querying data from highly heterogeneous distributed databases over wide-area networks has encouraged exploration into adaptive query processing. Here the optimiser is initially unable to accurately estimate a query's cost due to the lack of histograms, statistics and metadata from the diverse data sources. Therefore, adaptive query processing is where the optimiser adapts its execution in response to monitored data sizes and transfer rates as the query is being executed. This research has entailed examination of incremental updates, query materialisation points for data reuse, and result approximation. Examples of this work are pipelined hash join [31], hash ripple join [14] and the Xjoin [29]. Most of this work is with relational data and concerns aggregation queries as examples [1, 15], however some have looked at XML [17]. Nevertheless this work has been very focused and has not examined the complete database systems architecture.

## 3 Adaptive Data Management Architecture

This section presents a general architecture to support adaptive data management. It is the basic framework used in the subsequent applications illustrated below. Figure 1 illustrates a general component-based adaptive system architecture. The main feature is that this system can self-(re)configure with the help from monitors, which provide environmental data (e.g. current performance statistics).

The typical tools used to develop such systems consist of architecture description languages (ADLs) and constraint solvers [22]. An architecture is generally considered to consist of components and their interactions. Architectural description languages allow the software engineer to formally abstract the architecture so as to reason about it. An ADL can give a global view of the system and when augmented with constraints, the validity of change (the reconfiguration of components) can potentially be evaluated at runtime [22, 11].

Sophisticated adaptive systems can be composed of components that in turn are composed of sub-components. In our architecture a component consists of both the application logic, the architectural description of itself (i.e. the component structure) and a copy of the switching rules relevant to it as well as a lightweight adaptivity manager. A session manager is fed information from monitors or *gauges* (which aggregate raw monitor data for more lightweight processing). The current configuration operation is being monitored by the session monitor who constantly checks constraints and, if broken, consults the switching rules to decide how best to overcome the problem. When adaptivity is triggered the component architecture model allows an alternative execution plan to be designed. The session manager decides how to instantiate the alternative component architecture and passes his alternative over to the Adaptivity Manager. The Adaptivity Manager then carries out the unbinding and rebinding of components (establishing any glue necessary to achieve the binding). To do this it must ensure the instantiation adheres to transactional style prosperities. That is, the switch can be backed off if something goes wrong.

The architecture described here is essentially a closed-adaptivity model however it is hoped that the design is general and flexible enough to implement an open model. That is, the actual component architecture model, the constraints and switching capabilities do not themselves adapt. It could be argued that an open-adaptivity model would compromise the black-box nature of component-based software engineering, as access to sub-components is required to update their inner structures etc.

In a highly adaptive system the component can migrate, as can the data component. This is where the component migrates to a part of the system to ensure a constraint is not broken (an example of this is illustrated in the next section).

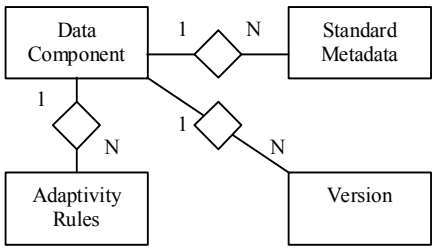


**Figure 2 Data Component Structure**

## 4 Ubiquitous Computing DB Scenarios

Ubiquitous computing highlights an interesting set of simple scenarios with regards to data management in that

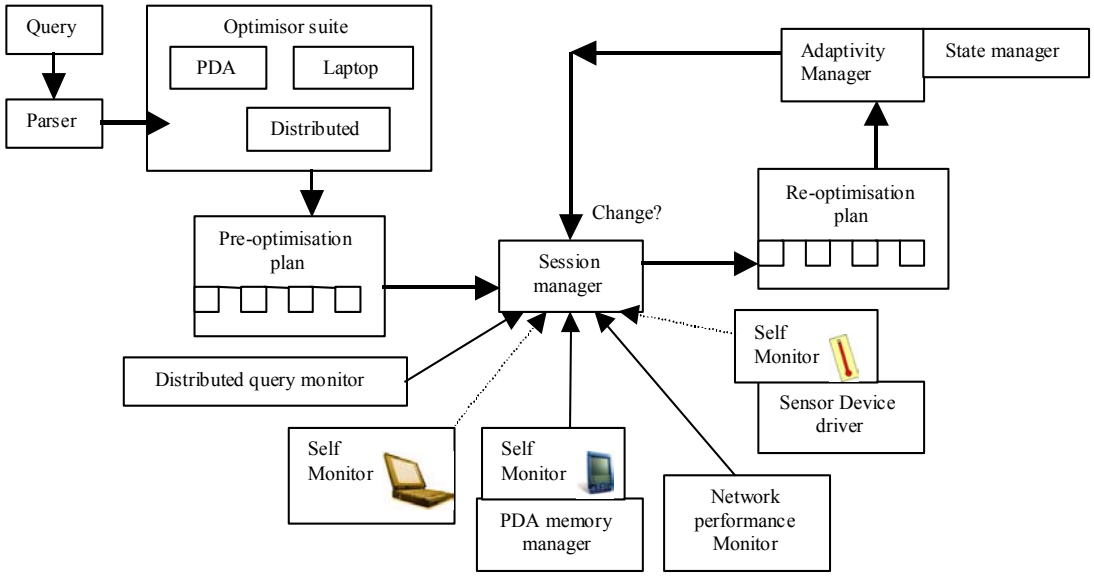


**Figure 3 Component Architecture from an adaptivity point of view**

it consists of a number of very differing devices of varying (usually low) capacities, storing data of very differing structures. Ubiquitous computing requires not only the system to adapt for performance and fault-tolerant reasons, but do so in a lightweight manner.

To illustrate how we think this would operate, we have a subset of a ubiquitous system that consists of a sensor, a Laptop and a PDA. The Laptop and PDA can make use of the sensor's data (which is streamed in XML format). Further the PDA and Laptop can share data. The systems architecture with interesting components, for this is described in figure 3.

The data is divided into the structure described in Figure 2. Example data could be OO structured data concerned with a person or a relational table used for transaction processing or an XML stream. The metadata represents the standard metadata found in traditional databases e.g. attribute statistics, triggers etc. The Adaptability Rules are the list of rules associated with the adaptivity constraints[3] and the action(s) to be taken when the session manager has detected that a constraint has been broken. The list of versions is indications of where alternatives can be found. Versions are not necessarily exact replicas; they could be compressed versions of the data (perhaps with associated decompression code) or be out-of-date. They also could be lower quality versions or summaries of the data.

The **first scenario**: *inter-query adaptation*. The query has been initiated by a PDA and requires data from the Laptop or another PDA over a wireless network. The data component takes the form[4]:

```
Personal data <id, name, address, age, metadata etc>,
        <Select BEST (PDA, Laptop)>,
        <Select NEAREST (PDA, Laptop);
```

When the query enters the system and the optimiser draws up an initial pre-optimisation plan. The optimiser activated at this point may be on the PDA device itself or be run on another device (e.g. the Laptop) either way the DBMS uses the PDA optimisor to build the initial plan. The Session Manager then takes the plan and checks the monitors that are feeding into it. Currently the PDA and Laptop monitors are providing performance of themselves. Further the network performance monitor is providing current bandwidth statistics. The DBMS understands the function BEST to mean the best device in terms of capacity and current load. At the moment the Laptop is better as it is not being used and has much more capacity compared with the PDA to that version is delivered to the PDA that initiated the original query. Functions like NEAREST could indicate the closest data resource and the constraint rules themselves can be prioritised. That is BEST, like NEAREST, is parameterised with representations of the two computing nodes to be compared.

[3] Note that these constraints are not the same as operation constraints such as integrity, these constraints work at the sub-operation level.

[4] Here we are using tuple structures to describe the data component and its constraints. Alternatives such as XML could equally be used as could more formal logics. As the example constraints are for illustrative purposes only they are acknowledgeably simple.

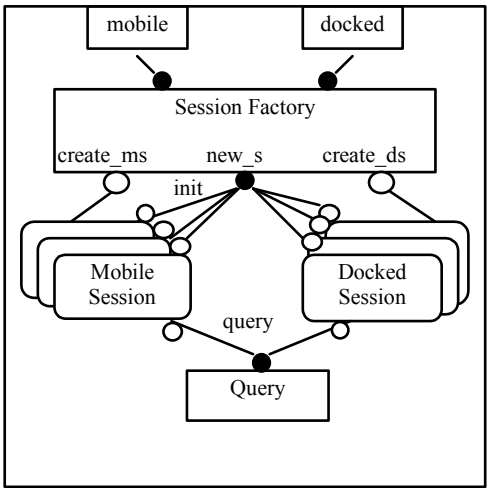


**Figure 4 Darwin description of mobile CBMS**

The **second scenario**: *system adaptation*. This is where the Laptop has asked for the information from the sensor device. The Laptop was plugged into the electricity and Ethernet (i.e. docked) when the request was initiated but in the meantime it has been unplugged and is now working off the battery and wireless network. This is an example of architectural reconfiguration.

Figure 4 shows the configuration of the components composing the management system *within* the Laptop. Some of the components may be already stored on the Laptop (e.g. wireless device driver) or can be retrieved of the network (e.g. wireless optimisor). This diagram is illustrated using the graphic form of the Darwin configuration language[5] [22]. The query is initially part of the docked session, but must switch over to the wireless session when the Laptop is disconnected.

Figure 5 shows the difference between the components that are activated during both sessions. Essentially the relevant device driver components will be swapped out and e.g. the wireless network driver activated (not in figure). Further the query optimisor was initially taking static resources into account and now the wireless_optimisor must activate and amend the query plan accordingly. This means that while the sensor is streaming the original optimiser initially planned for a strong high bandwidth connection, now it cannot guarantee this and so decides to send a compressed version of the data thus using more resources on both the sensor and the Laptop while saving communication time. The original query plan included safe points which allow the system to stop streaming at a safe time and continue the other version's stream. The Adaptivity manager ensures this happens in a consistent manner and provides an amended query plan accordingly (see final scenario).

The **final scenario**: *intra-query adaptation*. Again the Laptop is issuing a relational query, which involves heavy join processing with updates as opposed to a simple stream of information. Here the statistics provided by the metadata are not quite accurate enough for the pre-optimisor to build the optimal plan. It becomes obvious that the original cost calculations need revised; therefore the Session Manager indicates to the Adaptivity Manager that this is the case. The Session Manager is itself componentised in that it can have optimisor functionality added for data processing. The query plan is revised to perhaps change the join's inner-loop to the outer-loop or add an index to one of the tables. The components that carry out this are called upon and linked into the query pipeline at run-time. The Session Manager provides the Adaptivity Manager with a revised plan and hands off query execution. The adaptivity manager brings the query to a consistent state maintained by the State Manager[6] component. The query then continues from this point.

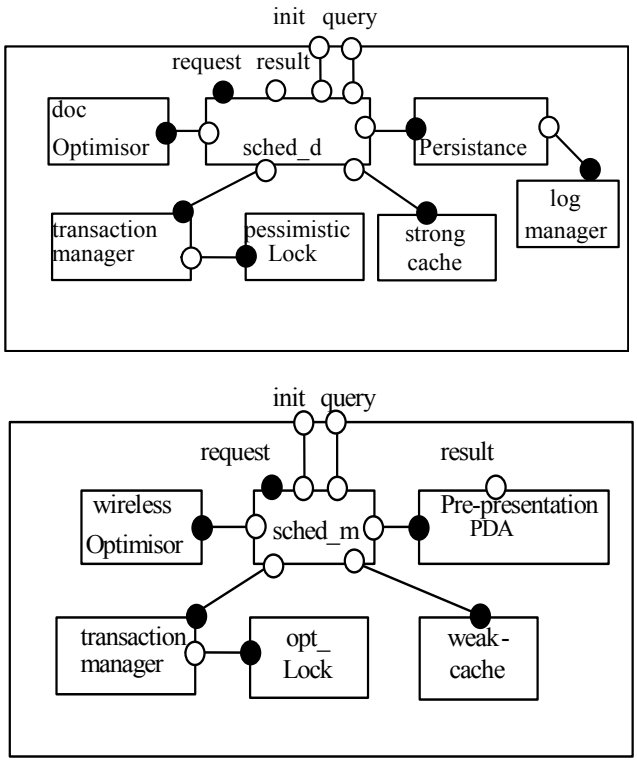


**Figure 5 Darwin description of switchover between docked (top) and wireless (bottom) session**

[5] The diagram uses the graphical form of the Darwin architectural description language. Darwin views components in terms of both services they provide (to allow other components to interact with them) and services they require (to interact with other components). A provided service is represented by a *filled circle* and a required service is represented by an *empty circle*. Components are shown as rectangles.

[6] Note that the state-manager component is only called upon at this time, as it was not needed by the previous two examples, as they were not carrying out an update.

## 5 What we have done in this area

This section introduces two projects (Go!, and Patia respectively) that are examining various aspects of the adaptive component-based data management architecture we describe above. We introduce the foundations of our system, which demonstrates that even the OS kernel can be componentised in a fine-grained manner without compromising performance indicating that a DBMS can also achieve this level of flexibility. We then briefly look at how adaptation is used in our component-based adaptive Webserver – this demonstrates similar behaviour to the general data management system in section 4.

### 5.1 Foundations – Go!

To obtain a highly flexible, configurable and yet lightweight system componentisation must exist throughout the architecture; i.e. including the OS 'kernel'. Our initial research lead us therefore to look at the foundations of such a data management infrastructure; component-based operating systems.

Our objectives are two-fold. Firstly our aim was to minimise the core of the OS and therefore ideally any service that has nothing to do with component management (e.g. interrupt and device management) would be handled outside that core. This helps with lightweightedness and increases flexibility [19]. For an OS to support applications such as mobile or ubiquitous computing we also require that overheads in general and inter-component communication be minimal. These aims have led us to focus on the OS protection mechanism. We envisaged that, in a decomposed system, separate *components* should be responsible for these distinct tasks to improve configurability, dynamism, robustness and software engineering.

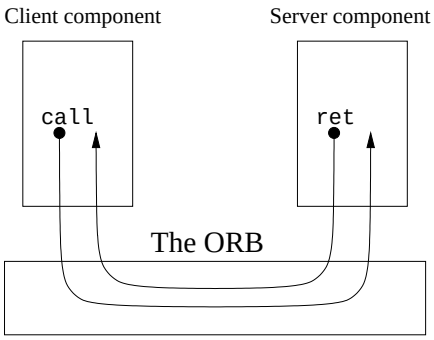


**Figure 6: Components invoke services via the ORB**

The reason that even the leanest research OSs typically have interrupts etc within their core is because the core manages protection as it alone has sufficient privileges. When the CPU is in *user mode* a subset of instructions become unavailable (e.g. instructions to control interrupts). However, the disadvantage is that separate processor modes *(kernel and user)* prevent protection function being cleanly separated from the management function and as a consequence there is a performance cost – context switching.

What was required was to overcome this problem by focusing on protection mechanisms. The resulting concept was SISR (*Software -based Instruction-Set Reduction*)[21]. What is unique to SISR is that there no longer are two process modes thus the switching overheads between user and kernel mode is eliminated. Instead, on loading, code is scanned for illegal operations and if detected the code is rejected insuring adequate process protection. That is, SISR removes the need for two separate processing modes by making use of *code-scanning* and *segmentation* memory protection (rather than paging like Unix etc). The unit of protection in SISR is the *component*, which is protected through its own data segment and is of a given type (which has its own segment). When a component instance is active on a CPU, the instance's data segment and type's code segment are referenced by the CPU's data and code segment registers respectively. Memory protection is enforced because SISR considers a segment-register load a privileged operation.

User level components are prevented by loading segment registers because code-scanning ensures that no components' text section will contain such instructions. This means that loading new values into code, data, and stack segment registers implements a context switch (which amounts to only 3 cycles on a Pentium).

A truly component-based OS can be seen as a *zero-kernel* system, where the kernel has been replaced by a set of components that cooperate to provide services usually found in traditional kernels. However, to invoke services on other components a privileged component known as the ORB[7] is used to load segment registers to 'switch a context'. This is the nearest part of the OS analogous to a kernel. For example, if component A wishes to evoke a service on component B then it indirects via the ORB component (which loads new code and data segments to perform the protected intra-machine Remote Procedure Call -- RPC). This is done by migrating the thread from caller to callee on the call and back again on return, as illustrated in Figure 6. These have the advantage of reducing the basic RPC times. These aspects of the OS are

| Operating System | Number of RPC (in cycles) |
|---|---|
| **BSD (Unix)** | **55,000** |
| **Mach2.5** | **3,000** |
| **L4** | **665** |
| **Go!** | **73** |

**Table 1: Relative RPC performance**

[7] like a Corba ORB however its functionality is more fundamental and it does not conform to the CORBA standard.

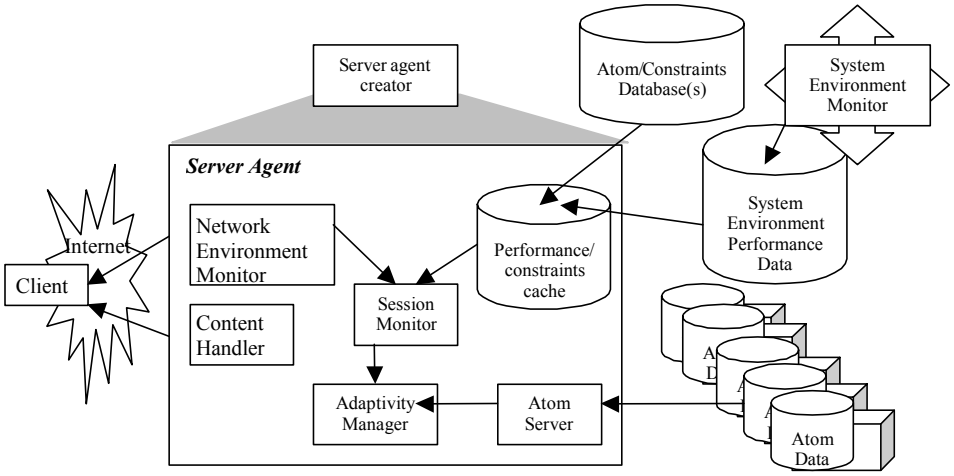


**Figure 7 Overview of the Patia Webserver architecture**

described in much more detail in [19].

The architecture then consists of an ORB and a library operating system, which contains the components composing the OS. As a proof-of-concept we have built the Go![8] OS, which has been developed for IA32 based architectures. So far, it has demonstrated that SISR offers greatly reduced protection overheads while allowing simpler and more decomposed architectures. The resulting relative performance improvement can be summarised in Table 1.

As can clearly be seen from Table1, Go! consumes many orders of magnitude less cycles than BSD which shows the ballpark figure of basic procedure call overheads of a modern Unix system. Furthermore, the component core (namely the ORB) uses very little memory -- the space required per component is just 32 bytes for each interface [20]. This is around two orders of magnitude improvement over page-based protection models found traditional operating systems.

### 5.2 Patia Webdata Architecture

A project overlapping with the Go! project is Patia[9] [26]. It is based on our adaptivity architecture and combines some agent-based technology. The amount of DBMS data that is being served via a Webserver has increased exponentially. Poor performance (or lack of response) is becoming more common, which is accentuated by unexpected flash crowds. Adaptivity in Webserver architectures can be achieved at the inter-request level and the intra-request level and can help with not only improving server performance but also network performance. An example of inter-request level adaptivity would be that a page has been delivered and when the client requests a given graphic the version of the graphic sent is one which best suits the monitored bandwidth between the server and that client. Intra-request adaptivity could be that while the server is delivering some streaming media (e.g. audio) the codec of the stream is chosen to best suit the bandwidth, and if the bandwidth should change during mid delivery, then a new less bandwidth hungry codec is swapped in [23]. These examples illustrate how adaptivity helps performance, however adaptivity can help with fault tolerance. For example if a monitor detects, through some from of trend analysis, that the number of requests are beginning to peak beyond a given threshold then it can dynamically spread its processing (e.g. to non-Webserver machines like a typing-pools' word processing computers), which can help with flash crowds.

To achieve flexibility *both* the data and the webservice applications are componentised. This means that the components that compose a webpage can be distributed over many machines. This can provide the advantage of intra-request parallelism as well as fault-tolerance where replication is used.

Each unit of data is known in Patia as an *Atom*. We are assuming that the web pages are composed of things like graphic/text/streams etc. In this scenario we define the Atom as the smallest web object that cannot be sub-divided[10]. Examples of this would be a video stream, graphic, a navigation button, a text frame etc. Webpage Atoms are distributed over the nodes in the system[11] and some may be replicated.

For each atom there is a unique identifier, name and set of constraints. The Atom follows the data structure of Figure 3 and its tuple structure is:

**Atom = <a_id, name, type, <constraint>>**

[8] Go! stands for Greg's Operating System.

[9] Patia is a shortened form of Hypatia, scholar of the ancient world

[10] We had considered that an Atom should be an object that it is *best not to further sub-divide* this means that the Atom can be a complete web page with text and graphics.

[11] We are not considering partitioning mechanisms in this scenario.

The Webserver code is also componentised. Figure 7 illustrates these components. The request comes into the system; is received by a *service-agent component* who takes this request finds the appropriate Atom and serves it to the client. The client may make subsequent requests directly to that *service-agent who delivers the embedded objects.*

Table 2 lists a section of our atom metadata. Note constraint 455 for atom 123 – this is a constraint which is used for fault tolerance in say for example the case of flash crowds. Here the session monitor receives processor utilisation from the respective monitor, when it detects that the utilisation rises above 90% the server agent is required to run on a different node (one of *node1* or *node2* delivering *Page1.html*). The different node could be a under-utilised machine in the typing pool that contains a replica of *Page1.html*. The action SWITCH indicates to the session manager that not only should the Adaptivity Manager save the data state, but also the processing state, as it is this that is about to migrate. That is, essentially the whole service-agent is mobile making the Adaptivity Manager's task more complex.

| Cons-traint | Atom | Constraint logic |
|---|---|---|
| 450 | 123 | Select *BEST* (node1.*Page1.html*, node2.*Page1.html*) |
| 455 | 123 | If processor-util > 90% then SWITCH ((node1.*Page1.html*, node2.*Page1.html*) |
| 595 | 153 | If bandwidth > 30 < 100 Kbps then *BEST* ( node1.*videohalf.ram(time parms)*, node2.*videohalf.ram(time parms)*, node3.*videohalf.ram,(time parms)*) else node3.*videosmall.ram(time parms).* |

**Table 2 Snapshot of Atom metadata for Patia Webserver showing Constraints**

## 6 Our Vision

As earely as 1991 Mark Weiser had a vision the next (3rd) wave of computing would no longer consist of mainframes or networks of PCs, but that computing would become ubiquitous and pervasive [30]. The key distinction between this type of computing and today's systems is not really that of extreme distribution of data and processing over many small devices (though it is an important difference), but that computing should be *calm*. Weiser defined calm to be systems that did not require the user to ask what the computer could do for them; rather the system works out how it can best serve the user. This is essentially user empowerment. Such a system cannot expect the user to carry systems housekeeping or technical support therefore systems must become more self-aware, adaptive and essentially self-healing. Consequently calmness is a fundamental challenge to us, as the added intelligence does not come without a cost. Therefore we believe that the system must be dissolved into its elementary elements, which are fine-grained and augmented with 'intelligence'. This is analogous to natural systems where cells and organisms evolve (usually) to best suit environments and demands.

It is our conjecture that future systems must be composed of well-defined file-grained self-reflective components and that this extra functionality does not impair performance. To this end we have investigated an alternative OS (*Go*!) composed of components and has shown that not only is it possible to dissolve the OS kernel in a fine-grained way, but that to do so can actually improve both the speed and space requirements of the OS. In parallel we have been examining the addition of rules and constraints to component-based systems architectures such as audio and Webservers (*Kendra* and *Patia* respectively) and experimenting with the very many levels of adaptivity [23, 26]. Both these systems are in relatively embryonic form at the moment and we are only beginning to look at how we can combine both elements of research aided by ADLs. Having said that this work is beginning to highlight some open issues and areas that need further examination. Some of the more important are listed below:

- For systems composing of thin/small clients and servers interacting with highly distributed data and with varying degrees of replication for fault-tolerance, novel physical data structures are required.
- More work on adaptive data operators like the Join algorithms mentioned in section 2 is required. Currently this work has mainly focused on Joins for aggregation queries, however this needs to be broadened.
- Self-learning systems must be lean and tractable.
- Continuation of work on reconfigurable specification languages for components data is required. Current ADL's are a good start but they are either domain specific or implementations reconfigure far too slowly. Further, typically ADL's with adaptive capacity only focus on closed-adaptive systems so more work on systems that learn from previous adaptations are required.

However the single most important observation we have noticed is concerned with the nature of the adaptive systems themselves. This was first noticed in our Kendra system, which is a simple adaptive audio server [23]. Thus far we are beginning to observe that our system has the potential to behave in a similar fashion to that of biological systems. That is, with finer-grained systems there are lots of (tuning) variables, many feedback loops to drive the adaptivity etc., and it was quite difficult to

attribute elements of performance to the processing and decision-making carried out by the system. The Kendra system was relatively simple, how much more complex is a truly adaptive system whereby thousands of self-aware components should eventually find their best solution to a given task (perhaps evolving the solution as it goes along)?

An opinion, specific to the nature of the CIDR conference, is that for this research to be fruitful holistic systems research is required. This is true for not only database system research but computer science in general. Primarily due to the funding of short-term projects, much systems research has been focused on in a narrow but deep way e.g. a single join algorithm, caching etc. This means we see very few publications showing how architectures or technologies that essentially have been around for 20+ years can be scrapped and replaced. Perhaps there is an argument that overly deep and focused work has detracted researchers away from the art of computer science producing exciting and genuinely new systems. However we acknowledge that the work presented in this paper certainly does not answer all the questions it highlights, we hope that the nature of the kind of architecture presented would encourage holistic research and specifically target the bigger problems e.g. cost of reflection, interfacing components etc.

Finally, back in 1983 Boral predicted the demise of the Database Machine (DBM) and he was right to an extent [5]. DBM architectures based on specialised hardware or tightly coupled to specific specialised machines were always going to be problematic. However as componentisation dissolves the DBMSs architecture into components and that this is integrated, without boundaries, with the operating system (which in turn only activated the components that are required by the DB function, thus tailoring the architecture down to the metal), means that at *that instant* the system becomes effectively a Database Machine but potentially without the problems of standardisation and portability of the past.

## 7 Acknowledgements

Though there is a single author on this paper there are many who have contributed to it from brainstorming to carrying out PhD's or Post Docs under my supervision. Thanks to David A. Bell for forcing me to look at database components and their performance back in 1988 (my PhD). Further thanks to (and in no particular order): Tim Wilkinson, Patty Kostkova, Steve Crane, Paul Howlett, Alan Messer, Akmal Chaudhri, and Paul Brown. These people I thank for their hard work and stimulation. In particular I thank Greg Law who's PhD showed that we *could* have fine-grain components 'down to the metal'.

Finally, thank you to the reviews for their very helpful comments where helped shape some of the arguments more clearly.

## 8 References

1. Avnur R. Hellerstein J. M. 'Eddies: Continuously Adaptive Query Processing', *Proc. ACM SIGMOD Int. Conf. Management of Data,* Dallas, TX, May 2000.
2. Batory D., Barnett J., Garza J., Smith K., Tsukuda K., Twichell B., Wise T., 'Genesis: An Extensible Database Management System', in *IEEE Transactions on Software Engineering,* IEEE November 1988
3. Boncz P., Kersten M.L.,'Monet: an impressionist sketch of an advanced database system', *BIWIT'95-. Basque Int. Workshop on Information Technology* Spain, July 1995
4. Booch G., *Software Components with Ada*, Benjamin/Cummings, 1987
5. Boral H., DeWitt D. J. 'Database Machines: An Idea Whose Time Passed? A Critique of the Future of Database Machines'. *International Workshop on Database Machines (IWDM*), pp 166-187, 1983
6. Chaudhuri S. Weikum G. 'Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database System', *Proceedings of the 26th International VLDB conference*, September 2000
7. Carey M., DeWitt D., Graefe G., Haight D., Richardson J., Schuh D., Shekita E., Vandenberg S., 'The EXODUS Extensible DBMS Project' in *An Overview, Readings in Object-Oriented Databases*, eds Zdonik S. and Maier D., Morgan-Kaufman, 1990
8. Cheung W.H., Loong A.H.S. 'Exploring Issues of Operating Systems Structuring: from Microkernel to Extensible Systems', Department of Computer Science, The University of Hong Kong. *Operating Systems Review, Vol. 29, No. 4,* October 1995,
9. Engler D.R., Kaashoek M.F., O'Toole J.W., 'Exokernel: An Operating System Architecture for Application-Level Resource Management', MIT Laboratory for Computer Science. *Proceedings of the 15th ACM SOSP, Colorado, USA*, December 1995.
10. Gabber E, Small C., Bruno J., Brustoloni J. and Silberschatz A., 'The Pebble Component-Based Operating System', *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, USA, June 6-11, pp. 267-282. 1999
11. Garlan D., Monroe R. T., Wile D., 'Acme: An Architecture Description Interchange Language', In *Proc. of CASCON '97*, November 1997.
12. Geppert A., Dittrich K.R*, Component Database Systems* Eds Dittrich K.R., Geppert A., Morgan Kaufmann, ISBN 1-55860-642-4, 2001.
13. Haas L., Chang W., Lohman G., McPherson J., Wilms P., Lapis G., Lindsay P.., Pirahesh H., Carey M., Shekita E., 'Starburst Mid-Flight: As the Dust

Clears', In *IEEE Transactions on Knowledge and Data Engineering,* IEEE, March 1990
14. Haas P. and Hellerstein J. 'Ripple joins for online aggregation'. In *Proc. of the ACM SIGMOD Conference,* 287-298, 1999.
15. Hellerstein J. M., Haas P.J., Wang H.J. 'Online Aggregation', Technical Paper, IBM (1997)
16. Heytens M., Listgarten S., Neimat M., Wilkinson K., 'Smallbase: A Main-Memory DBMS for High-Performance Applications', HP Labs Technical Report (March 1994)
17. Ives Z G., Levy A Y., Weld D. S., Florescu D., Friedman M. 'Adaptive Query Processing for Internet Applications'. *IEEE Data Engineering Bulletin vol 23 no 2*, pp 19-26, 2000
18. Jaeger T., Liedtke J., Panteleenko V., Park Y., Islam N., 'Security architecture for component-based operating systems'. *ACM SIGOPS European Workshop*. 9/98.
19. Kostkova P., Murray K., Wilkinson T., Component-based Operating System, *In 2nd Symposium on Operating Systems Design and Implementation* Seattle, October 1996
20. Law G. A new protection Model for Component-based Operating Systems, PhD Thesis, City University, School of Informatics, London, 2001
21. Law G., McCann, J.A., 'A New Protection Model for Component-Based Operating Systems', In *Proc. of IEEE Conference on Computing and Communications,* Phoenix, Arizona, Feb. 2000
22. Magee J., Dulay N., Eisenbach S. Kramer J., Specifying Distributed Software Architectures. In *Proc. of the Fifth European Software Engineering Conference*, Barcelona, 1995.
23. McCann J.A., Howlett P., Crane J.S., 'Kendra: Adaptive Internet System', *Journal of Systems and Software*, Elsevier Science, Volume 55, Issue 1, 5 November 2000, pp 3-17.
24. McCann J.A., Crane J.S., 'Component DBMS Architecture for Nomadic Computing', *16th British National Conference on Databases (BNCOD'98),* Cardiff, Springer-Verlag, 1998, pp 175-176.
25. Oreizy P., M. Gorlick M., Taylor R. N., Heimbigner D., Johnson G., Medvidovic N., Quilici A., Rosenblum D. S., Wolf A. L.. 'An Architecture-Based Approach to Self-Adaptive Software'. *IEEE Intelligent Systems,* vol. 14, no. 3, pp. 54-62, May/June 1999.
26. Patia: Adaptive Webserver. http://www.doc.ic.ac.uk/~jamm/research/patia.html
27. Singhal V., Kakkad S., Wilson P.,' Texas: An Efficient, Portable Persistent Store, Persistent Object Systems': In *Proc. Fifth International Workshop on Persistent Object Systems*, September 1992
28. Thomas, D., 'P2: A Lightweight DBMS Generator', PhD Thesis, University of Austin, Texas (1998)
29. Urhan T., Franklin M.J., Amsaleg L., 'Cost-based query scrambling for initial delays'. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1998.
30. Weiser M., 'Some Computer Science issues in Ubiquitous Computing', *Communications of the ACM*, vol. 36, no. 7, pp 75-84, July 1993
31. Wilschut A. N., Apers P.M. G.: 'Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proc. First International Conference on Parallel and Distributed Information Systems (PDIS),* pp 68-77, 1991