An Architecture for Scheduling of Services in a Distributed System

G. P. A. Fernandes*and I. A. Utting

Computing Laboratory, University of Kent, Canterbury, Kent CT2 7NF, UK
Tel: +44 1227 764000 x7754, Fax: +44 1227 762811
email: {gpaf,iau}@ukc.ac.uk

Abstract

Advances in technology, giving increasing importance to information service networks, and the increasing use of personal workstations are two factors which permit the construction of distributed applications running on a large set of interconnected systems. Existing frameworks for the development of distributed applications only provide for manual scheduling of application services, in particular there are no policies provided to distribute the load of the system across its nodes. This paper presents an architecture for the scheduling of services in a distributed system, that incorporates policies to enable distribution of load among the nodes of the system. A scheduling scheme and an infrastructure to support scheduling are also discussed.

Keywords: management, scheduling, load distributing, ODP, distributed systems

1 Introduction

Distributed computing systems are becoming critical for the working of many enterprises. These are expected to contribute to the financial and operational well-being of the organisations which rely on them. Management of these heterogeneous hardware and software environments, in order to supply and maintain the services required, is one of the most difficult problems facing computer users today.

Distributed systems are composed of a large number of autonomous processors, each having its own local resources. These systems promise high performance, availability and extensibility at low cost. The total computing capacity of such a system can be many times greater than conventional computing systems. However, to realize these benefits, a good *load-distributing* scheme is essential in order to allocate the considerable processing capacity available so that it is used to its fullest advantage. Load distributing attempts to improve system performance by redistributing the workload submitted to the system by its users and fully utilise the available computing resources.

*Work supported by JNICT Program *PRAXIS XXI* (Portugal) under grant No. BD/2804/93

Different platforms are now available for building distributed applications (*e.g.* APM's ANSAware, OMG's CORBA and OSF DCE), however, these platforms do not provide facilities for automatic management of those applications. In particular, performance management by distributing the load across the nodes of the distributed system is not included. When a new instance of a service is created, it is located on a node statically determined without regard to system's performance. The system may end up with some of its nodes overloaded, while others are idle or with a very small amount of processing. Furthermore, it is the service's developer or the service's client responsibility to choose a node to locate a server for that service. To fully realise the benefits of distributed platforms, the node where services are created should be transparent to the user of the platform, and its selection based on the system state.

This paper presents a management infrastructure suitable for scheduling and distributing services across the nodes of a distributed system. The main goal is to optimise the use of resources, improving overall performance of servers, applications and distributed systems. Besides trying to improve the system's performance by distributing the load, the infrastructure must be able to manage the resources available in order to fulfil the requirements of the application services. Another benefit of this approach is to release the application programmer from the job of allocating services to nodes, providing a degree of transparency to the existence of several nodes.

Section 2 presents a short survey of load distributing strategies. A very brief overview of ANSAware, the platform chosen to demonstrate the infrastructure, is given in section 3. The proposed management model is described in section 4. Section 5 introduces work related to the area. Finally, conclusions and some issues for future work are discussed.

2 Load Distributing strategies

Initial work in load distributing concentrated on designing strategies to allocate user processes to processors in a static manner, prior to execution. These algorithms require prior knowledge of processes behaviour, and may take poor assignment decisions as the state of the nodes in the system is not considered when making such decisions.

Dynamic algorithms can outperform static algorithms by using system-state information to improve the quality of their decisions. A load-distributing algorithm must ensure that it has a reasonably up-to-date view of the system's state. This could be achieved by using a *centrally-located allocator* processor (Zhou 1988), which would periodically be sent load information from all other processors and would make decisions based on the last-known state of the system. This central component introduces a potential bottleneck, limiting the scale of the managed system. Thus, a fully-distributed solution is favoured. This approach adds complexity to the system in dealing with possibly out-of-date information. On the other hand, care must be taken that cooperation between different processors does not overload the communications mechanism used to exchange load information.

A dynamic load-distributing algorithm has several components (Harget and Johnson 1990, Shivaratri, Krueger and Singhal 1992):

- *Information policy*: an information policy decides when, where and which information about the state of nodes in the system is to be collected. Load measurement is calculated locally on each machine (generating a *load index* for the machine), and then communicated through the network to its peers. Information accuracy must be sufficient to minimise the impact of out-of-date state information, however, frequent load information exchange will introduce additional overheads, leading to performance degradation (Harget and Johnson 1990).

- *Transfer policy*: determines whether a node is suitable to participate in a task transfer, either as a sender or a receiver. Many proposed policies are based on *thresholds*. When a new task originates at a node, if the load at that node exceeds a given threshold the node becomes a sender. On the other hand, if the load falls below another given threshold, the node can be a receiver for a remote task. An alternative to threshold policies are *relative* transfer policies which consider the load on a node in relation to the loads on other system nodes.

- *Selection policy*: once the transfer policy decides that a node is a sender, a selection policy selects a task for transfer. The simplest approach is to select one of the newly originated tasks. This type of migration is called *non-preemptive*, since it involves only tasks that have not begun execution. On the other hand, *preemptive* migration involves transferring a partially executed task. The need for collecting the task's state and transferring it to the receiver makes preemptive transfers complex and expensive. Although preemptive migration adapts more quickly to changes in processor load, this overhead means that it is not often used for load sharing.

- *Location policy*: the responsibility of a location policy is to find a suitable transfer partner (sender or receiver) for a node, once the transfer policy has decided that the node is a sender or a receiver. *Polling* is the most commonly used decentralised policy. A node polls another node (or several nodes) to find out whether it is suitable for load sharing. In a centralised policy, a node contacts a specified node, which acts as a *coordinator*, to locate a suitable node for load sharing. This coordinator collects information about the system (coordinated by an information policy) and the transfer policy uses this information at the coordinator to select receivers.

Adaptive algorithms are a special class of dynamic algorithms. They adapt their activities by dynamically changing their parameters, or even their policies, to suit the changing system state. Even when the system is uniformly so heavily loaded that no performance advantage can be gained by transferring tasks, a non-adaptive dynamic algorithm might continue operating, thus incurring overhead. To avoid overloading such a system, an adaptive algorithm might instead reduce its load-distributing activity when it observes this condition.

3 ANSAware overview

The design of the model presented in this paper was based on the Open Distributed Processing (ODP) Reference Model (ISO 1995a) and on ANSA (Advanced Networked Systems Architecture), an architecture closely related to this model. The distributed system manager model described in this paper is currently being implemented in ANSAware (APM 1989, APM 1991a, APM 1991b), a framework that implements the ANSA architecture and supports the development of distributed applications. Although ANSAware does not provide a management system, it includes some mechanisms which can be used for management implementation.

A typical distributed application is constructed from several cooperating objects. The basic building block of ANSA is a *service*. A service is an information handling function for the processing, storage, or transfer of information. A service is provided at an *interface*. Objects that use a service are called *clients*; objects that provide a service are called *servers*. An object can be both client and server of many services simultaneously. Services are divided into *application services*, which are specific to the task to be performed by the application (*e.g.* booking service for airline reservations) and *architectural services* which are generic to a wide range of tasks and have been identified by ANSA (*e.g.* *trading* service).

A *capsule* is the unit of autonomous operation in ANSAware. It is a collection of zero or more objects. Every object is instantiated within a capsule. An *object* is a collection of zero or more interfaces, that allow clients access to the service(s) provided by that object. An *interface* is a collection of zero or more operations and an *operation* is a specification of argument and result types, together with an implementation.

The *trader* is an object that provides a service which accepts and stores service *offers* from potential providers and hand out this information on request to potential clients. Servers that wish to advertise the services they provide, may do so by exporting them (registering their interface instances with appropriate attributes) to the trader. Clients may locate the services which they intend to use by importing offers (looking up interface instances) from the trader.

The *factory* provides a service for the dynamic creation and destruction of capsules. Once a capsule has been created, it in turn provides a service for the creation and destruction of objects within itself, and the creation and destruction of interfaces within an object. The *node manager* is a service responsible for the creation, simple monitoring and destruction of services on a single node. By using the local factory, it provides mechanisms for the static and dynamic creation of capsules and objects. The node manager also includes a database for monitoring and managing the services instantiated. This database contains a description of each service, identified by an *alias*. Each alias can be given various attributes that specify how it is to be managed. Once an alias for a given service has been installed in the node manager, the service may be activated (a service instance created) statically, as a result of an invocation from the command line of the node manager `RunAlias` operation, or dynamically, in response to an import of a previously posted *proxy* offer.

4 A Distributed System Manager

In this section we present a model for management of distributed systems, addressing in particular the issue of distributing the workload submitted to a distributed system by its users. Distributed scheduling is used in order to locate a new service on the most appropriate node, taking into account the current state of the system and the quality of service (QoS) requirements of the service.

4.1 Monitoring

The state of a distributed system must be observed, in order to make appropriate service allocation decisions and adapt these decisions to changes in the state of the system. Monitoring is an essential means for obtaining the information required about the components of a distributed system. It can be defined as the process of dynamic collection, interpretation and presentation of information concerning objects or software processes under scrutiny (Sloman and Moffett 1990).

A centrally-located allocator – *Distributed System Manager (*DSM*)* – is responsible for taking decisions in order to determine to which node in the system each service will be allocated. To determine if a node is suitable to instantiate a service, the DSM has to compare the QoS requirements of the service with the resources provided by the node. Information about the resources available on each node is gathered. This information includes load indexes giving, for instance, the amount of processing provided by the node, memory available and communication bandwidth. Information concerning specific hardware devices could also be collected. For instance, a multimedia service that requires some media device (*e.g.* an ATM host adapter) should be instantiated in a node that provides that facility.

The DSM makes placement decisions based on its last known state of the system. This information, stored by the DSM, is updated by the monitoring information it receives from node managers. The *Node Manager* is an entity local to a node, responsible for the management of that node and reporting monitoring information to the DSM. The node manager monitors and controls the services instantiated on the node and collects information about the resources available.

4.2 Scheduling

The scheduling strategy implemented is controlled by the DSM. This strategy is composed of an algorithm and input parameters to this algorithm. The parameters can be changed by an administrator, according to the priority that wants to be given by the DSM to certain resources when distributing the load. A scheduling algorithm must have a reasonably up-to-date view of the system's state. Having the DSM as a central unit, to which monitoring information is sent by all node managers in the system, can create a bottleneck in the DSM. This overhead can be reduced by sending load information to the DSM on demand.

Demand-driven information transfer also brings other benefits. Not all the node managers have to be polled, thus reducing communication overhead. When the DSM is first started, it polls all nodes under its control and uses the information received to initialise the monitoring information it stores concerning the load of nodes. After initialisation, the DSM issues requests for monitoring information only when it is trying to find a suitable location for a service. In this way, the information the DSM has about the load on the node it chooses to allocate a service to is always updated just before the allocation. Considering the service requirements, the DSM selects the nodes that can satisfy those requirements, and polls those which were less loaded the last time the monitoring information was updated, to check if they are still able to provide the same resources. However, if a long time elapses since the last request to activate a service, a node that was heavily loaded and had no more activations allocated to it might now be less loaded.

Thus, it may be worth taking that into account and polling that node too. To implement such policy, the DSM stores service history. After receiving the information from the polled nodes, the DSM selects the one that best satisfies the service requirements. If the optimum QoS level cannot be granted, the DSM has to notify the client that requested the service and eventually negotiate new QoS settings.

A purely centralised solution is not very reliable, since the failure of the central entity could cause failure of the entire management system. A solution to this problem may be based on the approach used in MICROS for processor allocation (Tanenbaum 1992). Instead of having one single entity responsible for global management, as in a centralised approach, there could be a *federation* of managers, each responsible for a group of nodes.

All newly created services are instantiated by the DSM upon request by the trader. Then, the DSM selects a suitable location for the service and asks the local node manager to instantiate that service. The management interactions between the DSM and the node manager are illustrated in figure 1.

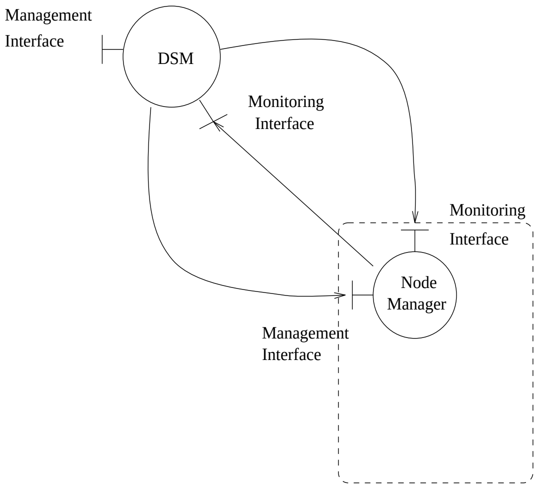

Figure 1: Management interactions.

Once services have been instantiated, preemptive migration could be considered for dynamic load balancing, but the costs associated with this operation would have to be accounted for. When an overloaded node is running out of resources, the local node manager may issue an alarm to inform the DSM. The DSM would then have to decide whether or not a service should be migrated from that node.

4.3 Infrastructure

Some of the concepts used in this section to describe an infrastructure for the management of a distributed system were adopted from ANSAware. In order to support the next generation of distributed systems, and in particular to ensure interoperability, the infrastructure is presented in the computational viewpoint of the ODP Reference Model.

There is a node manager on each node, responsible for managing that node and reporting information to a DSM. The node manager is an extension of ANSAware's node manager, with some alterations to overcome the limitations of that service in the area of system performance monitoring and load-distributing decisions.

In order to be managed, all services must be started dynamically via the DSM. In ANSAware's node manager, aliases for each service are installed statically from the command line. Once installed, these services may be statically or dynamically started via the node manager. This idea was adapted for the DSM, requiring dynamic installation of aliases through the invocation of an operation on a service interface of the DSM. The installation of aliases is in this case a function of the DSM and not of the node manager. After aliases have been installed, the DSM posts proxies for the services with the trader and instantiates them dynamically whenever a client tries to import them.

When a client asks the trader for a service, the reference in the trader for that service is a proxy exported by the DSM. The actual activation/deactivation of a service in a node is the responsibility of the factory local to that node.

The procedure for activating a service is illustrated in figure 2.

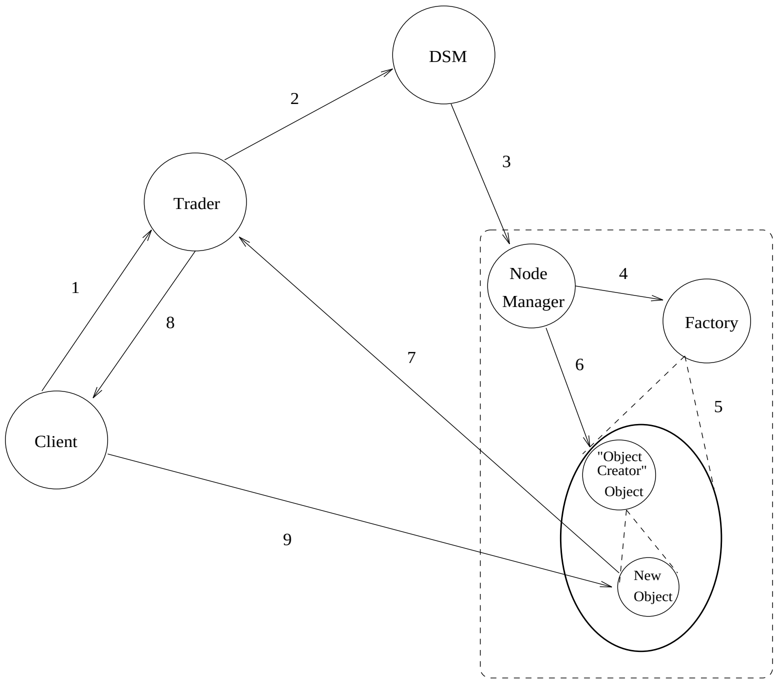

Figure 2: Activation of a service.

When a client needs a service, it tries to import a reference to that service from the trader (1). The trader recognises the offer as a proxy and forwards the import to the DSM (2). The DSM selects a suitable node on which to instantiate the service and asks the corresponding node manager to instantiate it (3). The node manager asks the factory to create a server capsule (4). The factory instantiates the capsule and returns its interface reference to the node manager (5). The node manager calls an operation on the capsule interface to instantiate the object that provides the service required (6). Once the service has been instantiated, it exports itself to the trader (7). The trader returns the reference received from the service to the client (8). The client can now interact with the service (9).

When a node is running out of resources, it may issue an alarm to inform the DSM. The DSM may then decide to *passivate* (the server is terminated and a *snapshot* is stored so the server can be later re-activated) or *migrate* one of the services that have been activated in that node. Another situation where passivation could be considered is when a high-priority service activation is requested and a suitable node cannot be found. It may be necessary to passivate a lower-priority service in order to provide the resources required by the high-priority service.

5 Related Work

There has been a lot of research on the conceptual basis and description of a management architecture for the design of a distributed management system (IDSM and Sysman 1993, Sloman, Magee, Twidle and Kramer 1993a). The emphasis, in this architecture, is on the use of domains to group managed objects and partition the management structure to cope with very large scale inter-organisational distributed systems.

A number of organisations, namely X/Open, IEEE POSIX and NMF, are trying to define frameworks to integrate management. The majority of work is still focused on the definition of objectives and requirements (Hungate and Fernandes 1995). However, an Open Distributed Management Architecture (ODMA) (ISO 1995b) is already being developed by ISO as a specific interpretation of the ODP Reference Model, to provide an architecture reference model for the management of distributed resources, systems and applications.

One approach to managing distributed systems developed in ANSAware has recently been proposed by Kuepper, Popien and Meyer (Kuepper, Popien and Meyer 1996). Their main goal was to extend the ANSAware trader to include updating of dynamic service attributes. With this strategy they intend to optimise the use of servers, already executing, and provide a client with a reference to the most suitable server. It is the trader's responsibility to select that server. However, this approach does not involve decisions concerning selection of nodes on which to allocate services by considering the resources available and the service requirements.

The field of load distributing and load balancing has been target of many interesting research efforts (Shivaratri et al. 1992, Harget and Johnson 1990). Most work has been focused on investigating algorithms to share the workload submitted to the system among the existing processors. However, no widespread or commercially used strategy has evolved.

6 Conclusions and Future Work

The work reported in this paper concentrates on a particular aspect of distributed systems management, which can be included in one of the distributed systems management areas defined by OSI management (ISO 1989) – performance management. An approach for global scheduling and load distributing has been presented. It is the system's responsibility to decide where services must be started, and hence distribute the load in order to maximise the system's performance. Those decisions are made considering the resources available and the service's requirements. Migration of services between nodes could also be considered, but the associated costs and overheads incurred would have to be weighed against the benefits.

The model described will be specified using ZEST (Cusack and Rafsanjani 1992, Wezeman and Judge 1994), an extension of the formal specification language Z, developed by PROST Objects to facilitate a clear, unambiguous specification of managed objects. From this model tests will be generated in order to validate the implementation.

The management infrastructure presented could be used to support the inclusion of management in existing platforms for developing distributed applications, such as ANSAware and CORBA.

Management of a distributed system should itself be distributed to reflect the distribution of the system being managed (Sloman, Magee, Twidle and Kramer 1993b). Large distributed systems will inevitably be partitioned into multiple domains with different responsibilities, reflecting geographical, technological or organisational structures. Different management purposes, such as security, accounting and fault management, also require the structuring of management to permit multiple coexisting views of the managed objects. Domains provide a flexible means of specifying boundaries of management responsibility and authority in a distributed system. In the model presented, a DSM is the agent responsible for the management of the nodes which are members of one domain. One domain could have more than one DSM, for instance a *committee* of DSMs, to prevent against faults. This is an issue for further development.

References

APM (1989). ANSA: An Engineer's Introduction to the Architecture, *Release TR.03.02*, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK.

APM (1991a). ANSA: A System Designer's Introduction to the Architecture, *Release RC.253.00*, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK.

APM (1991b). ANSA: An Application Programmer's Introduction to the Architecture, *Release TR.017.00*, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK.

Cusack, E. and Rafsanjani, G. H. B. (1992). ZEST, *in* S. Stepney, R. Barden and D. Cooper (eds), *Object Orientation in Z*, Workshops in Computing, Springer-Verlag, pp. 113–126.

Harget, A. J. and Johnson, I. D. (1990). Load balancing algorithms in loosely-coupled distributed systems, *in* H. S. M. Zedan (ed.), *Distributed Computer Systems: theory and practice*, Butterworths.

Hungate, J. and Fernandes, G. (1995). Distributed Systems: Survey of Open Management Approaches, *Technical Report NISTIR 5735*, NIST - National Institute of Standards and Technology, Distributed Systems Engineering, Computer Systems Laboratory, Technology Administration, U.S. Department of Commerce, Gaithersburg, MD 20899, U.S.A.

IDSM and Sysman (1993). IDSM/SysMan Management Architecture, *Technical report*.

ISO (1989). ISO/IEC 7498-4 | CCITT REC. X.700 Information Processing Systems - Open Systems Interconnection - Basic Reference Model - Part4: Management Framework. OSI Mgt.

ISO (1995a). ISO/IEC DIS 10746-3 | ITU-T Rec. X.903 Part 3 Open Distributed Processing - Reference Model: Architecture, ISO/IEC JTC1/SC21/WG7. ODP-RM.

ISO (1995b). ISO/IEC JTC1/SC21 N 9947: Open Distributed Management Architecture, Working Draft 3. ODMA.

Kuepper, A., Popien, C. and Meyer, B. (1996). Service Management using up-to-date quality properties, *in* Schill, A. and Mittasch, C. and Spaniol, O. and Popien, C. (ed.), *Distributed Platforms*, Chapman & Hall, pp. 447–459. Service Mgt using the ANSAware Trader.

Shivaratri, N. G., Krueger, P. and Singhal, M. (1992). Load Distributing for Locally Distributed Systems, *Computer* **25**(12): 33–44.

Sloman, M. and Moffett, J. (1990). Managing Distributed Systems, *Technical report*, Imperial College, Department of Computing.

Sloman, M., Magee, J., Twidle, K. and Kramer, J. (1993a). An Architecture for Managing Distributed Systems, *Proceddings of the Fourth IEEE Workshop on Future Trends of Distributed Computing Systems*, IEEE Computer Science Press, pp. 40–46.

Sloman, M., Magee, J., Twidle, K. and Kramer, J. (1993b). An Architecture for Managing Distributed Systems, *Technical report*, Imperial College.

Tanenbaum, A. S. (1992). *Modern Operating Systems*, Prentice Hall.

Wezeman, C. and Judge, A. J. (1994). Z for Managed Objects, *in* J. P. Bowen and J. A. Hall (eds), *Eight Annual Z User Workshop*, Springer-Verlag, Cambridge, pp. 108–119.

Zhou, S. (1988). A Trace-Driven Simulation Study of Dynamic Load Balancing, *IEEE Transactions on Software Engineering* **14**(9): 1327–1341.