MEMORANDUM

10/8/94 - 10:02 AM

A/To *Empfanger*		*De/From* *Absender*	Jean-Michel Marcastel
Copie/Copy *Abschrift*		*Date* *Datum*	Saturday, October 8, 1994
Objet *Betreff*	InterStage		

Introduction:

InterStage is both a developement toolkit and a run-time environment. The development environment allows users to create applications that can be run over distributed systems using InterStage run-time environment. In turn the InterStage run-time environment is intended to replace the need for the use of RPC. One further advantage of using the InterStage run-time system is that it allows all system configuration to occur at run-time instead of at design time. It also allows the users to reconfigure the system while it is running; this is a feature not availible to users of the RPC model. InterStage is a system that allows users to easily create applications to take full advantage of all of UNIX's networking and concurrency abilities.

The InterStage development kit provides programmers with an easy way to create application's designed to be run over a network. It does this by separating a systems application specific functionalities from its lower level properties. All the properties allowing the application over to run over the network are handled by the run-time kit. This allows the programmer to design his applications as if it would run on a single machine and would handle requests sequentially rather than simultaneously. In order to run the application over a distributed system the programmer simply decides of a way of handling load distrubution and uses the appropriate parts of the InterStaga API.With the RPC model the programmer had to be aware that the application would be run on a distributed system while he created the application code.

Programming in InterStage requires a design approach different from the approach used with standard procedural or Object Oriented languages. While InterStage uses a object oriented/procedural language,C++, as its platform, conceptually the structure of an InterStage program is very different from the structure of a C++ program..InterStage programming is similar to co-routine programming in that, a program is divided up into a set of independent tasks that pass data around and are executed according to a schedule that is set at run time. InterStage uses the network capabilites of UNIX to handle the data passing instead of using shared memory locations and function calls to control program flow, the way co-rotine programming does. InterStage als takes the co-rotine concept one step further by allowing the tasks to run concurrent lyand to run using independent system resources.

InterStage applications are signified by two major concepts the method and the agent. A method is an abstraction of a procedure. It is a piece of code that performs some functionality. An agent is a process that has the ability to handle certain methods. This is an extension of object oriented concepts, but where object oriented concepts associate data with methods, an agent in InterStage associates an individual UNIX process with a method. Agents have the ability to communicate with each other by means of message passing. In general, message passing involves a request to invoke a method of another agent as well as any data that the method might need to perform its functionality. In InterStage message passing occurs in pairs; there is a request and a reply. This has a direct analogy to functional programming in that a request contains the name of the method to invoke and the formal parameters of a function, and the reply contains the return values of the function. A reply may contain any number of variables. This is necessary because, unlike standard programming where a function may have the ability to change the value of a parameter outside the scope of the function, variables in InterStage have a scope local to any given agents. The only way for different agents to share data is to send it in as messages.

InterStage Paradigm:

An essential philosophy in InterStage is that the application should be independent of hardware. The InterStage paradigm requires that an application run equally as well on a single machine as it would if its processes were spread over a group of LANs. This requires that the fact that an application is run on a single machine, a local area network, or distributed over a group of local area networks be completely transparent to the application. The InterStage developers kit, through its API, contains tools that provide the application programmer a simple way of making the network configuration transparent to an application. The only additional work required of the programmer is to decide on a overall strategy for dealing with a network configuration, allowing him to choose the proper tools in the InterStage API.

InterStage is by its nature an object oriented system. Much of the terminology that applies to object oriented programming applies to InterStage. While programming, one creates agent types which encompass the possible states an agent can be in as well as the types of requests that it can service. At run time one instantiates using the InterStage run time facilities. Like object oriented programming languages, InterStage enables the use of inheritance and polymorphism but there is no standard implementation for using these things.

The object oriented nature of InterStage gives rise to two general types of agents: agents whose instances are identical and agents whose instances may be in different states. Agents with identical instances are the easiest to deal with; an agent needing a service provided by another agent that has many instances can send its request to any agent that exists. The use of this type of agents is primarily for load distribution and as such it is probably best to choose randomly from the list of instances that are in existence at any given time. An agent might also choose an instance based on other factors such as instance location or the load on an instance, although keeping track of the load on an instance is likely to add significant overhead to an application.

Agents whose instances are allowed to have different states are best used as dedicated agents. These agents should be dedicated to some other part of the system, such as another agent or to devices like printers and storage devices. By dedicating agents to other agents one can increase the efficiency of a process by utilizing concurrency, although concurreny in individual functionalities can be also be handled without the use of dedicated agents By dedicating agents to certain devices it is possible to implement a type of polymorphism where similar agents handle identical method types with different implementations. This would provide the programmer with a common public interface for different types of devices. For instance, it is possible to implements different storage strategies by using two different instances of one agent type to handle the different types of storage.

In order to facilitate the use of dedicated agents, a strategy for implementation must be developed because InterStage does not provide any predefined tools for this type of programming. One possible strategy would be to create an agent type with identical instances that would keep track of the dedications of each instance. This method is not perfect because there must be updates given to the managing agent and in a large system this can be quite time consuming. Another problem with implementing some types of dedicated agents is that it goes against the philosophy that the application should be independent of the configuration of the network. (Although this requirement is not essential to the creation of a functioning application, adhering to it as strictly as possible allows for the most flexibility at installation and run time)

Another major component of the InterStage paradigm is the fact that all applications are event driven. This means that code is executed because of the occurrence of events. The most important event in InterStage is the arrival of messages. This event trigger an agent to execute code that deals with the message. The application programmer is free to define other event types that will cause an agent to execute a piece of code. This style of programming is very different from standard functional programming and force the application writer to think in terms of chains of events.

Concurrency In InterStage:

There are fundamental diffuculties with using concurency in applications. Most notable is the problem of schedualing the execution of task in a concurrent system. Many different stratagies have been proposed to deal with this problem. InterStage uses the stragagy provide by the ACTOR model of concurrent computation which involves treating a computation as a chain of events. In essense the system schedules it self as a computation evolves. In InterStage communications occur in pairs which allows an agent to know when a request has been fulfilled. This fact is essential to chain of events concept. An agent starts a chain of events by sending a message or messages and then waits for a reply before it continues. Whenever an agent receives a message does whatever processing it can do (not concurcently) and sends requests for more services. It can then wait for a reply before continuing.

This model of computation structures the chain of events into a tree, with agents at the nodes of the tree and messages as the connection.. Each agent need only know about the agent who sent it the request, its parent in the tree, and the agents to which it sends requests, its chideren. Like any tree, this sort of communication tree is highly recursive; each node becomes the top of new tree and the top of a new chain of events. A message starts at the top of a tree and propagates down the tree until it reaches a terminal node where a chain of reply is begun. An agent will only send a reply when all its requests have been filled. This implies that scheduling need only be delt with at the agent level; the global notion of schedualing is represented by a chain of events. Each agents simply decides wether its need are best handled sequentually or simultaneoulsy and send its requests accordingly.

There is no reason that trees need be the only model of a computation. Other types of graphs will also suffice. But, while a send - reply concept is all that is need for scheduling with trees as a model, an additional device for scheduling is needed if an agents is to receive information from two different agents. This type of model also requires that agents now about the overall structure of the graph because two agents working on the same computation need to agree on a particular instance to which to send their results. Allowing a general graph as a model is not strictly necessary since all graph (I believe) can be mapped onto a tree that will visit every node of the graph, a graph to be better able to express a computation than a tree is some cases.

These two types of concurrencies may not work well together, because of the fact that system configuration occurs at run time. The second type of concurrency requires that certain agents be dedicated to each other so that one calculation is carried out by a group of agents while the first type of concurrency requires that there be repetition of agents to help to distribute the load of multiple, identical requests. The run time facilities of InterStage support configuration at the agent level, not at the level of groups of agents and some of the most obvious strategies for load distribution involve sending messages to random instances of an agent type. This will create problems with the use of agents that are dedicated to each other. The ability to dedicate agents to one another is enabled by InterStage but it is not supported; it also may be very difficult to accomplish agent dedication.

Anatomy of an Agent

At its most basic level an agent is just an object with a state and a list of methods. This state usually consists of a list of agents to which an agent can send messages as well as any agent specific state variables. An agent also contains facilities to accommodate a global message service. These facilities include a signal handler and an event scheduler. The signal handler handles UNIX signals and any agent can choose a subset of signals in which it is interested. In general the signal handler creates events in response to signals, and the event scheduler executes the events in the proper order. I/O signals, which inform the agent about the arrival of messages, are the most important signals, and, as such, an agents contains a list of the message types that it is capable of handling. Agents also contain code to be executed at startup and code that is executed at shutdown. The startup code can be used to start a chain of events.

Anatomy of an InterStage System:

An InterStage system consists of two separate parts: a user created application and a run-time system configuration. The user created application consists of a set of agent types. Each agent type handles a certain set of messages and a certain set of signals. The standard system configuration consists of one instance of a specialized agent type running on each machine. These specialized agents are responsible for instantiating the user defined agent types and for updating each agent instance about the current state of the overall system. The application programmer uses the InterStage API to connect these two separate parts.

InterStage API:

Note: This is only a brief introduction to the API. For a more complete description refer to the InterStage 2.1 Developers Guide.

An agent consists of the one instance of the *BServiceObject* type. This type gives the agent its behavior as an agent . Within the *BServiceObject* type, the InterStage API can be divided into two parts: InterStage facilities, and application specific facilities. The InterStage facilities are used to connect the application to the run-time InterStage system, which consists of a specialized agent, the mdbind agent, that keeps track of configuration of the system on the network. These facilities consist of three major functions. There exists a function, *bindRegisterClass()*, that is used to tell an mdbind agent the what type of agent an instance is. There is another function *bindExpressInterest()* that is used to tell mdbind that it would like to receive update about all instances of a certain agent type. Both these functions accept a string as an argument and this string should be a unique string used to identify the agent type. Finally there is a *classChangeCallback()* function. This function is called every time a change occurs in mdbins's global database. This function should be used to organize the agent's own personal database.

InterStage provides a routine and a class to facilitate the application specific message passing. There is a *BServiceObject* method *dispatchRequest()* which is used to determine how an agent will handle an incoming request, and there is the BDialogue class which is used to handle the actual message. The dispatchRequest method accepts an unsigned int, which represents the name of the request, as it argument and returns a BDialogue that is capable of servicing the request.

The BDialogue class has four methods that allow requests to be serviced. These methods exist in pairs. The first pair is the *handleRequest() - sendReply()* pair. Its function is to deal with incoming requests to an agent. HandleRequest accepts a request code, and a message in the form of an array of BER data structures and an integer representing the number of items in the array. The handle request method should do all of the processing to the method and issue an sendReply command with the return message. The other pair is the *sendRequest () - handleReply()* pair. This pair is used for the agent to send messages to other agents. The send request method accepts an agent address, this should be retrieved from the agent's personal communication list, and a message consisting of a request code, some BER data and an unsigned int representing the number of data items sent. The programmer can use the handleReply() method, which accepts the same parameters as the handleRequest() method, to do any processing that needs to be done on the reply data.

A BDialogue can only handle one communication at a time. This means that a different dialogue must be used for each incoming communication and a dialogue must wait for a reply to its requests before it generates another request. The easiest way to enable an agent to deal with multiple identical requests is to generate a new dialogue for each incoming request. This requires the use of additional memory for each request, and while each dialogue is a transient object, in a system where agents handle large numbers of requests this may not be an efficient solution. InterStage provides a flow of control method, the *busy* method to allow the agent to block a request. If an agent on allows itself to handle a fixed number of requests of any give type at a certain time, it can tell other agents that it is busy. An agent that receives a busy signal will hold the message and try again at a later time.

In order for a dialogue to make to multiple requests simultaneously it must create a sub-dialogue for each request. There are no standard facilities to accomplish this. The best way to handle sub-dialogues is to pass the constructor a pointer to the parent dialogue. The sub-dialogue can then inform the parent dialogue when it has finished and the parent dialogue can keep track of all the sub-dialogues and wait until all have finished to continue.

InterStage provides a compiler that facilites the generation of BER data structures. This complier translates ASN.1 code into C++ code. It generates both a header file which can be included in the files that use the data types, and a source file that can be compiled and linked to the agent code.

Using the InterStage API :

Programming an agent involves two steps. The first step is deciding what requests the agent will handle and how to handle those requests. This step is what makes an agent unique and consists of four separate parts. The first part is registering the agent with the *mdbind* agent. The agent should use the *bindRegisterClass()* method to tell the mdbind agent what class of requests it is able to handle. Often this step is accomplished using the *startup()* method of the BServiceObject. The second step is deciding the number of dialogue types that will be needed to handle all the possible requests. Often it is easiest to create a separate dialogue type to handle each different request. The third part is to write the *dispatchRequest ()* routine. This routine should return a dialogue of the appropriate type for any given request. The fourth and most important part is to write each dialogue. This work consists of writing a *handleRequest()* routine. If a dialogue needs services from other agents the programmer may have to write a *handleReply* method to handle the replies to its requests. If an agent needs to make requests of other agents the programmer should have the *handleRequest* method call the *SendRequest* method and either the *handleRequest* or *handleReply* should call the *sendReply* method to reply to the initial request. Note that one will find nowhere in a programmers code a call to the functions handleRequest of handleReply; the message service takes care of calling these functions after the receipt of a SIGIO.

The second step in creating a agent is to create the interface between the user's agent and the mdbind agent. This only needs to be done if the agent is going to sendRequests to other agents and is accomplished with the classChangeCallback method. This is a method that is called by the message service whenever a change in the global system configuration occurs. This method is passed a string and an AddressGroup. The string is the name of the agent type for which the change has occured and the AddressGroup is a list of all the instances of that agent type. In order to inform mdbind that agent would like updates it calls the method bindExpressIntrest() with the name of the agent type about which it would like information. The code in the classChangeCallback method should be written to allow the agent to organize its own personal database of the agents that are in existance.

Additional work must be done if an agent type is to be capable of communicating with the outside world, i.e. human users or non-InterStage devices or programs. This is made difficult by the nature of an event driven system. In an event driven system, some outside source is needed to generate an initial event, and once this initial event has been genterated it can be used to generate other events. Since InterStage uses signals to generate events, the best way to interface InterStage to the outside world is through the use of signals. In the case of a human interface this is easily accomplished using Xt Intrinsics, because X servers will send SIGIO's to a process on the occurrence of an event dealing with the X windows system. The InterStage devopement kit provides a simple interface between the Xt Intrinsics and the InterStage message service.

If one is not willing to use Xt Intrinsics, the human-InterStage interface becomes more complicated. Without using the Xt Intrinsics, the easiest way to generate an initial event it to use the startup method which generates a startup event. This event can be used to other events by calling code that will send messages to other agents thereby starting a chain of events. The problem with using this kind of interface is that a program can only issue one request. This occures because once an event, a startup event for example,begins to execute, no other events will execute until that event finishes. This means that the startup event can not be used for interactive communication since output generated by other events, in the message service, will not be done until the startup routine is complete. In brief, any loop placed used to generate repetitive requests, that is placed in an event, will make all the requests before it will deal with any of the replies.

This problem is not as serious as it may sound for InterStage provides a facility for generating command line programs. This facility is called the anonymous agent. The anonymous agent is an agent that is started from the command line, not from the mdbind agent. These agents are particularly well suited to handling single requests because the user can use the UNIX shell to generate multiple requests. Anonymous agents can also be used to start an Xt application that would allow the user to make multiple requests.

The Guts of an Agent:

Like any UNIX process, an InterStage process has both a signal handler and a main thread of execution. The signal handler is used by the agents to communicate with each other through the TCP or UDP communication protocols. When data is received on in Internet port an a SIGIO signal is sent to the process that is listening to that port. This SIGIO is caught by the agent signal handler and the agent's signal handler decides what to do with the signal.

In order to connect the signal handler to the main thread, InterStage provides the facilities of the *MDriver* object. An *MDriver* object, or an object derived from the *MDriver* class, contains a list of signals that are interesting to it. An application may contain any number of *MDriver* objects and may prioritize these use. When a signal arrives all the interested *MDrivers* are polled in order of their priorities. The *MDriver* class provides the poll method to accomplish this. The method is used to generate *MEvents*, which are placed on the event queue by the event scheduler. Once an event is enqueued it will eventually be executed in the main thread of execution. This type of interface allows events to be generated asynchronously while allowing the programmer to deal with each event as an atomic item, allowing the programmer to let InterStage worry about the concurrency problems like scheduling.

Every agent has at least one MDriver object. This is implimented by the fact that an agent is an instance of a BServiceObject which inherits an MDriver object. In actuality the BServiceObject is just a polite interface to the message service; most of the message service work is carried out by the ServiceObject class which is BServicesObject's superclass. The poll method of the MDriver class is a virtual method so any class derived for MDriver must define its own poll method. The ServiceObject's poll method is only interested in SIGIO which is used in message passing.

Every agent has an internet port that it uses to send and receive messages. This port is interfaced to the ServiceObject by the use of the FileRegister class which the Service object inherits. Actually the class hierarchy is MDriver, FileRegisterBits, which provides a low lever interface to file descriptors, FileRegister, ServiceObject, and finally BServiceObject. (The class hierachy uses only single inheritance).

Flow of control in an InterStage application is through the use of events. Each agent has a startup method which is used to generate a startup event. If an agent wants to be at the top of the chain of command then it can use the startup method to generate events in other agents by sending messages. Whenever a message is received by an agent, ServiceObject's poll method instructs the agent to read the data on the port by creating an MEvent that reads on either a TCP port or a UDP port. This MEvent will create another MEvent that will deal with the message. It the message is a request the dispatchRequest rouine is called and the MEvent will allow the handleRequest method of the BDialogue to be executed. If the message is a reply to a request then the MEvent will allows the handleReply method to be to be executed.

In order to generate events using Xt Intrinsics one has to use the *XtObject* class. The constructor of this class takes the same parameters as the Initialze routine and creates an application context and a shell that is used for the application. This *XtObject* is derived from the *MDriver* , from the *FileRegister* class which is derived from the *MDriver* class. This allows InterStage to catch any signals that are generated by the X server inresponce to an X event. One can use the callbacks of the the *XtObject* to send messages, thus allowing an XEvent to start a chain of events.

Conclusions:

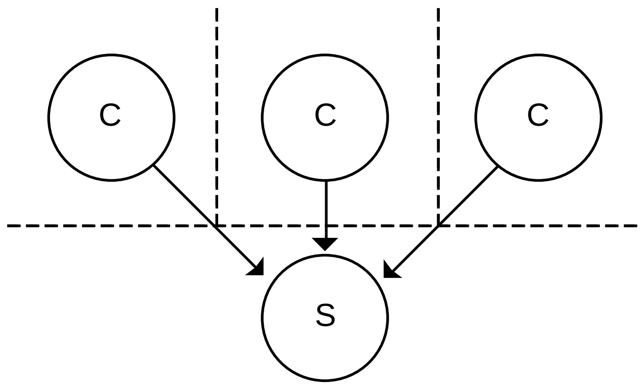
The InterStage system provides a novel approach to the design of distributed systems. It completely separates application functionality from the underlying network that the application is running on. Unlike traditional the traditional programming techniques used to create applications on distributed systems, InterStage doesn't require the programmer to know anything about the network while writing the application, saving the programmer the extra time required to deal with network configurations. InterStage also provide the user the ability to change the configuration at any time during the application's execution. This can be done by simply executing a few commands to the runtime system. The ability to reconfigure a system while it is running is not available with the traditional methods of programming on distributed systems.

InterStage also provides the programmer the ability to use concurrency afforded by a UNIX system and to use all the resources provided by a distributed system. Being able to use these types of resources requires little additional effort on the part of the programmer; it is very easy to enable a program to take advantage of concurrency.

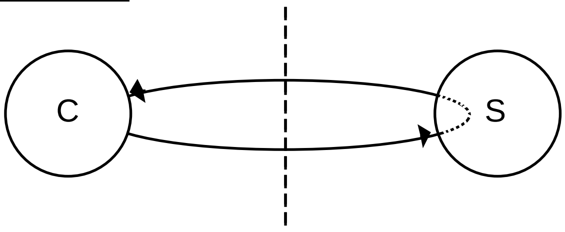
Moving from standard procedural programming to InterStage programming is not very difficult. It requires that a programmer accept two new concepts. The first is an object oriented concept. Instead of factoring a program into small functions which have controlling functions, a programer utilizes the notion of objects communicating with other objects. This switch is made easy by the InterStage paradigm which treats concurrency as if it were a chain of events instead the occurence of events that need to be scheduled by some outside source. The chain-of-events concept can easily be translated to a functional model. The other major concept that a programmer must accept is that all InterStage programs are event driven. Execution occurs because of the occurance of evnets from something external to the InterStage system.

Figure 1

Client / Server



Client / Server Using RPC



Multi-client / Server Using RPC

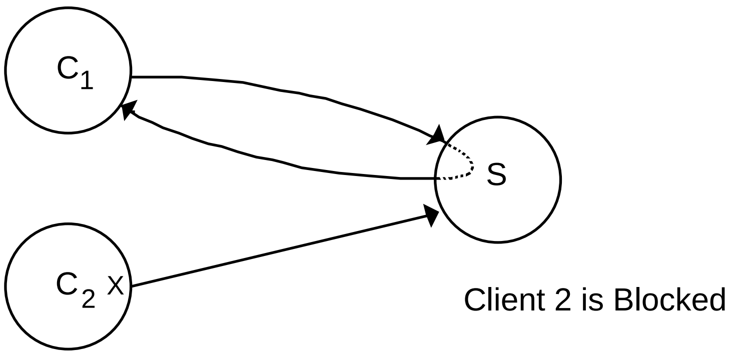
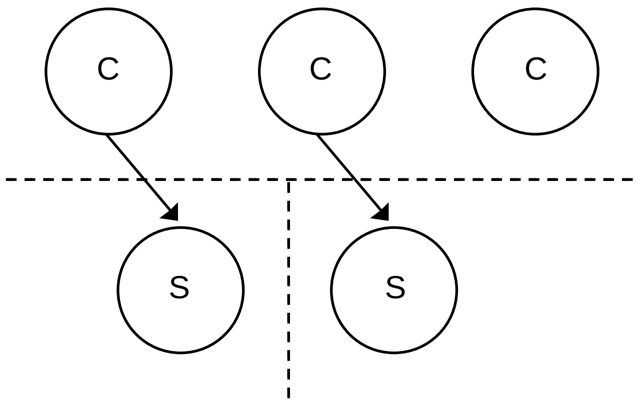


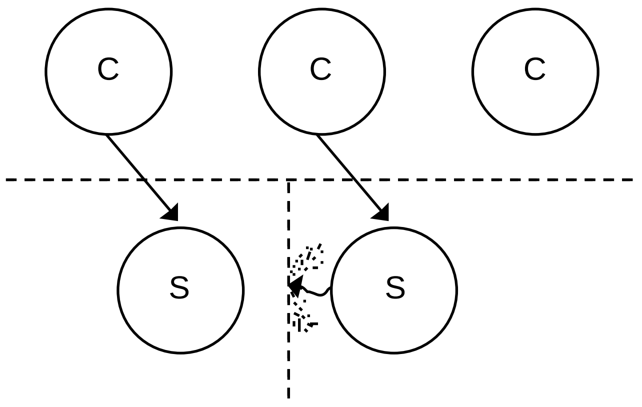
Figure 2

Multi-Client / Multi-Server



Partitioned service domains can help partially, but,

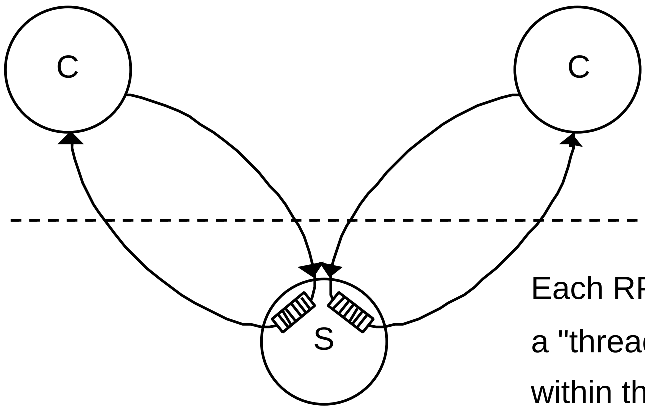
since a server can not make blocking requests to another server....



....there is no way for the servers to coordinate.

Figure 3

Thread-Based RPC

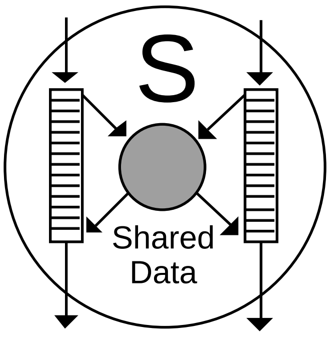


Each RPC call generates a "thread" (or lightweight process) within the server process.

- Clients are still blocked unless the client is also multi-threaded.

However,

- Significant complexity is incurred in making these processes thread-safe.



Threads executed independently and are scheduled outside the control of the programmer

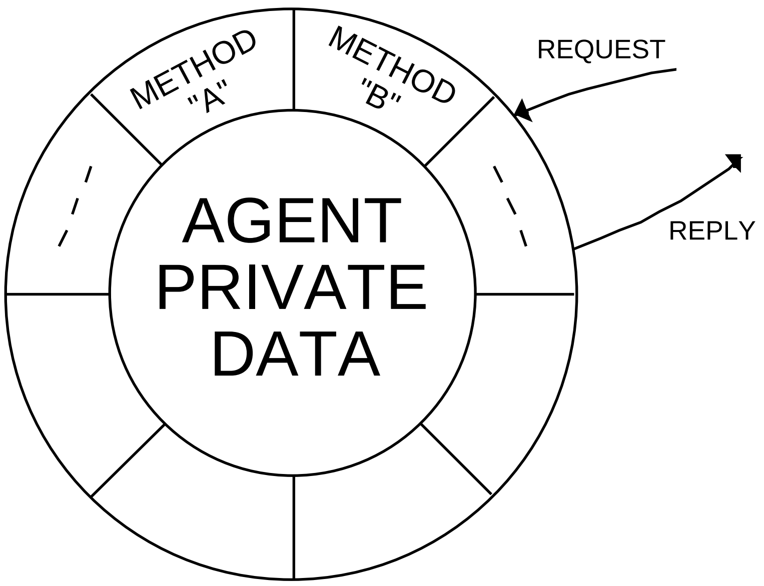
- Complex locking protocols must be devised
- Debugging significantly more complex

Figure 4

The Concept of an Agent

- Encapsulation of Data representation and Method Implementation at the process level ("large-grained objects").
- Network-wide provision of services, and usage of services, via methods
- Asynchronous message handling bid internally generated "dialogue" objects eliminate blocking.

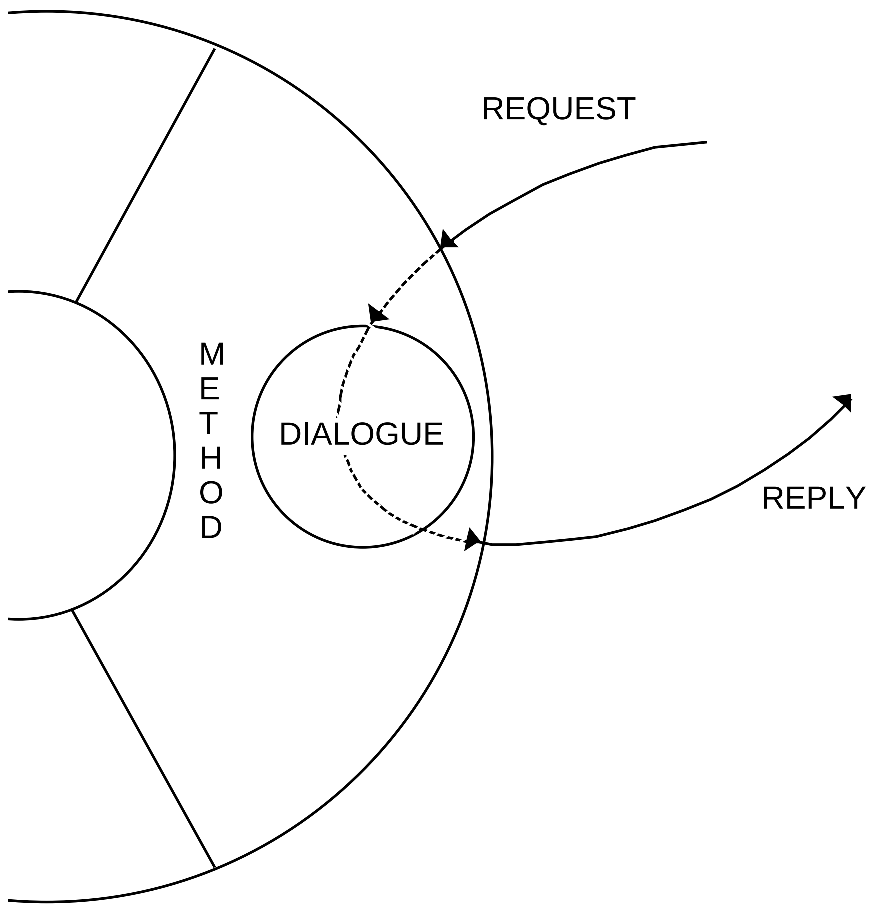
Symmetry Can act as either "Client" or "Server", thus supporting arbitrary patterns of communication.



- In support of the 0-0 paradigm, apecialized agents can be derived from more generic agents.

Figure 5

The Concept of a DIALOGUE



- Dialogue objects represent an invocation of method.
- Dialogue objects retain the state required during the servicing of a request.
- If a dialogue must make use of external services it suspends itself to free the agent to service other requests.

Figure 6

Dialogue use of External Services

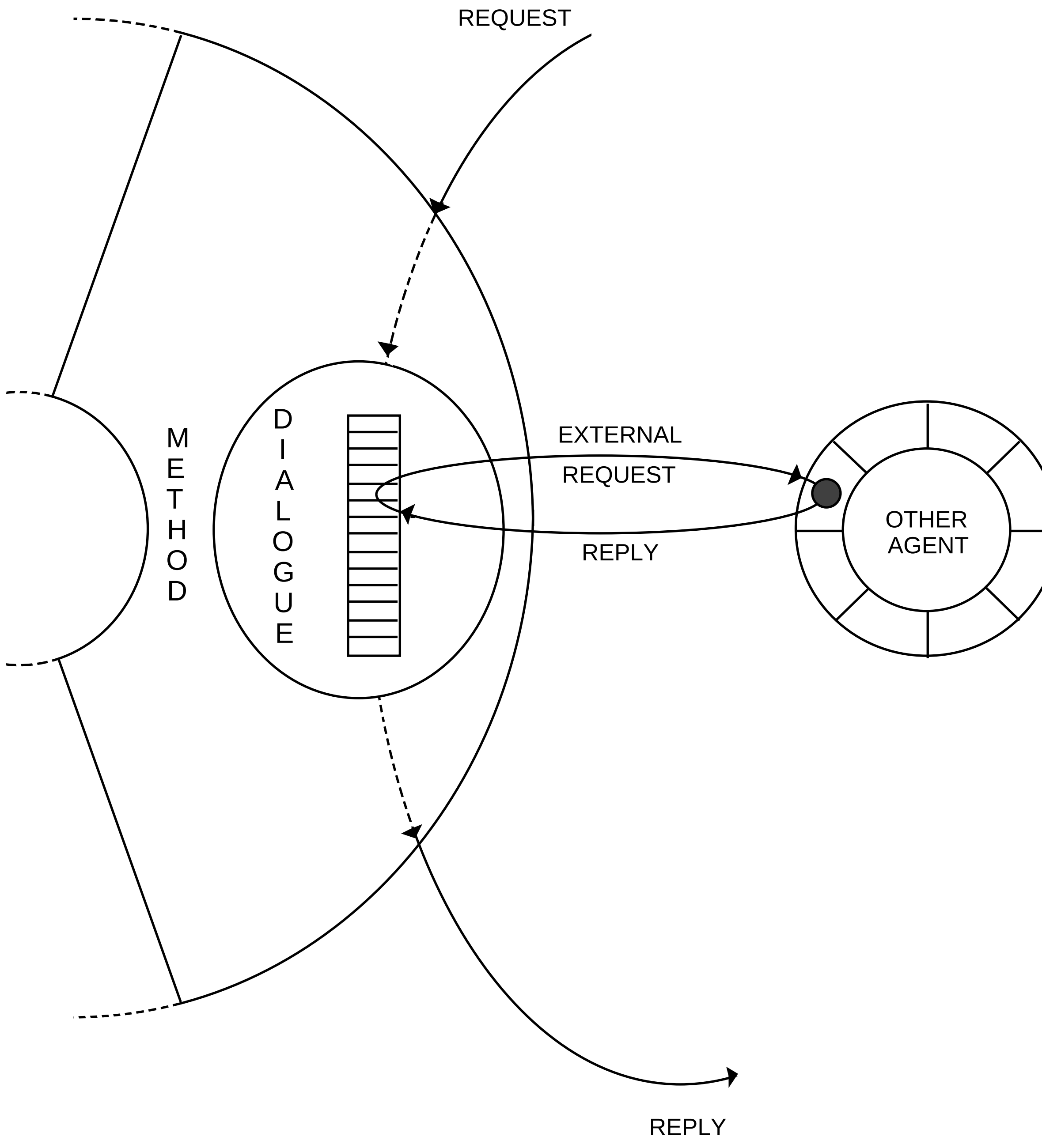
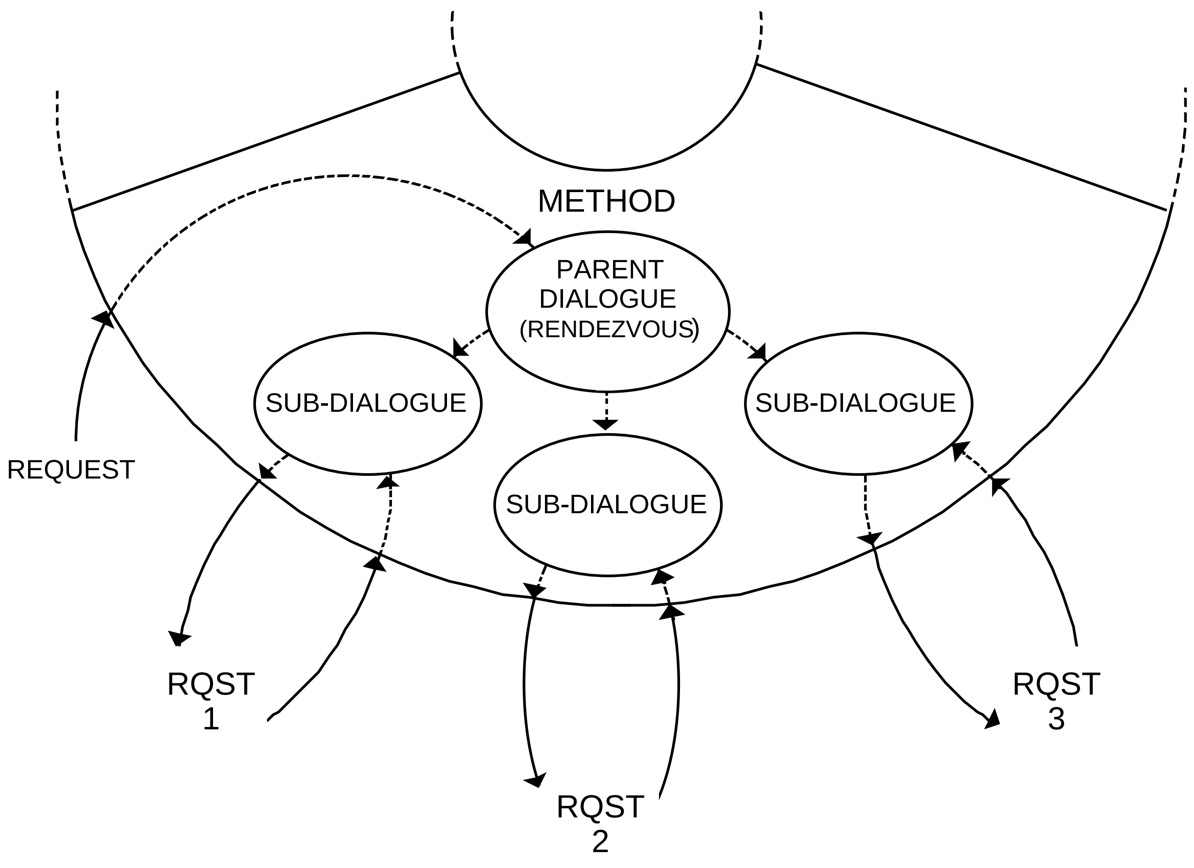


Figure 7

Dialogue - Parallel use of External Services



- A dialogue can invoke the services of other agents in parallel - if it is logically appropriate

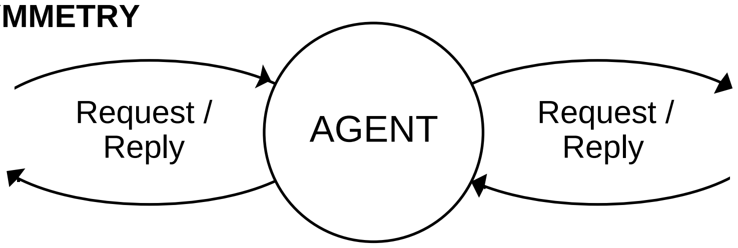
- The initial dialogue object creates sub-dialogue objects to manage this

Figure 8

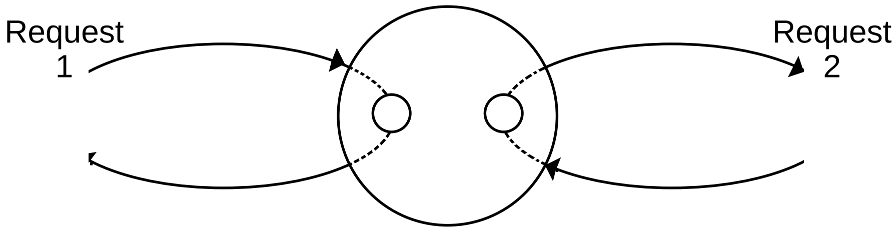
Arbitrary Patterns of Communication (1)

- In contrast to the client / Server model, distributed system design and implementation can now take advantage of:

AGENT SYMMETRY



ASYNCHRONOUS HANDLINE



PARALLEL REQUEST DISPATCH

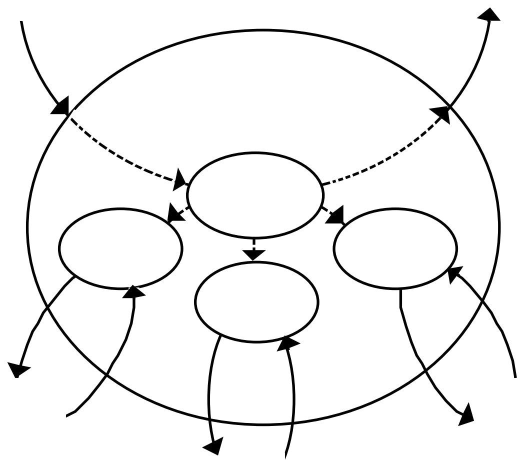


Figure 9

Arbitrary Patterns of Communication (2)

- As a means of providing scalability and robustness, these features allow system designs by more easily taking advantage of the following techniques:
  - Partitioning
  - Replication

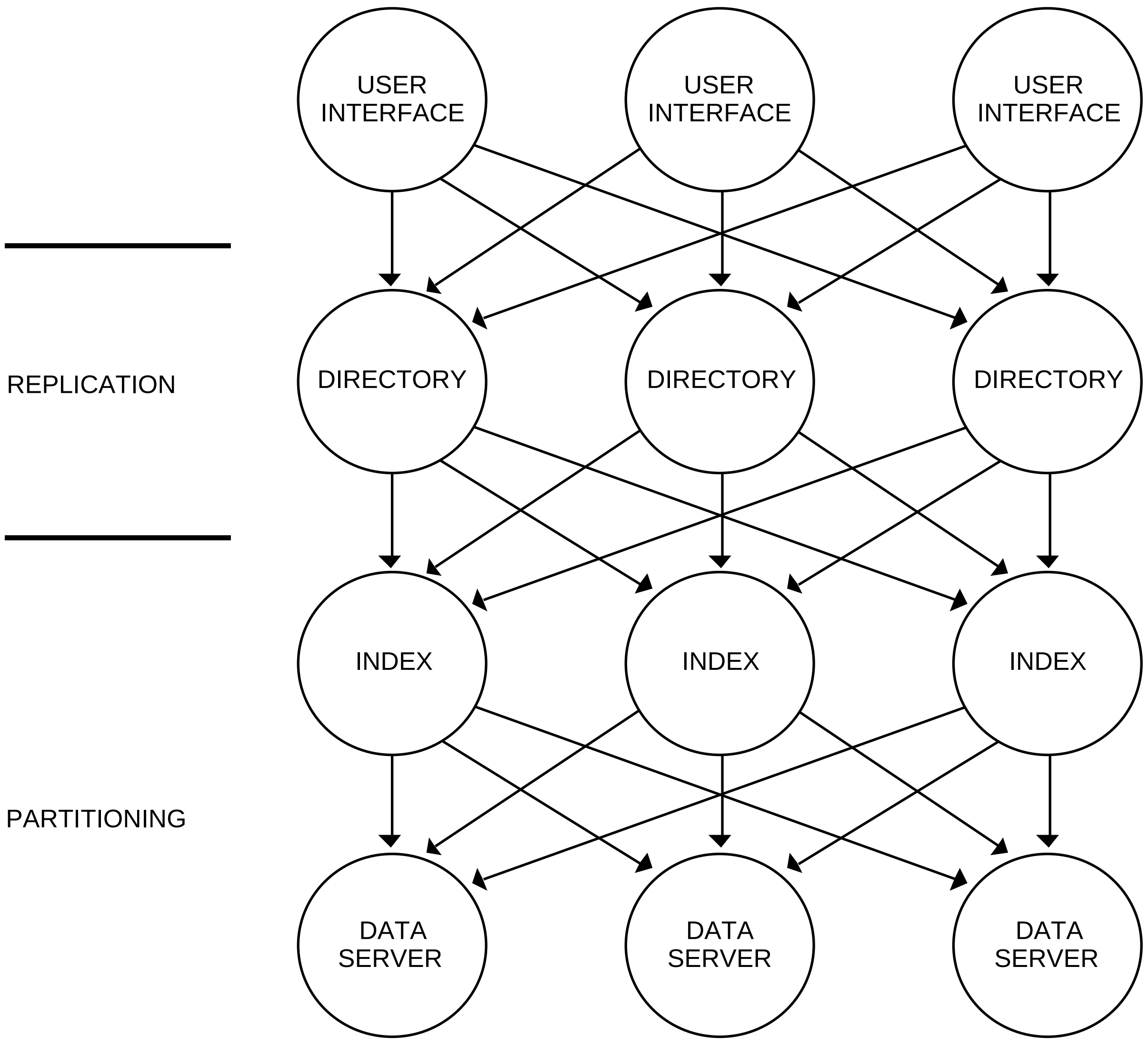


Figure 10

No Single Point of Failure

- The InterStage System lends itself to the design and construction of systems with no single point of failure. For example:

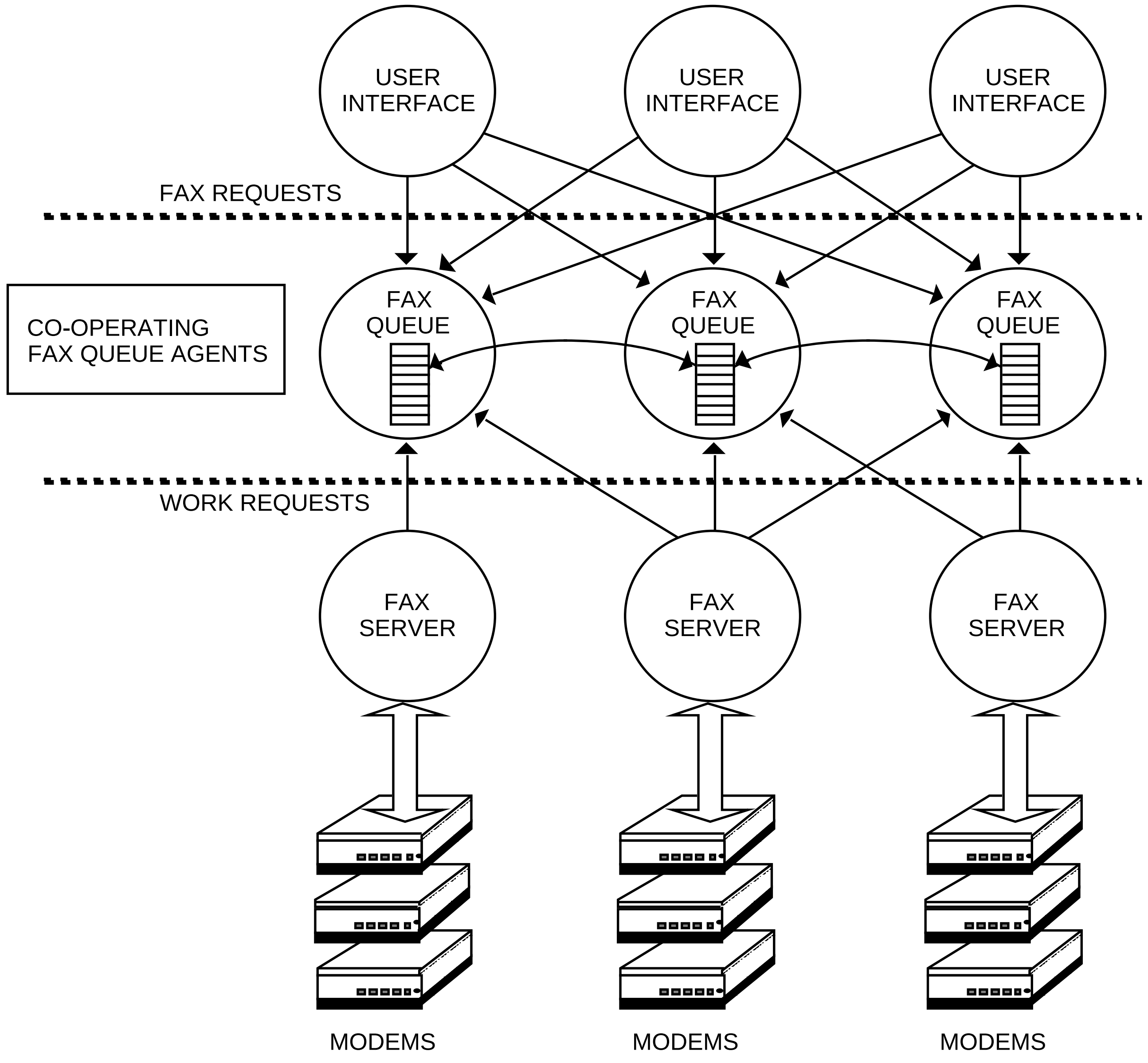


Figure 11

InterStage Supporting Infrastructure

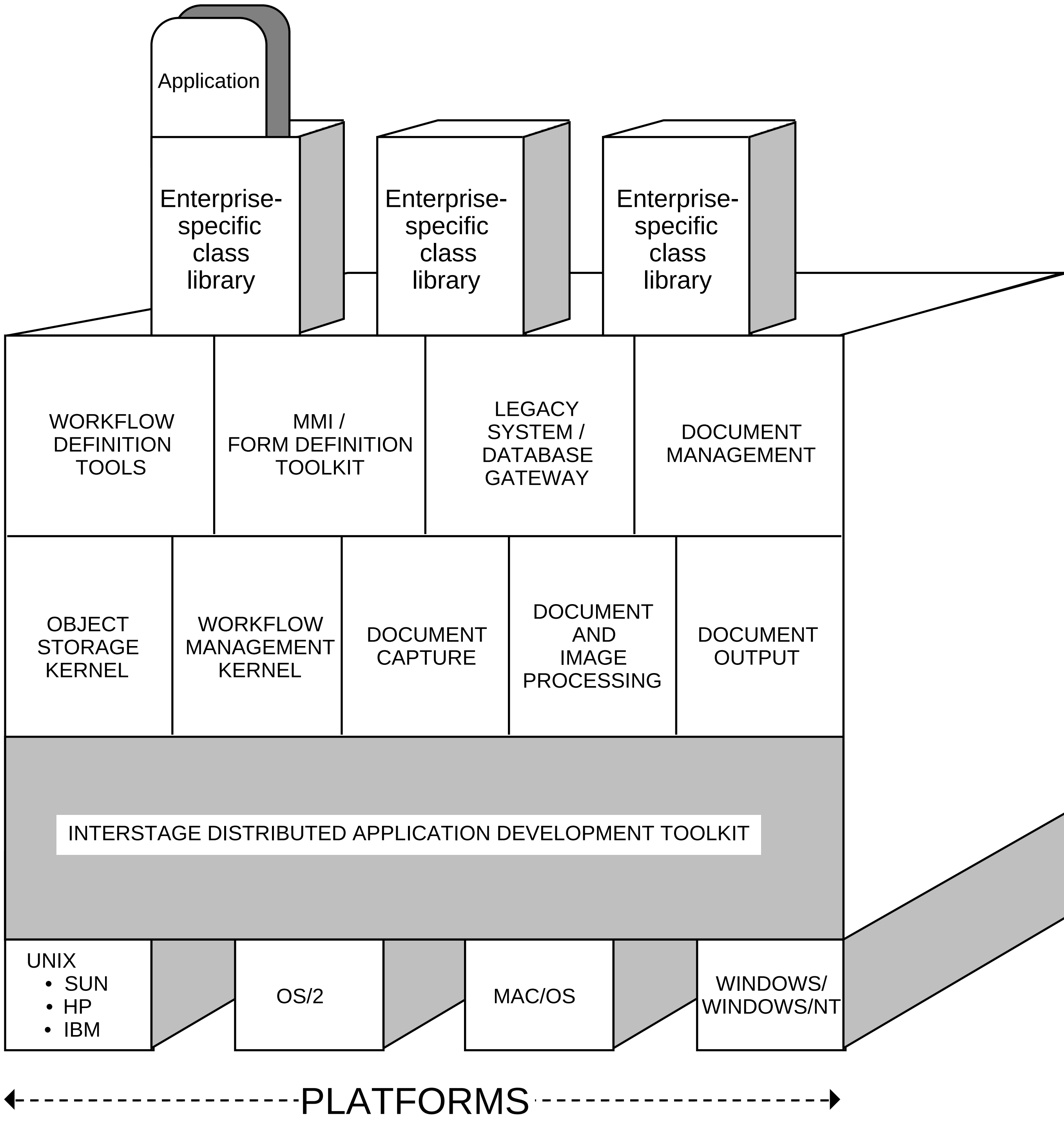
- Distributed Directory for Agents supplying services
  - Replication and convergent boradcast technique ensures no single point of failure
  - Designed to scale-up to a system with up to 10,000 nodes
  - WAN support
  - Administration and management of software configuration cn be performed with SNMP standard protocols

- Cross platform / Architecture support
  - X.208 / x.209 (ASN.1 / BER) used for all Inter-Agent messages. This provides architecture-independent messag syntax and encoding
  - ASN.1 complier auto-generates code to perform message manipulation

- Efficient use of network
  - TCP connections used for large object transfer
  - UDP pockets for small messages

Figure 12

InterStage as a base for Generic Subsystems and Generic Serivces


Application
Enterprise-specific class library
Enterprise-specific class library
Enterprise-specific class library
WORKFLOW DEFINITION TOOLS
MMI / FORM DEFINITION TOOLKIT
LEGACY SYSTEM / DATABASE GATEWAY
DOCUMENT MANAGEMENT
OBJECT STORAGE KERNEL
WORKFLOW MANAGEMENT KERNEL
DOCUMENT CAPTURE
DOCUMENT AND IMAGE PROCESSING
DOCUMENT OUTPUT
INTERSTAGE DISTRIBUTED APPLICATION DEVELOPMENT TOOLKIT
UNIX
• SUN
• HP
• IBM
OS/2
MAC/OS
WINDOWS/ WINDOWS/NT
PLATFORMS