Distributed Software Engineering

Invited State-of-the-Art Report

Jeff Kramer

Department of Computing,
Imperial College, London SW7 2BZ.
(jk@doc.ic.ac.uk)

Abstract

The term "Distributed Software Engineering" is ambiguous[1]*. It includes both the engineering of distributed software and the process of distributed development of software, such as cooperative work. This paper concentrates on the former, giving an indication of the special needs and rewards in distributed computing. In essence, we argue that the structure of these systems as interacting components is a blessing which forces software engineers towards compositional techniques which offer the best hope for constructing scalable and evolvable systems in an incremental manner. We offer some guidance and recommendations as to the approaches which seem most appropriate, particularly in languages for distributed programming, specification and analysis techniques for modelling and distributed paradigms for guiding design.*

1. Introduction

Distributed processing provides the most general, flexible and promising approach for the provision of computer processing. Interconnected workstations are widely used for local processing, to support user communication for interaction and cooperation, and to provide access to shared facilities and information. Conventional and special purpose processors are interconnected to support a wide range of applications from chemical plants to cars, from stock market trading to campus meal reservations.

Why are distributed systems so attractive? The answers are as multifarious as the applications. The users of computers, the information they require and provide and the applications themselves are often physically distributed. To match these needs, both the hardware and software can be designed and constructed in a flexible modular fashion - as interconnected and interacting components. Particular resources, services and information can be accessed across the network and shared among the system users. Some seek to exploit the potential for improved availability by the use of replication and the removal of single failure points. Others seek performance gains by improving the response time through local processing or the throughput by the use of parallel processing. Thus distributed computing offers advantages in its potential for improving availability and reliability through replication; performance through parallelism; sharing and interoperability through interconnection, and flexibility, incremental expansion and scalability through modularity.

However, to gain these benefits, we must cope with the issues that distributed computing raises. The interactions between the concurrent components give rise to issues of non-determinism, contention and synchronisation. Component separation and autonomy gives rise to issues of partial information and partial failure. These issues demand that we adopt effective engineering methods and tools. Our techniques must avoid constraining the resultant software unnecessarily by the use of conventional sequential or centralised designs but take cognizance of and exploit the component-based nature of these systems. Software engineering itself must be extended and adapted to address these distribution issues: hence Distributed Software Engineering (DSE) [Shatz 89].

It is not possible or sensible to cover the whole area of DSE. What is presented is the author's unashamedly biased view of the field, hopefully conveying some insight into what is so special and exciting about distributed systems. The paper provides a brief overview of some of concepts in distributed systems as background material for later discussions. We then examine the particular need for appropriate concurrency and distribution models and the associated specification and analysis techniques. Software development methods are

[1] This ambiguity is exploited by covering both aspects in the DSE Research Section at Imperial College.

briefly discussed together with the need to consider and utilise distributed algorithms. Finally we briefly discuss the construction of a successful, very large, international information distribution system.

2. Distributed Systems Overview

One of the major advances towards distributed computing was the provision of standard and reliable serial communication networks. This lead to the provision of *Networked Systems*: autonomous computers with independent operating systems connected to the network for information transfer. These systems exhibit no single system master, and interaction between the individual systems is essentially for file transfer rather than close cooperation.

Distributed Systems are essentially an extension of these networked systems except that the interaction is expected to include closer cooperation [Sloman 87]. This implies some form of system-wide management and control. Such control is complicated by the fact that there is no global coherent view; the overall system "state" is partitioned and distributed across its constituent computers. It must therefore be capable of making decisions based on partial information. Terms such as "open" and "loose-coupling" are used to denote the ease with which distributed systems support interaction and yet ensure autonomy and independence of their constituent components.

A Distributed Environment

The most common use of distributed systems is the provision of shared services which are provided transparently by the cooperative effort of local and remote processors. The provision of services by server processors is illustrated by a simple example from office automation. Figure 1 gives an outline of such a system in which users access the system services from terminals and workstations. Different services are offered by servers, which provide access to a variety of facilities such as file stores, printer services, mainframe processing and database services. Some services are provided locally at the site and accessed via a LAN, others at remote sites accessible via a gateway to the Internet.

User and application programs act as the clients to servers in a *client-server* relationship. Client processes make requests for service and are allocated a server process which is relinquished after use. A server may well use the services of other servers in the performance of its function. A name server (or trader) provides a directory service to the system to enable services to be registered

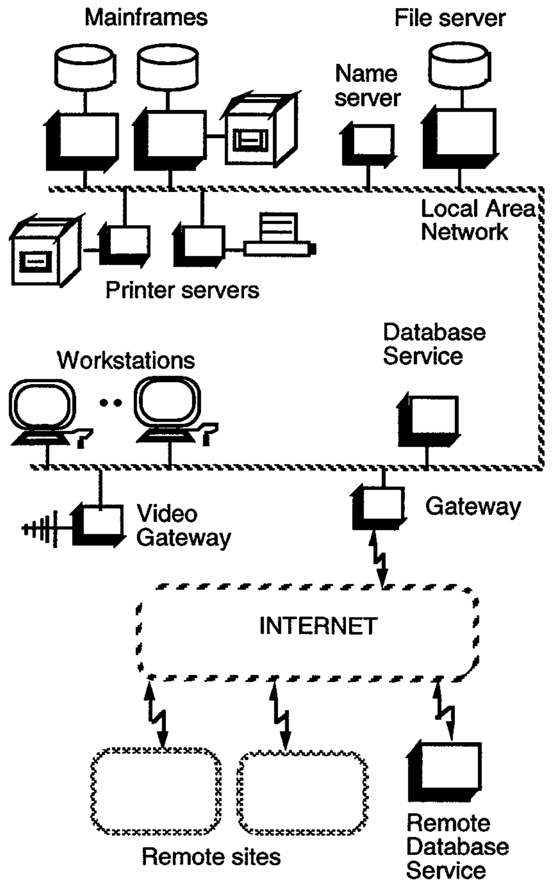

Figure 1 - Distributed Office Automation

and subsequently found by users and application programs. Clients may thus dynamically join systems, locate and use required services and then depart.

Distributed System Infrastructure

A *Distributed Operating System* (DOS) provides the system-wide management of distributed services. It utilises local management for local services and cooperates with the DOS component of remote nodes to provide access to remote and distributed services, such as file transfer, remote execution, security services (authentication and encryption), synchronisation services for atomic transactions and global scheduling of resource access. One of its primary responsibilities is to coordinate resource usage, providing resource sharing, protection, and recovery. It aims to provide the "open" access required for distributed use. Transparency requires coherent and uniform naming and access conventions to permit local and remote resources to be accessed without knowledge of their physical distribution. Hence, users generally specify their required service rather than the particular server. DOS components are usually organised as cooperative but autonomous in that they can refuse requests and remain mutually suspicious in order to protect their domain.

The advantages of the client-server approach and the use of shared services in a distributed system (cost, extensibility, performance, availability) are such that many general computing systems are now provided in this way. They are generally partitioned functionally into a number of services, each of which may be provided by a number of servers. It is usually easier to provide centralised services, with replication for fault tolerance, than to try and provide the more general and robust decentralised solutions.

For instance, the Athena distributed computer system implemented at MIT [Chanpine 90] provides computing services to around 10,000 users. In 1990 it reportedly had about 1000 workstations in 40 clusters. It is based on the client-server model, providing file servers, postscript printers, name servers (Hesiod), post office servers for electronic mail, notification servers (Zephyr) and authentication servers (Kerberos). Moira provides a centralised management service, keeping track of the hardware and software configuration, allocation, and access control lists. These services are replicated for fault tolerance. The services are built on and compatible with the Berkley UNIX programming and user interface. The workstations act as dataless nodes. The software is loaded and cleared at the start and end of each user session, and the application processing is generally performed on the workstation itself. Hence, although services are generally remote and distributed (eg. the mail service), the processing is generally local.

At a different level, the Mach kernel (Carnegie Mellon) [Rashid 86, Accetta 86] is designed as a specialised kernel of a DOS to support both tightly-coupled and loosely-coupled multiprocessors. It provides multiple threads (tasks) in clusters (large virtual address space) with synchronous message-based communication via ports. The kernel is designed to support a variety of programming interfaces, including Berkley UNIX. The Andrew File System [Morris 86], also used in Athena, was developed for use with Mach. Other examples of research work on kernels and DOS facilities are the V kernel (tested for lightweight processes and interprocess communication experiments) [Cheriton 88], Amoeba (threads and objects with capabilities for protection) [Mullender 90], Chorus (threads, clusters and ports to provide UNIX-like distributed processing) [Rozier 87] , and the ANSA (Advanced Networked Systems Architecture) platform (an object-oriented platform for open distributed processing) [Oskiewicz 88, ANSA 91].

The research is mature and much of the work on distributed systems infrastructure has been adopted in standards. OSF (Open Software Foundation), a consortium of more than 300 companies, have produced and made available an implementation of UNIX based on the Mach kernel. OSF/DCE, their distributed computing environment, provides facilities for threads, remote procedure calls (rpc), a distributed directory service, a time service, distributed file system and a security service [OSF 92]. Other standards organisations are also active. OMG (Object Management Group) promote CORBA (Common Object Request Broker Architecture) supporting distributed objects and object interaction [OMG 91]. ISO/ODP (Open distributed Processing) offer support for rpc communication and trading [ISO 93]. CCITT provide the X.400 [ISO 88] and X.500 [ISO 89] standards for communication and directory services. In addition, the computer vendors, such as SUN, HP, and DEC, offer products with features similar to DCE and CORBA.

How should one use these facilities to construct distributed services and programs? What languages, methods and tools are available? Unfortunately, there is less consensus as to the answers to these questions. Nevertheless, we can examine some of the requirements and assess the current state-of-the-art.

3. Distributed Programming: the need for languages

Although distributed systems can be constructed using the infrastructure facilities described above, this is reminiscent of the use of assembly languages. High-level languages with access to distributed programming features are needed to provide a safer programming environment. In this section we briefly overview some of the basic language requirements.

Distributable Components

In order to provide software which can be distributed and execute concurrently in a distributed environment, we need modular software components (cf. objects in OOP) which do not share memory but interact by some form of communication. In its simplest form, a component can be a simple sequential process (eg. tasks in Conic [Magee 89]); however many systems provide for some form of shared data component (cf. cluster) which can contain a number of threads or light-weight processes (resources in SR [Andrews 88]). Note that, if multiple threads/processes do share data in the cluster, it is necessary to provide some means for synchronising such access. The main principle is that the component *encapsulates* resources in the form of data or devices.

Each component should provide an *interface* for interaction with other components. This interface describes the services offered and required, including the type of the information received or transmitted by the component. In order to provide *context independent* components - capable of being used in different circumstances and interacting with different components - they should not directly address any external entity, but rather accept and make calls at local service points offered or required at their interface. In heterogeneous environments in which components of different types communicate, data transforms can be invoked. A separate Interface Definition Language (IDL) is frequently used as a common language for interface descriptions and checking, including interaction points and data typing (eg. Matchmaker [Jones 85], MLP [Hayes 87], IDL [ANSA 91] and CORBA IDL [OMG 91].

Components should be defined as a type (cf. class) from which instances can be created. A distributed program is then constructed as a structure or *configuration* of interconnected instances of components. Instances must be mapped (allocated) to the physical structure of interconnected computer nodes. For instance, the logical configuration given below could be described as follows, where the *performance* component type requires service *data,* the *analyser* offers *in* and requires *out* and the *reporter* offers *print*: :

```
use  performance, analyser, reporter;
                                        {component types}
inst MONITOR:performance;
     ANALYSER: analyser;
     PRINTER: reporter;
                                        {instances}
bind MONITOR.data -- ANALYSER.in;
     ANALYSER.out -- PRINTER.print;
                                        {interconnection}
```

Configuration information is often specified in the distributed programming language, or as operating system commands. However work on Conic [Kramer 85, Magee 89], REX [Kramer 92], Regis [Magee 94], Polylith [Purtilo 92] and Durra [Barbacci 93] suggests that there is a benefit in clarity and system management if a separate language (such as that used above) is provided specifically to express configuration structure [Kramer 90]. Component instantiation and interconnection (binding) can take place at configuration time before allocation and also at run time to provide the greatest flexibility.

Client-server binding is usually required to be performed at run time, according to client need and service availability, and is not predefined in a configuration description. As mentioned above, a client uses a binding to a name server, service broker or trader to locate a server providing a particular service. For their part, servers register their interfaces, names, addresses and type/quality of service with the broker. Clients and servers thus use the broker as an intermediary to perform run time binding. For instance, the ANSA platform provides a trader [ANSA 91] for this purpose. Although this provides a highly flexible form of interconnection, there are obviously execution overheads in registering and deregistering services, and in performing the lookup and dynamic bind operations.

Interaction

Distributed components communicate in order to cooperate and synchronise their actions. The underlying communication service provides for messages to be sent from one component to another. This service is offered to the application in a number of different forms, from simple unblocked (asynchronous), unidirectional communication analogous to the datagram, to the blocked (synchronous), bidirectional remote procedure call (rpc) analogous to the conventional procedure call. The choice of communication primitives determines how easily systems can be designed and implemented, given that they execute in an environment which is subject to delays and failures.

As mentioned, the remote procedure call (rpc) is the most widely adopted and accepted approach for general client-server interaction. It is provided by most operating systems for remote communication, and embedded in languages such as Concurrent CLU [Cooper 88] and SR [Andrews 88]. It is a natural outcome of the desire to provide a distributed mechanism analogous to the conventional procedure call. The notion is that programmers need not be aware of the distribution but can use familiar mechanisms whether or not the invoked procedure is local. An object (or instance of a form of abstract data type) has encapsulated data and offers a number of access "procedures" to manipulate the data (cf. object methods in OOP), for example:

```
object x;
  <<local data>>
  proc service1 (in pv1, pv2, ...out pr1, pr2, ...);
    begin ... end
  proc service2 (in pv1, pv2, ...out pr1, pr2, ...);
    begin ... end
           ....
end.
```

A process instance of a procedure is created for each occurrence of invocation. Thus there can be concurrent access to the data and some form of internal synchronisation may be usually required. As an alternative to process creation on receipt of a remote call, a component may instead *rendezvous* with the caller. The component again offers a number of entries or ports at its interface, different entries for different types of messages and/or different types of service.

The aim of the rpc is to provide uniform call semantics in that the behaviour should be the same for remote as for local procedures. This is termed *exactly once sematics.* However, this transparency cannot be achieved in the case of server crashes. One possible semantics is *at least once* in which case the system keeps retrying the call until the server recovers or an alternative service is available; however this may result in multiple invocations. Another alternative is *at most once* in which case the system does not retry but reports the failure. Neither of these is ideal, and it is usually necessary for programmers using an rpc to be aware of the possibility of failure and take measures to deal with it.

Many other communications primitives and language facilities have been proposed and are in use, providing support such as atomic transactions and persistent objects as in Argus [Liskov 88]. A comprehensive survey of work in the language area is provided in [Bal 89].

4. Specification and Analysis: the need for models

As mentioned, distributed systems introduce concerns which are akin to concurrent systems: non-determinism, synchronisation and properties such as safety and liveness. These are complex issues and we are still groping to find the right formalisms to provide us with the ability to reason about the behaviour of such systems. Distribution introduces further concerns due to the separation and autonomy of its components and the latency and failure of the interactions. Properties such as timeliness and robustness (dependability) are introduced.

Two further concerns which are prevalent and important in distributed systems are scale and dynamics. Although individual subsystems or parts thereof may have been separately designed and constructed, and may not themselves be large, the attraction and utility of distributed computing arises from the ability to compose them and support interaction. Therefore, scalability of associated techniques of specification, design and analysis cannot be ignored. Furthermore, many of these interactions arise dynamically, according to need and/or failure and recovery.

Thus our view of distributed systems is of a complex community of components where use of appropriate models is crucial for sound design and analysis. Models are used both to provide an abstract specification of the system behaviour and also to support some form of analysis to increase our confidence in the adequacy of the specification. We do not discount the need for prototyping and testing. These are essential complementary techniques. Without empirical results we could not have any confidence in the accuracy of our models. Nevertheless in this section we concentrate on modelling.

Note that we believe that compositionality is the key property, both because it is appropriate for this component/interaction view of distributed systems and also as the means for handling scale. It is akin to a constructive approach for distributed system design and construction, based on the composition of components.

Levels and Aspects

We must be prepared to model our system at many levels of description (figure 2). Abstract high-level descriptions may hide many of the concurrency and distribution issues and concentrate on properties such as consistency. On the other hand, design level descriptions must be prepared to explicitly model the component-based structure and interactions, and deal with properties of safety, liveness, timeliness and so on. Implementation level descriptions may include further distribution decisions thereby allowing more detailed analysis of properties such as failure, response or performance.

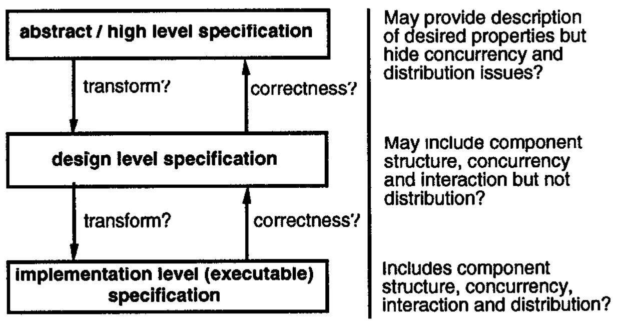
abstract / high level specification
transform?
correctness?
design level specification
transform?
correctness?
implementation level (executable) specification
May provide description of desired properties but hide concurrency and distribution issues?
May include component structure, concurrency and interaction but not distribution?
Includes component structure, concurrency, interaction and distribution?

Figure 2 - Levels of Specification

Further, in order to deal with this complexity, we must recognise that no single formalism or model will suffice for every aspect. Even at a single level, it is unlikely that a single model will be adequate. Rather, it is essential that each is selected to accurately reflect the particular system property which requires examination, and to abstract away from others. Typically each model will capture and support analysis of a particular property (functional behaviour, performance, failure, ...) but find it difficult to reflect others.

For instance, if we consider a particular design level, a system could be described as a particular composition of components, each of which would provide a specification of its intended behaviour (functional, performance or failure characteristics) as illustrated in figure 3 [Kramer 90, 93]. An associated analysis technique should then be provided for each. Correctness can be considered as the use of the analysis to show satisfaction of the properties of the higher level specifications.

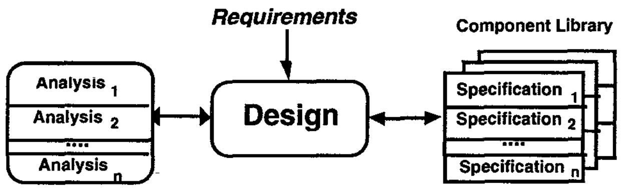

Figure 3 - Corresponding Specification and Analysis for each Model

Modelling: *Which model should we choose?*

The assumptions under which the model is valid are obviously important. In his excellent paper, [Schneider 93] provides examples of alternative models which illustrate this for distributed systems.

• Synchronous or asynchronous?

An asynchronous system makes no assumptions about process speed or communication delays. Although this leads to robust designs, this generality has an associated cost in that the designs tend to be more complex or expensive. All synchronisation must be explicitly provides by protocols between the participants. On the other hand, a synchronous system assumes processes whose relative speeds are bounded, and communication with bounded delay. This permits time to be used as another source of information in a design. For instance, the non-appearance of a message can be used to provide information only if the system is synchronous. The validity of the assumption may depend on the particular platform to be used for implementation or on the desire to suppress certain concerns for that model.

• Which failure model?

In distributed systems faults are attributed to processing components or communication channels. Fault tolerance relies on the ability to replicate those components which are likely to fail. The assumed behaviour of a faulty component is crucial in determining how the system will behave. The simplest and least disruptive is failstop [Schneider 84] where component failure is equivalent to halting, and that failure is detectable by other components. This means that other components can perform actions in the knowledge that the failed component will not. Note that, as Schneider points out, if components in an asynchronous system could not detect the failure, they could also not distinguish between failure and a component which is continuing but merely slow. The most complex and disruptive failure model is that of Byzantine failure [Lamport 82] where component failure can produce any arbitrary behaviour. In order to tolerate t failures, we need only have t+1 replication in the failstop model yet require 2t+1 for the Byzantine case.

What characteristics should we look for in our choice of models? [Cheung 94a]

• Logical (descriptive) or behavioural (operational) approach [Ghezzi 91]?

Those taking a logical approach describe behaviour as a set of axioms. System behaviour is given by the maximal set of event sequences that satisfy these axioms. This approach supports logical reasoning in that a system is said to possess a certain property P if P is derivable from the axioms. Specifications can be combined using logical connectives such as conjunction [Zave 93]. This property-oriented approach is appropriate for specifying rules and constraints, and is more appropriate for high-level abstract specifications such as requirements. However, it does not reflect the component/interaction view relevant for design level specifications of distributed systems.

The alternative is a behavioural (or constructive) approach where the system is modelled by an abstract program. System behaviour is given as the set of all event sequences that can be generated by the program. Behaviour can be examined by analysis of those traces or by simulation (execution). Abstract programs can be formulated as the parallel composition of subprograms. This component-based approach can closely reflect the component/interaction view and is thus considered more appropriate for design specifications.

Consequently, the sensible approach would appear to be to adopt the logical approach for requirements and the behavioural for design, and to verify designs by their ability to satisfy the required logical properties.

• Static or dynamic analysis?

Analysis is the means by which we gain greater confidence in the adequacy of our designs. Dynamic analysis can be used to show the presence of errors through model execution, but not their absence. Static analysis is performed without execution and aims to show the absence of errors by satisfying prescribed safety and liveness properties. However, complete coverage of all properties is generally difficult. Therefore, static and dynamic analysis should be considered as complementary, and an ideal model should be able to support both.

• Compositional or global analysis?

In general, global analysis is computationally too expensive for any but the simplest of system. Analysis using logical reasoning is undecidable for the first order calculus and state-based models generally suffer from state explosion. We therefore require that our models support compositional and incremental analysis. This bounds the scope of the analysis at any one time and helps to contain problems such as state explosion. Furthermore, distributed systems are generally designed and constructed by decomposition into a hierarchy of simpler components and/or by composition from elementary components. Hence incremental analysis mirrors the development process and can be employed by the designer at each stage in the design process. Hierarchical and compositional approaches provide the soundest means for coping with scale, and offer the potential for dealing with dynamic structures through composition.

It therefore seems that logical, property-based specifications do not scale well, and that it is easier to build and analyse the abstract machine specifications advocated by the behavioural approach.

• Automation?

The complexity of analysis techniques suggests that manual use is not practical in general. Models must at least offer the hope (if not the fact) of automated support. In addition, one should support different forms of analysis according to need. Approximate analysis can be used to compromise accuracy for tractability. In this way, an efficient approximate analysis technique could be readily used to detect some errors at the early design stages when designs are tentative and error prone. Exhaustive compositional analysis can be left to the later stages of development when the designs are more stable and the need for design confidence is greater.

• Other aspects?

Obviously, in order to make the model accessible to software engineers, we would like it to be simple and familiar. This can be further enhanced if the model also provides a graphical notation.

Do current approaches support our required characteristics?

Temporal logic [Pnueli 86] is a powerful formalism for describing and reasoning about sequences of events ordered in time. Distributed system properties are specified by constraining those events which must occur in a particular order. Safety and liveness properties can be readily expressed using the temporal operators 'always' and 'eventually' respectively. Although there is recent work towards dynamic analysis through executable specifications [Gabbay 89], analysis generally relies on theorem proving. In most cases, this requires a great deal of ingenuity and is not easily supported by automation [Clarke 86]. In addition, temporal logic does not support the compositionality we require.

Petri nets are one of the most popular graphical techniques for specifying concurrency and interaction. Invariants and reachability are used for analysing safety and liveness properties. Analysis can be dynamic or static, supported by automation. However, standard Petri nets are not compositional [van Eijk 88] and thus fail to scale.

Process algebras, such as CSP [Hoare 85], CCS [Milner 89], ACP [Bergstra 85] and the π-Calculus [Milner 91] support most of the characteristics that we require for modelling distributed systems. They support the desired behavioural component/interaction view using notions of communicating processes. Processes are modelled as algebraic expressions representing sequences or a non-deterministic choice of actions. Communication is modelled via synchronous interaction at an action. Parallel composition of processes is supported as is compositional analysis. A possible disadvantage of process algebras is their use of interleaving semantics rather than "true" concurrency; however this is generally not a problem as we rarely need to distinguish the two.

A recommendation?

We favour Labelled Transition Systems (LTS) which provide the underlying semantic model for the process algebras. In particular, LTSs also satisfy our automation requirements, and are based on the familiar and graphical notation of state machines. One can introduce explicit buffering processes to model a particular degree of asynchrony. More details and evidence of the utility of this approach are given in [Cheung 94] in these proceedings. Furthermore, as advocated by Weihl [93], one can introduce non-determinism to model the possibility of failures or communication delays.

5. Design: the use of Distributed Paradigms

The aim of distributed systems design is to identify the distributable components and their interactions which together satisfy the system requirements. How is this achieved? This obviously involves a process of decomposition with composition as the means for showing requirement satisfaction and of actual system construction. Are there formal methods which are useful, or must one rely on informal methods?

Formal approaches using refinement have been successfully applied to small problems and to derive distributed algorithms (often post hoc). For instance, UNITY and its associated refinement techniques are clearly described in [Chandy 88]. However, there is seems to be no evidence of their use in large, realistic cases. One approach which has proved helpful in guiding initial design is the use of invariants to express certain safety properties. This is then distributed across the system as a set of local constraints [Carvalho 82].

Less formally, there are a number of systematic techniques which use decomposition coupled with heuristics. A number of OO design techniques [Booch 91, Rumbaugh 91] are helpful in identifying objects (cf. components) and classes; however they are not particularly aimed at producing distributable objects. They tend to ignore issues such as object to process allocation, communication delay and failure, and independent object failure. Gomaa [93] has gone some way to providing guidance as to the design of concurrent and distributed systems, giving some criteria for forming distributable components. In addition, most approaches support the use of some form of interaction diagram to describe component interaction for particular scenarios (cf. a trace of communication actions in the process algebras).

Design is a creative and intuitive process. As we move from level to level, new information and concerns are necessarily introduced (see figure 2). Hence it is not surprising that correctness preserving transformations and refinements find it difficult to cope with the task. We therefore advocate informal design together with rigorous analysis to provide the feedback required to either identify sources of error or to provide further evidence as to the validity of our design. Each successful design analysis acts to increase our confidence in our design. The more facets that can be analysed (functional behaviour, timing, fault analysis, ...) through static and dynamic means, the greater our confidence. Furthermore we advocate incremental design. This is the ability to incrementally extend and evolve designs (and the corresponding systems) by the addition, replacement and modification of its components and structure [SEJ 93, Kramer 93].

In addition to the feedback provided by analysis, the informal decomposition process should be guided by a sound knowledge of the various distributed programming paradigms and techniques. These encapsulate the experience of distributed programming. For instance, a simple heuristic is that, unless state information is specifically required to be replicated for reasons of fault tolerance or local response, such replication should be avoided. Why? State replication introduces the need to maintain consistency between the copies which will entail overhead or, if not handled well, could introduce possible erroneous behaviour.

What paradigms are commonly used in distributed systems?

As mentioned, current distributed systems are often partitioned functionally into a number of services, each of which may be provided by a number of servers. Clients (either users or components which require some service) make requests to servers. Servers may themselves require other services to perform their functions, and so in turn act as clients to other servers. This client-server relationship is common in distributed environments, most often based upon the use of the remote procedure call (rpc) for client-server interaction. It is often easier to provide centralised services with replication for fault tolerance than to try and provide the more general and robust decentralised solutions. This is not to say that all distributed systems must follow this 'centralised' structure; just that it has been found to be appropriate in many applications.

Decentralised paradigms use an associated logical structure which supports the required interactions. The simplest of these are the *ring* algorithms for assigning some privilege. Components are connected together to form a logical ring so that, for the algorithm, each component only communicates with its neighbours. A circulating token is generally used to which some privilege (such as exclusive access) is associated. The token circulates round a ring giving every component some privilege at some time.

Diffusing computations use a hierarchical or tree structure to reach consensus decisions of global concern such as deadlock, termination or commit algorithms. Those components involved in the decision process are formed into a tree structure where a component only communicates with its ancestor or descendants. A control message is initiated by one component (the root or environment), and the control "diffuses" through all components in the tree. Each component can only issue a message to its descendants if it received one, and can only reply to its ancestor when it has all the descendent replies.

Consistent event ordering is necessary for consistent decision making. A partial ordering of events can be determined by *timestamping* using local, logical clocks in each process [Lamport 78]. Local clocks increase their values according to each event of sending a message or receiving one, and correct their clocks by the rule that a send must precede the corresponding receipt. This timestamping is consistent with causal ordering which "captures all the essential ordering information needed to describe the execution" [Birman 93]

What properties should our designs possess?

In producing distributed designs,there are a number of properties which act as signposts to a sound design [Raynal 88].

• Symmetry and resilience to failures

For a particular algorithm or coordination activity, the more symmetric the participants, the less likely there is to be a single point of failure. For instance, if one component has a special role which cannot be taken over by any other participant, then the participants are asymmetric (different code and role) and less robust. If all participants have the same code then, even though one may play a different role at a particular time, some other component could take over that role if necessary.

• Connection properties

Our designs should require the fewest properties of the connections between components. For instance, as mentioned in the section on modelling, we would like to be able to design for asynchronous systems and assuming any failure mode. This would result in resilient algorithms which could use unsophisticated networks. In practice, we often try to support failstop failure and provide some bounds on the asynchrony of a connection.

• Local states

As far as possible, individual components should be designed so as to be able to make decisions based on local knowledge. This reduces the message traffic and improves resiliency. In practice this is often a compromise between reducing response or overhead. Response can be reduced at the expense of overhead if components are kept uptodate with the information required before the point of decision making, while the the reverse is true if a component is designed to seek out the information when required.

There is still much to be learned in the design of distributed programs, but much guidance can be obtained by adopting and adapting proven paradigms [Andrews 91]. This is apparent where there are particular concerns such as reliability, and the need arises for process (component) groups with group communication which preserves causal ordering [Birman 93].

6. Conclusions

As described, the advocated approach to distributed system design and construction is one which reflects the component/interaction view of distributed systems. It is a combination of informal but informed design, and extensive use of models which support specification and analysis. In particular, the favoured techniques make extensive use of composition, during design, analysis and construction. This seems to provide the right balance, favouring rigour for formality where the latter is not practical.

Can this approach be used for all systems? Consider an example of a very successful system which, it appears, was never rigorously specified, analysed or modelled. It is a fascinating example of a system which has evolved by making excellent use of a number of different but mostly standard and proven technologies to provide "a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents": the World Wide Web (W3) [Hughes 93]. It was first proposed in 1989 to allow information sharing within an international community in High Energy Physics. The W3 project merges the techniques of networked information and hypertext, and is currently the most advanced information system deployed on the Internet. At the end of 1993 it reportedly had well over 500 Web servers.

For scalability, W3 was designed without any centralised facility and no central control. Anyone who wishes to publish information need only provide a server; anyone who wishes to read data need only run a client. Clients communicate with servers on the Internet. W3 supports and uses the numerous different information retrieval protocols (FTP, Telnet, NNTP, WAIS, gopher, ...) as well as the data formats of those protocols (ASCII, GIF, Postscript, DVI, TeXinfo, ...) to access existing FTP files, WAIS databases, news articles and gopherspace and new hypertext files. It provides a consistent hypertext user interface using a new protocol (HTTP- hypertext transmission protocol) and a new data format (HTML - hypertext markup language) both geared toward hypermedia. A Web client can use a hypertext link to retrieve documents or pictures which are held remotely.

These can in turn be used to access further information, hence providing a vast "web" of information. Links are held as URLs (Uniform Resource Locators) which specify the method of access (file, http, ftp, news, gopher, ...), the internet address of the server, and any local access information.

The Web employs the client-server model, copes with heterogeneity by providing translators, and is dynamic, scalable, decentralised and very widely distributed. It is an excellent example of the synthesis of technologies and the composition of different components to great effect. It has evolved and increased in size by incremental steps, comparable to an organic body. However, we should note that it is not a critical application; if it were, should the approach be very different? Perhaps not, provided that we ensure that the base components and technologies are sound, that the system is developed incrementally and that the architecture is amenable to evolution. The essential ingredient for critical applications is the ability to model and analyse the composite system structure to provide the necessary feedback and assurances of satisfactory behaviour.

The other face of DSE......?

Finally, as mentioned at the start of this paper, the other face of distributed software engineering is the process of distributed development of software. The concepts and techniques described above have much to offer in guiding distributed cooperative software development. Rather than adopting a centralised approach, its seems to be sensible to mirror the development process with its multiple participants and partial products. Partitioning and distribution of the participants' viewpoints (cf. components) capture different aspects, formalisms, specifications, and stages in software development [Finkelstein 90, Kramer 91]. Information transfer and consistency checking (cf. interactions) can be conducted directly between the viewpoints [Nuseibeh 93, Finkelstein 93]. This offers all the advantages of independent development, scalability, redundancy, and flexibility which are apparent in distributed computing.

Acknowledgements: I gratefully acknowledge the advice and cooperation of my colleagues in the Distributed Software Engineering Research Section, and the financial support provided by the DTI under grant ref: IED4/410/36/002.

References

[Accetta 86] Accetta M., et al, "Mach: A New Kernel Foundation for UNIX Development", Proc. of Summer USENIX Conf., 1986, pp 93-112.

[Andrews 88] Andrews G.R., et al, "An Overview of the SR Language and Implementation", ACM TOPLAS, Vol. 10, no. 1, Jan. 1988, pp 51-86.

[Andrews 91] Andrews G.R., "Paradigms for Process Interaction in Distributed Programs", ACM Computing Surveys, Vol. 23, No. 1, March 1991, pp 49-90.

[ANSA 91] "ANSAware 3.0 Implementation Manual", Document RM.097.00, APM Ltd, Cambridge, Jan. 1991.

[Barbacci 92] Barbacci M.R. et al, , "Durra: a Structure Description Language for Developing Distributed Applications", IEE/BCS Software Engineering Journal, (Special Issue on Configurable Distributed Systems), Vol. 8, No. 2, March 1993, pp 83-94.

[Bal 89] Bal H.E., Steiner J.G. and Tanenbaum A.S, "Programming Languages for Distributed Computing Systems", ACM Computing Surveys, Vol. 21, No.3, September 1989, pp 261-322.

[Birman 93] Birman K.P., "The Process Group Approach to Reliable Distributed computing", Communications of the ACM, Vol.36, No. 12, December 1993, pp 37-53.

[Booch 91] Booch G., "Object Oriented Design with Applications", The Benjamin/Cummings Publishing Company, 1991.

[Bergstra 85] Bergstra J.A. and Klop .W., "Algebra of Communicating Processes with Abstraction", Theoretical computer Science 37, 1985, pp 77-121.

[Budkowski 87] Budkowski S. and Dembinski P., "An Introduction to Estelle: A Specification Language for Distributed Systems", Journal of Computer Networks and ISDN Systems, vol. 14, 1987, pp 3-23.

[Carvalho 82] Carvalho O. and Roucairol G., "On the Distribution of an Assertion", ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Ottawa, 1982.

[Chandy 88] Chandy K.M. and Misra J., "Parallel Program Design : a Foundation", Addison-Wesley, 1988.

[Chanpine 90] Chanpine G.A., Geer D.E. and Ruh W.N., "Project Athena as a Distributed Computer System", IEEE Computer, vol. 23, Sept. 1990, pp 40-51.

[Cheriton 88] Cheriton D.R., "The V Distributed System", Comm. of the ACM, vol. 31, March 1988, pp 314-333.

[Cheung 94] Cheung S.C. and Kramer J., "An Integrated Method for Effective Behaviour Analysis of Distributed Systems", Proc. of 16th International conference on Software Engineering (ICSE-16), Sorrento, May 1994.

[Cheung 94a] Cheung S.C., "Tractable and Compositional Techniques for Behaviour Analysis of concurrent Systems", PhD thesis, Department of Computing, Imperial College, 1994 (submitted).

[Clarke 86] Clarke E.M., Emerson E.A. and Sistla A.P., "Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications", ACM TOPLAS, Vol. 8, No. 2, 1986, pp 244-263.

[Cooper 88] Cooper R.C.B and Hamilton K.G, "Preserving Abstraction in Concurrent Programming", IEEE Trans. on Software Engineering, Vol. SE-14, No. 2, Feb. 1988, pp 258-63.

[Finkelstein 90] Finkelstein, A., Kramer, J., Goedicke, M., "ViewPoint Oriented Software Development", Proc. of Third Int. Workshop on Software Engineering and its Applications, Toulouse, December 1990, 337-351.

[Finkelstein 93] Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B., "Inconsistency Handling in Multi-Perspective Specifications", Proc. of 4th European Software Engineering Conference, ESEC '93, Garmisch, Sept 1993.

[Gabbay 89] Gabbay D., "The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems", Technical Report, Department of

Computing, Imperial College, January 1989.

[Ghezzi 91] Ghezzi C., Jazayeri M. and Mandrioli D., "Fundamentals of Software Engineering", Prentice-Hall, 1991.

[Gomaa 93] Gomaa H., "Software Design Methods for Concurrent Real-Time Systems", Addison-Wesley 1993.

[Hayes 87] Hayes R. and Schlichting R., "Facilitating Mixed Language Programming in Distributed Systems", IEEE Transactions on Software Engineering, vol. SE-13, no. 8, Aug. 1987.

[Hoare 85], Hoare C.A.R., "Communicating Sequential Processes", Prentice-Hall, 1985.

[Hughes 93] Hughes K. , "Entering the World-Wide Web: a Guide to Cyberspaces", Report, Honolulu Community College, September 1993.

[ISO 88] ISO 10021/CCITT X.400 (1988), Message Handling Systems: Information Processing Systems - Text Communications - MOTIS.

[ISO 89] ISO 9594 1-8 / CCITT X.500-X.521, 1989.

[ISO 93] ISO WG7 93: ISO/IEC JTC1/SC21/WG7 Information Retrieval, Transfer and Management for OSI, Working Draft, June 1993.

[Jones 85] Jones M.B., Rashid R., Thompson M., "Matchmaker: An Interface Specification Language fro Distributed Processing", Proc. of 12th Annual Symposium on Principles of Programming Lnaguages, Jan. 1985, pp 225-35.

[Kramer 85] Kramer J., Magee J., "Dynamic Configuration for Distributed Systems", IEEE Transactions on Software Engineering, SE-11 (4), April 1985, pp. 424-436.

[Kramer 90] Kramer J., "Configuration Programming - A Framework for the Development of Distributable Systems", Proc. of IEEE COMPEURO'90, Tel-Aviv, Israel, May 90, pp 374-384.

[Kramer 91] Kramer, J. and Finkelstein, A., A Configurable Framework for Method and Tool Integration", European Symposium on Software Development Environments and CASE Technology, Königswinter, June 1991, Endres A, Weber H (eds), LNCS 509, Springer Verlag, 1991, 233-257.

[Kramer 92] Kramer J., Magee J. , Sloman M., Dulay N., "Configuring Object-Based Distributed Programs in REX", IEE Software Engineering Journal, Vol. 7, 2, March 1992, pp139-149.

[Kramer 93] Kramer, J., Magee, J., Ng, K., and Sloman, M., "The System Architect's Assistant for Design and Construction of Distributed Systems", Proc.of IEEE International Workshop on Future Trends in Distributed Computing Systems in the '90s, Lisbon, Sept. 1993, pp 284-290.

[Lamport 78] Lamport L., "Time, Clocks, and the Ordering of Events in a Distributed System", Communications of ACM, vol. 21, no. 7, 1978, , pp 558-565.

[Lamport 82] Lamport L., Shostak R. and Pease M., "The Byzantine Generals Problem", ACM TOPLAS, Vol. 4 No. 3, July 1982, pp 382-401.

[Lamport 90] Lamport L., "A Temporal Logic of Actions", Dec Systems Research Center, SRC Report 57, 130 Lytton Ave., Palo Alto, CA 94301.

[Liskov 88] Liskov B., "Distributed Programming in ARGUS", Comm. ACM, vol. 31, no. 3, March 1988, pp 300-312.

[Magee 89] Magee J., Kramer J., Sloman M., "Constructing Distributed Systems in Conic" IEEE Transactions on Software Engineering, SE-15 (6), June 1989.

[Milner 89] Milner R., "Communication and Concurrency", Prentice-Hall, 1989.

[Milner 91] Milner R., "The Polyadic π-Calculus: A Tutorial", Technical Report, ECS-LFCS-91-180, Dept. of Computer Science, University of Edinburgh, 1991.

[Misra 91] Misra J., "Loosely-Coupled Processes", Proc. of Parallel Architectures and Languages Europe (PARLE'91), Netherlands, June 1991, Springer-Verlag, LNCS 506, pp 1-26.

[Morris 86] MorrisJ.H., et al, "Andrew: A Distributed Personal Computing Environment", Comm. of the ACM, vol. 29, March 1986, pp 184-201.

[Mullender 90] Mullender S.J., et al, "Amoeba: A Distributed Operating System for the 90's", IEEE Computer, vol. 14May 1990, pp 44-53.

[Nuseibeh 93] Nuseibeh, B., Kramer, J., and Finkelstein, A.,"Expressing the Relationship between Multple Views in Requirements Specification", Proc. of 15th IEEE Int. Conf. on Software Engineering (ICSE-15), May 1993, 187-198.

[OSF 92] "Introduction to OSF DCE", Open Software Foundation, Cambridge, USA, 1992.

[OMG 91] Object Management Group, "The Common Object Request Broker Architecture and Specification V1.1", OMG December 1991.

[Oskiewicz 88] Oskiewicz E., Otway D., Sventek J, "ANSA Testbench Manual", Report T1.28.02, ANSA Project, Cambridge, April 1988.

[Pnueli 86] Pnueli A., "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of current Trends", in Current Trends in Concurrency, Springer-Verlag, LNCS 224, 1986, pp 510-584.

[Purtilo 92] Purtilo J., "The Polylith Software Toolbus", ACM TOPLAS. (Computer Science Dept., University of Maryland, TR-2469, 1990).

[Rashid 86] Rashid R.F., "From RIG to Accent to Mach: The Evolution of a Network Operating System", Fall Joint Computer Conf., AFIPS, 1986, pp 1128-1137.

[Raynal 88] Raynal M., Distributed Algorithms and Protocols, John Wiley and Sons, 1988.

[Rozier 87] Rozier M. and Martins L., "The Chorus Distributed Operating System: Some Design Issues", in Distributed Operating Systems: Theory and Practice, (ed. Paker, Y. et al), NATO ASI Series, vol. F28, Springer Verlag 1987, pp 261-287.

[Rumbaugh 91] Rumbaugh J. et al., "Object-Oriented Modeling and Design", Prentice-Hall, 1991.

[SEJ 93] IEE/BCS Software Engineering Journal 8 (2), Special Issue on Configurable Distributed Systems, March 1993.

[Schneider 93] Schneider F.B., "What Good are Models and What Models are Good?", in Distributed Systems, (ed. Mullender, S.), 2nd edition , Addison-Wesley 1993.

[Schneider 84] Schneider F.B., "Byzantine Generals in Action: Implementing Fail-stop processors", ACM Trans. on Computer Systems, Vol. 2, No. 2, May 1984, pp 145-154.

[Shatz 89] Shatz S.M. and Wang J.P., Tutorial: Distributed-Software Engineering, Washington, DC: IEEE Computer Society Press.

[Sloman 87] Sloman M. and Kramer J., "Distributed Systems and Computer Networks", Prentice-Hall 1987.

[van Eijk 88] van Eijk P.H.J., "Software Tools for the Specification Language LOTOS", Cip-Gegevens Koninklijke Bibliotheek, Netherlands, 1988.

[Weihl 93] Weihl W.E., "Specifications of Concurrent and Distributed Systems", in Distributed Systems, (ed. Mullender, S.), 2nd edition, Addison-Wesley 1993.

[Zave 93] Zave, P. and Jackson M., "Conjunction as Composition", ACM Transactions on Software Engineering Methodologies, July 1993.