Why non-determinism in middleware should not be ignored

Aurel Kleinerman
Chief Executive Officer
MITEM Corporation

Management introduction
Non-determinism is a computer science concept, with significant practical implications. Computer science has ignored the practical implications — because the theoretical solution is simply stated (if unachievable in a commercial environment). Academically, non-determinism lacks 'intellectual' interest.

Unfortunately, in the practical world of IT, the theoretical solution does not apply. Indeed the cost of resolving the difficulties raised by non-determinism rapidly become prohibitive as the number of 'components' which need to work together rises. This is particularly applicable to distributed systems connected by middleware, with major implications for project management, application maintainability and overall dependability.

In this interview, Dr. Aurel Kleinerman — the CEO of MITEM Corporation, a Menlo Park, California middleware company with specific expertise in this area — discusses:

- *what non-determinism is*
- *how it is relevant to all organizations which embrace distributed systems.*

What is non-determinism?

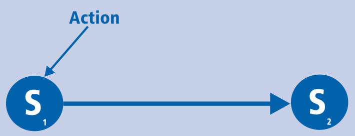
Computer science defines non-determinism in terms of finite state automatons which consist of:

- a system, that has a collection of states
- actions, which you can apply to the state
- functions, which allow a state — when an action is applied to it — to transition from one state to another.

A finite state automaton (Figure 3.1) consists, therefore, of a number of states (S), the actions you can apply (A) and the transition functions (delta) which enable you to make a transition. These systems (finite state automatons, or FSAs) can be defined as being:

- either deterministic: for any action applied to a given state the transition function always takes the system to a single, unique and pre-determined state
- or non-deterministic: for a given action applied to a given state the transition function could lead to any one of many possible states — and which one is impossible to predict ahead of time.

Figure 3.1: Finite State Automaton I



In the case of a deterministic system a given action applied to a state S_1 will always transition the system to another state S_2

In real life a non-deterministic system will usually have one state which is more likely to occur than the alternatives. Typically such a state will have an expected probability associated with it (say 95%); but a different state (or states) may occur (say) 5% of the time.

In a financial system, for example, that 5% might affect a $5000 securities trade or a $1B securities trade. The cost of the $5000 one ‘going wrong’ may be acceptable because the error cost may not be high. In contrast, an error with the $1B trade could be too expensive. Non-determinism means you are not sure how you will be affected.

Modem and database examples

In another context, the classic example of non-determinism occurs with a modem. The modem is good example of a software ‘gate’ — a layer of software which has two states:

- one state (S_1) passes any piece of data which comes from the input and is going to go to the output — without being intercepted or otherwise affected by the gate
- a second state (S_2), where the input data invokes a command mode.

The command mode means that any input received should not be passed through to the output but should be processed as a command to the ‘gate’ (modem in our example). The transition between S_1 and S_2 typically is by a special command or signal which is interpreted by the modem as an instruction to transition from S_1 to S_2 (in the Hayes command set this transition is achieved via ‘@@’ followed by a 2 second wait).

The problem is that transitions can happen at any time. You might have an ‘@@’ embedded at any time in a data stream. If by chance the data stream is delayed by a 2 second break after the ‘@@’ (for whatever accidental or other reason — a rare occurrence indeed), then whatever follows in the data stream will be interpreted as a command.

The net effect is to leave the ‘system’ in a totally unexpected state. All of a sudden the system is now in a different state than it was before — because of a non-deterministic event. You cannot predict this since you cannot know when it may happen — or even how it may happen.

Another example can occur when we say we are ‘going to a database’. In reality we are not really talking to the database but to a control program managing the database which, in turn, is another example of a software ‘gate’.

The control program is what actually makes the requests to the database (doing a select, obtaining data, performing a search, etc.). The control program interprets the request and then interrogates the database before returning the result. Like the modem software, it can (simplistically) be thought as having two states:

- a command mode
- a pass through mode.

To give an instance — situations can arise where the control program may be designed in such a way that when you are trying to access a particular field, the control program may intercept the request and, depending on the 'caller' account's permissions, it may or may not require a 'field access' password. Since such an access permission is not defined by the 'caller' (or user) of the database, but by the database manager, a user may not be able to predict when and why he may suddenly be required to provide a 'field password'.

Software gates are ubiquitous and well hidden within computer systems and are always capable of generating unexpected transitions. They are generators of non-determinism.

The most obvious example — the GUI

Non-deterministic systems occur least in simple systems. It is when you do not (or cannot) know all the possible states that the difficulties arise.

Perhaps the most obvious example lies in user interfaces which necessarily have to 'deal with' non-determinism because they have to interact with humans. Humans are a great source of non-determinism to a program — because humans do not respond as programmers would like them to behave.

Practice has shown that people interpret data and messages (and even undreamed of inputs) that the programmer does or cannot anticipate. By definition user interfaces are non-deterministic. And, because of this, almost every program which uses human interfaces must explicitly address non-determinism which means that programmers spend much time seeking to eliminate or reduce the choices which people have. In so doing, developers have to check for everything possible to encourage the user into a predictable pattern with known consequences.

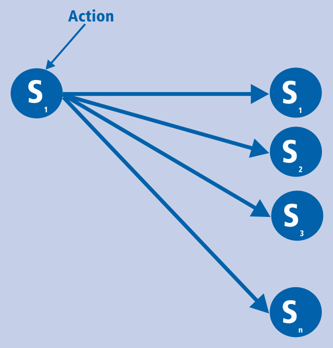
Programmers want determinism. The goal of any programmer, especially when he or she writes user interfaces, is to try to reduce the user's choices to such an extent that the system becomes deterministic. However, as I shall discuss, this has a significant impact on project development and timing.

Differentiating between determinism and non-determinism

Without digging into too much theory, the reason for differentiating between deterministic and non-deterministic systems is that basically we only know how to program deterministic finite automatons (DFAs). The programming languages which we use to tell computers what to do assume a deterministic environment.

As stated above, non-determinism is academically uninteresting. The reason for this is that there is a basic theorem in computer science which says that a non-deterministic finite automaton (NFA) with n states is always equivalent to a DFA with 2^n states. [For every possible sequence of consecutive states from the NFA (a path) we associate a single state in the equivalent DFA.]

Figure 3.2: Finite State Automaton II

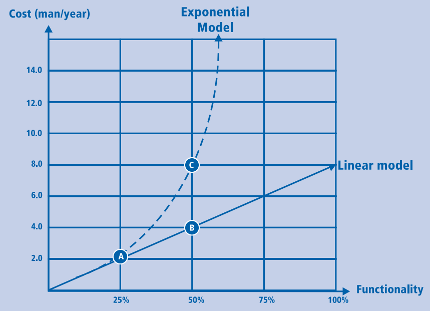


If the system is non-deterministic the same action applied to a state S_1 will, unpredictably, transition the system to any of many possible states S_1, S_2, S_3, ..., S_n

But, once you have said this, it means that all non-deterministic systems convert to deterministic systems — if you assume that all the possible states/results which can occur have been identified. Therefore, from a theoretical point of view, there is no challenge.

This is fine in the academic world. But it is impractical in the commercial world — where, in effect, developers are confronted with writing code to cover a DFA system which has 2^n states.

This is further complicated by the fact that it is impractical in the commercial world to identify all possible states in advance. Put another way, there is no ‘methodology’ for testing for all possible states when such states may occur very rarely, like (say) once a year. [Such a rarely occurring state may be tolerable in a program that is used by few people but is unacceptable for a program used by thousands — in which case a ‘once a year’ state may occur on average three times a day.]



Assuming a linear development model, it takes 2 man/years to complete 25% of the project (point A) and 4 man/year to complete 50% (point B). If the project, on the other hand, is exponential then it may take 8 man/years to complete 50% of the project (point C). Note that, in this example, 8 man/years are required to complete 75% of the project if the linear model is assumed but more then 1000 man/years will be required in the exponential case.

Figure 3.3: Non-linearity in development

Although academically uninteresting, non-determinism is a practical problem because of 2^n. This brings us to the project management implications.

The impact of non-determinism on project management

To understand the significance of this non-determinism and project management, think about this example. Consider writing a computer program which consists of transitions through 32 different states. For each state, assume 50 lines of code must be written on the average. Assume, also, that a good programmer can write 10 lines of debugged code per day (the average for a programmer in the USA is 2 lines/day).

It is, therefore, reasonable to expect that this project will be completed in 32*50/10 = 160 ‘man days’. This is at a total cost of 160*75 = $12,000 (if we assume $75 per ‘man day’).

These calculations were made based on the assumption that the transitions from state to state in this program are deterministic. If this is not the case — and if we assume the worst case where all transitions are non-deterministic — then the project will be completed in 2^{32}*50/10 = 21,474,836,480 ‘man days’, at a total cost of 21,474,836,480*75 = $1,610,612,736,000. This is larger than the entire US budget.

In reality not all transitions are non-deterministic and some clever algorithms may be identified to reduce the problem further. Nevertheless, a multi-million budget could reasonably be expected to be spent to address just the non-determinism. Any project manager who has to deal with 2^n states has to cope with an exponential function (it is not linear and is typically represented on a graph (Figure 3.3). In software, the problem is that we almost always assume linear development of systems. Almost nobody considers the issue of non-determinism.

But the reality is that we do not have the time nor the resources for the work required to tie down all the NFAs. Instead we look to produce results as quickly and inexpensively as possible — which usually inhibits full identification of all the possible NFAs.

In a linear system this would not matter. We would assume that if it takes me two days to deliver a given result, then it takes another two days to double the amount of code. Equally, if I have two man years to reach point A (in Figure 3.3) and if I put in another two man years, I will double the output.

But a non-deterministic system is not linear. It is exponential — which produces dramatic differences. The key (as in Figure 3.3) can be seen when comparing:

- the first quarter, even half of a given project
- with the third or fourth quarters (or second half) of the project.

At the beginning, the divergence of the linear from the exponential is relatively small. In the second half it is dramatically different. Indeed non-linear systems appear, in the early phases, to be linear or to be approximated by a linear system. This also explains why so many projects go ‘off the rails’ the further they continue. If the differences are not understood up front, then by the time they become visible they are already growing enormous.

Now add the issues raised by not investigating all the possible states in advance — and assume these only start to become clear in the middle of a project. You can see why the old truism, that the last 10% takes 90% of the effort, has validity. In the early phases we almost always underestimate the effort needed to reach completion. If you assume a linear system you will underestimate the effort to complete.

This is one of the most insidious effects of failing to understand and adjust for non-determinism. Indeed, the irony is that most developers (and their management) do not realize there is even such an issue as non-determinism. In consequence they underestimate the complexity of what they are attempting to do.

Apply this to connecting multiple systems with middleware and the reason why non-determinism is so important becomes instantly clear. And, as I mentioned earlier, all programming systems have an element of non-determinism in them.

A curious additional dimension is that many project managers do recognize that there is a problem. However this is not attributed to non-determinism but rather to, for example, poor communications, insufficient preparation, inappropriate skills, etc.

What then occurs is the recognition that significant additional effort will be required to complete a system, that a project is growing exponentially in complexity. But the presumption is made that it is the people who are the problem, not the system. The reality is that it is actually the latter.

Reducing non-determinism

Our goal is to find ways to reduce non-determinism. One way to achieve this is to figure out ways to reduce the number of states and of all the possible non-deterministic transitions. If I need only four states, instead of 120, then I will reduce significantly the magnitude of non-determinism.

The problem becomes the undiscovered states. What do I mean by this? For a programmer, most non-deterministic systems are not well defined. Think about this in the following terms. Assume I have a state, S, and want to apply an action to it. Non-determinism means that I will be moving to one state from a set of possible states, which I shall call S_1, S_2, to S_n based on some probability distribution associated with each possible state.

It turns out that most programmers trying to build such a system do not know (or think through) all the possible transitions in the early stages of a project since many of the possible transition states may not necessarily be encountered or anticipated (they may be aware of S_1, S_2 and S_3 but not of S_4 to S_n).

The classic example comes at the user interface. Predicting how people will actually interact may be very different to how the developers expect them to interact ... [We all have heard the story about the user literally ‘taking the diskette out of the cover’.]

In effect we build systems based on our expectations rather than on predicting human frailties. The example I like the best about this is associated with microwave ovens. Should, or could, the designer anticipate that there will be a person who takes his (or her) dog, washes it and briefly puts it in a conventional oven to dry. Having bought a new microwave, which also heats food, does not mean that placing the dog in the microwave will produce the same results as the regular oven.

The point I want to make is that no matter how good we are at trying to predict non-determinism we can never know all the possible transition states. This is a real problem — how do you discover very rare states. This is not addressed by the academic world.

It is also a very serious problem although, at the user interface at least, you can do something with focus groups and other similar mechanisms to contain the variations. However, when you look at communicating software (such as in distributed systems — where there is minimal external manifestation of what is happening) the difficulties are both greater and more serious.

The middleware dimension

In 1976, Professor Tony Hoare, then at Queen’s University, Belfast, introduced the concept of communicating sequential processes (CSPs). He was trying to build an operating system which allowed communicating systems to work closely together.

From his research he concluded that two computers talking to each other over a wire represent an example of a degenerate process or a degeneration of what is considered parallel processing. In parallel processing two processes need to synchronize.

In a system in which they (the processes) share memory, synchronization can be delivered almost instantaneously (through setting, and looking, at semaphores).

This model becomes degenerate when you have two computers talking to each other over a wire — because of the latency associated with sending messages between the two (or more) systems which have to be synchronized. Indeed, the problem turns out to be even more complicated because you have the added complexity associated with both systems needing to have the ability to receive messages when they might also be sending. This is analogous to the ‘in-doubt window’ of decision theory or what is sometimes referred to as the ‘Two Generals Problem’.

The Two Generals Problem

In the latter case you have two allied armies on top of two hills with an enemy in a valley in the middle. If the Generals in command of each army can co-ordinate (synchronize) they can beat the enemy in the middle. But if only one (say A) attacks, the enemy in the middle defeats that army (A) and then goes on to defeat the other one (B).

To win, General A must tell General B to attack at the same time. So General A sends a message to General B saying ‘attack at dawn’. But he needs confirmation that General B has received the message. Assume General B receives A’s command and sends a message back saying ‘OK’. Does General B know that his messenger arrived? Perhaps he (the messenger) was killed or captured en route ... and so on. Critical uncertainty abounds.

In systems connected by wires (middleware-based systems), this problem swiftly becomes complex. The issue is how do you synchronize two (or more) systems which:

- are supposed to operate in a parallel environment
- do not share common memory.

Non-determinism arrives all over again. The nature of the uncertainties creates non-deterministic transitions which are almost impossible to predict or describe in advance. If you cannot predict these, how do you build solutions and test them? When you are dealing with commercial transactions, this becomes critical.

What Tony Hoare was trying to do with CSPs was create an operating system to take care of these types of problems (although it can be argued that he was really concentrating on system availability and not on system synchronization). In effect, if he could build a layer between the linked systems which was always available, then the messages would reach their destinations and still try to set up semaphores such that even if one system was not available at least it could look at the semaphore. In practical terms what we want is a system that is always ready to receive messages.

But this still does not resolve all the issues. Take collusion. In the case of the Generals, General A sends a message to General B saying ‘attack at dawn’. But Generals A and B have the same rank and B has also sent a message — but this says ‘attack at noon’. Now you have equally ‘authorized’ but conflicting sets of instructions. To resolve the contradiction, not only must both sets of orders arrive but a resolution loop must then start ...

This is very similar to a deadly embrace. These are notoriously difficult to resolve in system (as opposed to human) terms.

To summarize, what we have here is another example of non-determinism which is found in communicating sequencial processes or ‘non-determinsim of CSP’ (to use Tony Hoare’s terminology). Each system (General) is a DFA. But combining the two DFAs produces an NFA with n x m states where n and m are the number of states of each original DFA respectively.

In a general way, this becomes basically a DFA again, with 2^{nm} states, which is a large number of possibilities to predict and resolve. The key reason why we encounter this form of non-determinism is that we have two (or more) systems. Any distributed system will, therefore, be non-deterministic unless developers take specific actions.

Trying to avoid non-determinism

In practice people try hard to avoid the non-determinism of the CSP problem (Figures 3.4 and 3.5). The most common technique used is what I refer to as a ‘ping-pong system’.

This is basically a single system, although consisting of two CPUs which work serially (as opposed to in parallel), to execute what in practice is a single program. The program starts on one CPU, then it is passed to the other CPU for further processing. During this time the first CPU is for all practical purposes stopped until the second CPU completes its task, and so on.

While this solves the problem it does so by behaving as one single system and not as two separate systems working in parallel. In effect we have artificially imposed a discipline — which explains one of the reasons for the popularity of remote procedure calls (RPCs). RPCs are seen as reducing (non-deterministic) complexity because system A will not transition unless B comes back with a response.

It turns out, however, that RPCs do not remove all non-determinism. While non-determinism introduced by the CSP is removed at the expense of performance, the communications between the two systems (which may be thought of as a third system in itself) also introduces non-determinism because of all the 'gates' (communication layers) which are featured between A and B.

RPCs are, theoretically, a great idea — if it was not for all the complexity in between. It appears to address non-determinism, but the reverse is the reality. Non-determinism unfortunately is like radio noise: it is always present and needs to be addressed.

As described above, one way that initially deals successfully with the problem of non-determinism of CSP is to use a blocking technique for synchronization with the other system. This has two disadvantages:

- the blocking means you lose the performance advantage
- you lose the parallelism of two (or more) systems.

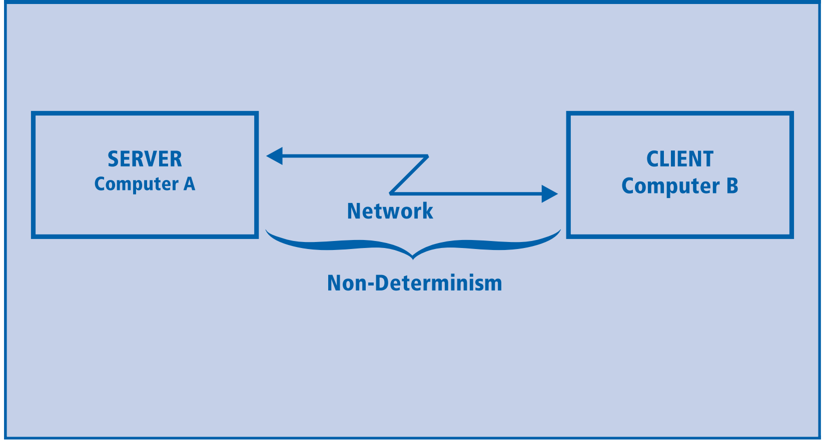
Another way to reduce non-determinism is to try to reduce the number of states which exist. If you keep n and m small, then building a solution may be a practical possibility (you prevent 2nm becoming too large). In other words, choose either a very thin client or a very thin server.

An example is the SQL client/server (thin server) on one hand and terminal model or Web browser at the other end (thin client). Simplistically the DBMS (a thin server) can be thought of as having only five states: (open, closed, selected, updated and committed) while a Web browser (thin client) has three states (transmitted, received and closed). In either case the number of transitions for the system is either 2^{5n} and 2^{3n} where n is determined by SQL client or the Web server.

If n is kept small then such a system can and has been built. But the problem with keeping n small is that such a system can only do a few things. It is, therefore, not that useful for building complex applications.

A third way, which is of some interest to the academic world, has been to try to discover if an algorithmic solution exists which will reduce the number of possibilities when we convert an NFA to a DFA. Unfortunately, despite much research, no answer has yet appeared which generally automates the bypassing of non-determinism.

Figure 3.4: Traditional 2-tier distributed system



Messages and state

Let us revisit the two generals problem again. Assume General A sends a message to General B saying 'let us attack at down' — and receives back the answer 'OK'. But, at dawn, when he attacks, General's B army is nowhere to be seen. This example is designed to emphasize the common confusion between messages and states.

Two computers talk to each other by sending messages to each other. Each side interprets the message in order to determine the state of the other side. Returning to the Generals, the message may say that General B should attack at dawn — but General B's army may not be prepared to attack then (in spite of ‘B’s’ best efforts and intentions). The point is that General B may wish to attack but unexpected events (occurring after the confirmation was sent) may prevent the General B from attacking at dawn.

Thus a message can unintentionally be misleading about the state of one system — with the result that incorrect expectations may be set. Apply this to a financial transaction and the consequences can readily be appreciated.

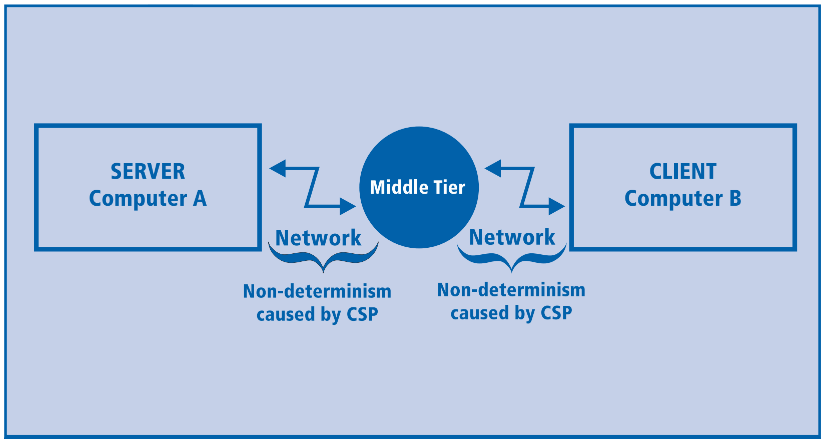


Figure 3.5: Middleware-based 3-tier distributed system

Another example is relevant. In UNIX, the caret (prompt) is a message. It says that UNIX is ready to accept an input. But it does not tell you what state the UNIX system is in; it could be that the system is ready to accept the a/b/c folder or the x/y/z folder: you do not necessarily know.

Messages are not necessarily synonymous with state. The fact that you send a message does not necessarily describe in what state the other system may be by the time the message is received. Again reverting to the Generals, they may agree to attack at dawn. But what happens if the enemy in the middle attacks earlier, say at midnight.

Messages do not address the problem of non-determinism and in fact it may complicate it — as exemplified above. This is relevant when considering queuing of messages in an intermediary system. While queuing of messages is a valid technique when and where the system parts do not require synchronization, for example when no timing component is required, it may exaggerate the problem of guessing the state of the message sender. [By the time I receive, in California, an e-mail from a customer in England his computer may still be available but there is no way to tell just by receiving the message.]

While excellent for guaranteeing that a message and its receipt are eventually delivered, message queuing actually exacerbates the problem of non-determinism of CSP. Indeed it can be said that message queuing systems are useful for collaborating systems but not for real time co-operating systems.

The implications for distributed systems

If you assume that there is no technology that solves the problem of non-determinism of CSP (the core problem of distributed systems), the first implication of all the above applies to management, specifically development and project management. The first action is to accept that distributed system development is non-linear — which means that you must accept the exponential dimension (and costs). An immediate corollary of this is: do not make the mistake of underestimating the cost of developing distributed systems. This may explain low project success rates (the Wall Street Journal estimated 42%, which seems to me to be an over-estimate).

A second implication arises from the complexity of middleware-linked systems. Although difficult, it is important to understand that you can never completely deal with all the unknown states in advance. They cannot all be pre-described (frequently they are ignored until a problem occurs). One consequence of this is that your distributed systems may be much more brittle than you either expected or planned — which is proved as these break in unexpected ways.

A third implication is the issue of maintenance. The problem with most non-deterministic systems is that by the time you make them work, it is almost a miracle. Any time you try to modify them or to touch them, you can easily (albeit unintentionally) re-introduce non-determinism. A valid question to ask yourself is if middleware linked systems are genuinely maintainable?

Fourthly, islands of automation are created. What do I mean? An island is a system that exists by itself. This may be because individual elements or systems are so difficult to maintain or modify that trying to bring these together into a bigger system (as in enterprise application integration) is almost impossible, no matter how much you spend. Look, for example, at how much organizations are spending on SAP's R/3 — which then sits in glorious isolation. You would have thought that 'modern' application solutions would avoid (or at least minimize) this result. It does not seem to be true.

Can non-determinism be cracked in practice?

From all that I have discussed above, non-determinism of CSP (non-determinism which occurs when building distributed systems) may appear to be an uncrackable constraint. At MITEM, after 14 years of research, three patents and extensive field testing we do not agree. The first step is to understand and accept that non-determinism exists as a problem. Until you do this you are operating with your hands tied (and eyes blindfolded). If you do not know about it you cannot act upon it.

The second step is to realize that there is nothing that one can do which will ever wholly remove the non-determinism of CSP. If you look at the way people have tried to address this issue — blocking, an algorithmic approach, etc. — the difficulty is that the resulting solutions do not provide a global solution. In turn this means that the problem will always re-occur at the interface between two systems.

Ideally what you need to be able to do is to provide a framework which returns the development effort to a linear problem. If you achieve this, you have a chance to address the development costs, brittleness, maintainability and other issues I have raised.

In the work of Tony Hoare, where you have two systems A and B, non-determinism appears in the communication between A and B. Our suggested framework must be implemented as a middleware layer (unless I could put A and B back on one system — when I would no longer have the problem).

Now the issue is, how do you design a middleware layer such that the interactions between systems A and B — and with the middleware — are such that linearity of development is preserved? You need to think about middleware not as a transport but as a layer where:

- A does not talk to B
- B does not talk to A
- both think (separately) in terms of talking to M, the middleware itself (Figure 3.6)
- M insulates both A and B from non-deterministic transitions.

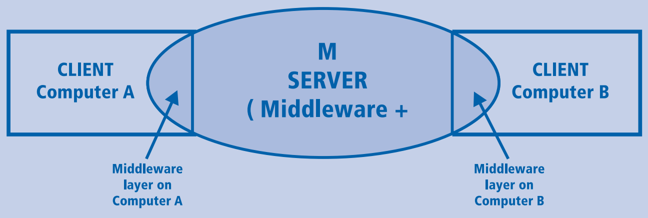
There are consequences which emerge from this. We know that the non-determinism will always occur in the communication path; therefore, in order to get rid of the non-determinism, you should locate M on the same system as A or B or on both A and B.

For example, it constantly puzzles me to see that people (who talk about middleware) think that by introducing an intermediate layer (or system) or third tier that you make development of distributed systems easier to build. It is not true. Non-determinism of CSP occurs now at both interfaces between the middle layer and the two other tiers. Even if I use RPCs all I have done is to complicate — and slow down — your solution even more.

Returning to systems A and B, you can establish some basic principles — for example, that middleware must:

- itself be distributed
- be designed to deal with non-determinism
- make the interface deterministic (between the mid dleware and B or A or both)
- force the developer to talk to M (the middleware), not to the far end.

Figure 3.6: The MITEM approach to non-determinism



In this case the non-determinism of CSP is hidden by various middleware components. The original client on Computer B is designed to interact with the middleware Server component. Note that in the case where the CLIENT on Computer A is a legacy system the Middleware layer on Computer A' is the terminal control layer (VTAM in the IBM world).

In other words middleware is not just a transparent layer between A and B. It must act as a distributed server or, another way to say this, is that A and B are clients to M (Figure 3.6). The corollary is that the only server to think about is the middleware. In reality the middleware becomes the server; everything else is a client to the middleware. You do not write a system A that talks directly to a system B.

The question that could be asked is whether or not this implies a collaborating solution rather than a co-operating one. I say ‘No’. While this will support collaboration it will also support co-operation — because the synchronization between systems is delivered by the middleware. It is the middleware which is the co-ordinating entity.

This is what we offer in MITEMView as a way to overcome the challenge of non-determinism. In fact, to maintain simplicity, we have chosen initially to implement the middleware on one machine (rather than many).

This is akin to making a terminal an ‘eye’ into the system. Legacy terminals are an extension of the application. The terminal in itself is part of the application. The terminal was already designed to be synchronous with the application: it is the way to send messages. By embedding the terminal into the middleware and putting this on the client machine you achieve the correct effect. In effect we have one system that is synchronous and, because of this, we deal with non-determinism at the interface.

Our goal has been to provide a system which restores determinism at the interface between clients and middleware. The client does not think that he is talking to the host application; instead it (A) is talking to the middleware which synchronizes state with the middleware. Similarly it is the middleware which synchronizes state with B.

Management conclusion

Determinism is an issue whatever middleware you choose. The biggest problem, however, is whether users of other middleware understand the implications — and weaknesses — which a lack of awareness of the implications of non-determinism brings. It is not that other middleware products are functionally wrong. Rather it is that the expectations are incorrectly set. It is when the expectations are linear and the reality is exponential that the problems arise.

This becomes a particular issue when evaluating middleware and subsequent proof of concept trials. By their nature, proof of concept trials are modest in scale. Coincidentally this masks the size of large scale deployments — because the size of the proof of concept is such that all appears under control (not least to complete the trial as fast as possible).

Yet the results may be utterly misleading once you try to scale up. Since people have a linear expectation set by the trial, they extrapolate for the larger projects. Unfortunately the non-deterministic effects will not become apparent until they are midway through an implementation. At this point there are two alternatives:

- *stop development, which may explain the less than stellar record in distributed application development delivery*
- *cut back on the original objectives (thereby under-delivering against the original justification).*

To Dr. Kleinerman this is most frustrating, especially when he sees MITEMView as possessing an architecture to minimize or control the debilitating effects of non-determinism. The effects described above do not have to happen. They happen primarily because of ignorance of the implications of non-determinism.