An agent based architecture for constructing Interactive Simulation Systems

Zhao, Z.

Citation for published version (APA):
Chapter 4

Implementation and performance analysis

In this chapter, we discuss the implementation of ISS-Conductor. We start with Communication Agents and Module Agents, and then discuss how they are combined in the Actor and Conductor of an ISS-Conductor component, after that we study a test case to investigate the performance characteristics of the system*.

4.1 Communication agents

A Communication Agent has a data object manager for managing the structure and contents of data, a distribution manager for sharing the data with other ComAs, and a reflex task-processing engine for responding to events received from the external world.

4.1.1 Data object manager

In a ComA, the structure of the data is described as data classes, which contain a set of attributes, and the contents of the data are managed as instantiations of the data classes. A class is called shared when its instances can be accessed remotely by other ComAs, otherwise called internal. The shared classes that have persistent instances are called shared object classes, otherwise they are called message classes, and their instances are called shared objects and messages respectively. A shared object class can be syntactically mapped to classes which are defined in the other ComAs. The data object manager manages the lifecycle of data objects and provides name services for them. It also buffers the contents of the shared objects that are updated by remote ComAs before the ComA processes them. Inherently, ISS-Conductor predefines a number of data classes in the ComAs:

*Parts of this chapter have been published in Z. Zhao, R. G. Belleman, G. D. van Albada and P. M. A. Sloot. “State Update and Scenario Switch in an Agent Based Solution to Constructing Interactive Simulation Systems”, in the proceedings of the Communication Networks and Distributed Systems Modelling and Simulation Conference, San Antonio, US, 2002.
1. *Monitor* is an internal class. Its attributes point to the data structure defined in the Actor for tracking and accessing their run-time values;

2. *Task* is an internal class, which describes the structure of the events;

3. *ComA Message* is a shared class, which describes the structure of the information exchanged between the Actor and the Conductor in a component;

4. *MA Message* is a shared class, which describes the structure of the information exchanged between Conductors;

5. *Control Message* is a shared class, which describes the structure of the information exchanged between all Actors and Conductors.

### 4.1.2 Distribution manager

Using a software bus to communicate, ComAs exchange both shared objects and messages using a publish/subscribe mechanism. The multicast groups for distribution are handled by a *distribution manager* through routing spaces. The routing spaces are defined as two-dimensional planes, where regions are determined by the co-ordinates of two diagonal points. One shared class can only be associated with one routing region at a specific time. At run time, a ComA can change the policies for distributing a shared object by modifying the region of the associated routing space. The basic rule is that ComAs can only exchange a shared object when they have overlapped regions. To simplify the region adaptation, ComAs provide four routings profiles: *inside a component, Actors/Conductor only, all components, and away from the others*, as shown in Fig. 4.1. Four routing spaces are predefined in ComAs: *componentRouting, agentRouting, controlRouting* and *objectRouting* which are used for delivering ComA Messages, MA Messages, Control Messages and persistent objects respectively, as shown in Table 4.1.

<table>
<thead>
<tr>
<th>Messages</th>
<th>Routing spaces</th>
<th>Default profiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComA message</td>
<td>componentRouting</td>
<td>Inside a component</td>
</tr>
<tr>
<td>MA message</td>
<td>agentRouting</td>
<td>All Conductors</td>
</tr>
<tr>
<td>Control message</td>
<td>controlRouting</td>
<td>All Actors and Conductors</td>
</tr>
<tr>
<td>Data objects</td>
<td>objectRouting</td>
<td>All Actors</td>
</tr>
</tbody>
</table>
4.2 Module Agents

In the Conductor, a Module Agent incorporates a reasoning kernel to realise the intelligence for controlling the component behaviour. The reasoning kernel contains five parts: a Capability for describing the basic functionality of the component, a Story for describing the interaction constraints with the other components, a World Model for tracking the state of the external world, a set of Control rules for searching activity and co-ordinating with the other peers, and a Reasoning engine for interfacing with the event interpreter. The basic structure of a MA has been discussed in the previous chapter.

4.3 Putting it all together

4.3.1 Current implementation

The implementation of HLA specification Version 1.3 and the RTI Next Generation Version 5 is used as the underlying software bus. Using HLA terminology, an ISS is called a federation, each ComA represents a federate, and the RTI is the physical
world for the agents. The sensors and effectors of a ComA are realised by the local RTI library (libRTI). The data object manager uses the Object Management services to declare shared classes, to update object values and to reflect the changes of the objects from the RTI. The distribution of data objects is realised using the Data Distribution Management services and time Management services. The run-time system is managed using Federation Management services.

The basic information of a ComA, such as name, type and the location of the capability specification is described in a structure called ComARole, which is used to initialise the kernel of an agent. The event-processing loop only starts after the ComA has been initialised as a federate and has joined a federation. Fig. 4.2 shows the basic lifecycle of a ComA. The reasoning kernel of an MA is written in Prolog. In an MA, the task interpreter is implemented using C++; it is coupled with the reasoning kernel using a client-server style. The interface is realised using the Logic Server of Amzi Prolog [139].

![Diagram of ComA lifecycle](image)

**Figure 4.2: A detailed lifecycle of a ComA.**

### 4.3.2 Actor and Conductor

Using the ComA and MA, the Actor and Conductor can be constructed. An Actor wraps the legacy assets of a simulation or an interactive visualisation system. The data structures are encapsulated as internal/shared data classes, and the computational routines are incorporated as actions which are registered in the lookup table of the ComA. Following general engineering principles, the development of an Actor takes a number of steps, e.g. requirement analysis, capability specification, code incorporation, validation, and executable generation. In the next chapter we will discuss these issues with a test case. A Conductor contains a ComA and a MA. The task
4.3 Putting it all together

The processing loop of the ComA and MA are merged, and the lookup table of the ComA only contains the communication-related actions.

The Conductor of a component is equipped with a user interface, which can be launched optionally at run time. The interface presents basic execution information of ComAs, reasoning procedures of MA, data objects' states, the states of the other MAs and a tool for interacting with the other users. Fig. 4.3 shows a snapshot. In the next chapter, we will discuss the utilisation of this tool in more detail.

![Figure 4.3: The snapshot of the GUI of a Conductor.](image)

### 4.3.3 Capability and story descriptions

Using the ISS-Conductor architecture, an ISS can be constructed by instantiating suitable components and describing the interaction constraints between the component instances. Each component has an explicit description of its capability. XML schema [110] based templates are defined for describing the capabilities of components, and the interaction story between component instances. The capability template contains sections for describing the data classes, the actions and states, and the activity-transition graph. It is derived from the Object Model Templates (OMT) provided by HLA for documenting the object models for simulation and federation.

Using the specification, a tool called isscTempG is provided to generate the source framework of the component. The story template allows the user to describe the participating component instances, the projections between their shared classes, and the activity flows.

### 4.3.4 Run-time configuration files

The design of the ISS-Conductor treats HLA and the Amzi Logic Server as black boxes, their kernels are not modified in the ComA and MA. The run-time configuration files required by HLA and the Logic Server e.g. the specification file for the
federation object and the configuration file for the RTI, and the configuration file for the Amzi logic server, are generated automatically from the specification files of capability and story, as shown in Fig. 4.4.

![Graphical representation of the file generation process](image)

**Figure 4.4: Generating run-time configuration files.**

### 4.4 Performance analysis

The performance study of the current implementation focuses on communication-related issues, such as latency and throughput for ComAs to remotely update shared objects, and the reasoning related issues, such as overhead for the Module Agent to take decisions on activity control. We use these experiments to briefly evaluate the implementation.

#### 4.4.1 Example components and the test bed

We construct two example components using the current implementation of ISS-Conductor. A *Producer* component (*Producer* for short) maintains a shared object, *dataObj*, which has an adjustable-size attribute called *dataField*. The *Producer* has four actions: *init* for resetting the size of *dataField*, *adaptSize* for increasing the size of *dataField*, *exportData* for updating the value of the object attributes, and *stop* for exiting the system. Each invocation of *adaptSize* doubles the size of *dataField*. A *Consumer* component (*Consumer* for short) contains three actions: *init*, the initial action, *consumeData* for receiving data objects, and *stop* for exiting the system. Fig. 4.5 shows the capability of these two components. Using these two components, we build a simple communication scenario for updating shared data objects.

The test bed is the super computer of the Dutch ASCI research school (DASII) [140], which contains 200 Pentium III nodes, distributed in 5 clusters. The clusters are connected via SurfNet [141]. Each node has one Myrinet card [142] and one fast Ethernet
4.4.2 Delay for remote updating shared objects

The first thing we studied is the delay of updating shared objects. The services for updating shared data objects and for reflecting their changes are provided by the Object Management services from the RTI, which are realised using TAO, a real-time implementation of CORBA [121]. At its basis, the connections use TCP sockets over a fast Ethernet.

Fig. 4.6 shows three phases for updating a shared object, first the ComA locally updates the object contents locally, then pushes the changes of the attribute value to remote ComAs which have subscribed to the data class, finally the remote ComAs reflect the data from the local RTI library. The delays of the local-update and the reflection include the overhead of the data object manager in the ComA, and the encoding and decoding of the RTI services. The delays of the remote update include the overhead of the RTI Object Management services, TAO and the TCP sockets. We
measure these delays at the ComA layer and treat them as one. For comparison purpose, the communication performance of a pure TCP sockets has also been measured. In the experiment, the RTI execution is launched in the fileserver of the cluster das2.nikhef.nl, and two nodes (node201 and node202) in the cluster are used for executing Consumer and Producer respectively. Each measurement takes 23 time steps, which starts from the attribute size of 16 bytes and doubles the size after each step until 64 Mega bytes. In total, the measurement has been done 50 times.

From the measurements, we observed a number of things. First, the delay for a remote update is larger than the local updating and reflection, as shown in Fig. 4.7. Second, the RTI introduces certain overhead on the communication; for small size objects (smaller than 8K bytes) the delay remains nearly constant as 0.002, which is larger than transferring data using TCP Sockets, as shown in 4.8. Third, the delay for remotely updating linearly increases when the size of the data object increases. In the figures, we observed strange curves in the measurements, for both the RTI and TCP sockets, which are inherent to the Ethernet cards on the DAS II system. Improvements are expected when the system is upgraded [143]. Fourth, for large objects, the throughput of the remote update can achieve 8 Mbytes per second, which is comparable to the throughput of TCP sockets.

![Figure 4.7: The delays of the local update and reflection and for the remote update of a shared object. The error bars indicate the standard deviation at each step.](image)

**4.4.3 Location of the RTI execution**

In distributed problem solving environments, the RTI execution is a software resource, which can be shared by different applications. How the location of the RTI execution influences the remote update of shared objects is an important issue for executing an ISS. In this experiment, we measured the remote update between two

---

1The cluster of das2.nikhef.nl is located at University of Amsterdam; fs2, and node201 and node202 are in this cluster.

2We did not observe the similar behaviour from the measurements between two local Linux workstations. The discussion on this particular issue is out of the scope of this thesis.
nodes as in the previous experiment, but run the RTI execution in three different locations, as shown in Fig. 4.9. From the measurements, we find that the delays for the remote update are reasonably close to the original system, as shown in Fig. 4.10. It shows that the location of RTI execution does not have a clear influence on the remote update of shared objects.
4.4.4 Remotely updating objects to multiple Consumers

In an ISS, a shared object is often updated and consumed by more than one consumer, the correlation between the update delay and the number of consumers is important for analysing the overall performance of the ISS. We run the test case in four configurations, one producer with one, two, four, and eight consumers respectively. The experiments are executed in a single cluster; the delays of the remote update between the producer and each consumer are measured. When a configuration contains more than one consumer, the delay $T_R$, which is defined as the range from the moment that the first consumer starts the reflection until the moment that the last consumer finishes, is also measured.

We first looked at the mean of $T_R$. The measurements clearly show that the $T_R$ increases when the size of the object attribute increases and when the number of consumers increase, as shown in Fig. 4.11. The error bars indicate the standard deviations at each time step.

![Figure 4.11: The comparison of the $T_R$.](image)

By analysing the updating delay for each consumer, we found the remote update occurs in a one-by-one manner, which starts from the first consumer that joins the federation and ends with the last consumer, as shown in Fig. 4.12. An important reason for being so is that the RTI being used in the experiment does not support the multicast over TCP sockets. But the delay for the Nth consumer is smaller than N times the first consumer, as shown in Fig. 4.13. Because the local RTI library in the producer omits the operations for initialising remote updates when there are more consumers. The delay between the producer and the first consumer is consistent within the range of standard deviations, see Fig. 4.14.

4.4.5 Message passing

Messages are another type of data being exchanged between components. In ISS-Conductor, a typical message size is 90 bytes. The RTI of HLA treats the message distribution differently from the object distribution because of the non-persistency; but the underlying communication services are both based on sockets. Using the TCP sockets, messages are distributed to multiple receivers in a same manner as in data
4.4 Performance analysis

Figure 4.12: The remote update of all eight consumers.

Figure 4.13: Compare the remote update delay of the 8th consumer and 8 times delay of the first consumer.

Figure 4.14: The remote update of the first Consumer in the federation in different configurations.

objects distribution. Fig. 4.15 shows the delay of sending a message to four receivers (each receive in a separate host).

We studied how a federate simultaneously handles both object distribution and message passing. A scenario, as shown in Fig. 4.16, is used in the experiment. Since we knew that both objects and messages are delivered sequentially, when the size of object is large, the message sent from Consumer A will arrive at the Producer A and
Consumer B when they are still sending and receiving respectively. We did the measurement in a same number of iterations as the previous scenarios. Fig. 4.17 shows the averages of 50 measurements. From the results, we see a federate has a delay in receiving income messages when it is sending large size data objects, but it has little influence when it is receiving data objects. The local RTI library of a federate handles the income messages before receiving the data objects, as shown by the curve in the Consumer B. This helps agents to respond to the incoming event in real time.

Finally, we studied the correlation between the structure of an object model and its remote update delay. In the experiment, we only focus on the object models with different number of attributes. We modified the object model in the previous experiment to have 16 attributes. And run the experiments in 4 configurations, in which the Producer updates 1, 2, 4, 8 and 16 attributes respectively and the attributes have equal sizes. In each configuration, the total size of the data attributes remains same. Fig. 4.18 shows the results of 50 measurements. We can see that the number of attributes does not influence the remote update of the object. It implies that the total size of the attributes is the main issue influencing the remote update delay.
4.5 Performance for action reasoning and story execution

In this section, we focus on three main issues: first, the overall quality of the Module Agent, e.g. its correctness for making decisions on actions, second, the overhead of the
reasoning kernel, e.g. the delay for searching solutions, and third, a brief comparison of the performance of the three execution paradigms.

The benchmark story employs six roles, in which Producer A (PA), Producer B (PB), and Producer C (PC) are instances of the Producer, and Consumer A (CA), Consumer B (CB), and Consumer C (CC) are instances the Consumer. The story, as shown in Fig. 4.19, has three scenarios. The first scenario, called Scenario A, involves three roles: Producer A, Consumer A, and Consumer B. The Producer A exports data for the Consumer A and Consumer B, and continues after they finish the consumption, as shown in Fig. 4.20. The second scenario, Scenario B, involves Producer A, Producer B, and Consumer C. In this scenario, Producer A and Producer B have critical transitions SbT7 and SbT8, as shown in Fig. 4.21. And the third scenario, Scenario C, involves Producer A, Consumer A, Producer C, and Consumer C. This scenario has a dependency tree which is more than two layers, as shown in Fig. 4.22. The scenarios are switched in the sequence of Start Story, Scenario A, Scenario B, Scenario C and End Story. The dataObj is mapped to Object1 in Producer A, Producer B, Consumer A and Consumer B, and is mapped to Object2 in Producer C and Consumer C.

![Figure 4.19: Benchmark story. It has three nested scenario nets: T2 (scenario A), T3 (scenario B) and T4 (scenario C).](image)

The experiment has been executed in a single cluster of the DASH. Roles are executed in separate nodes; one role, both its Actor and Conductor, is executed in one node. The producer A is the responsible role for the initial scenario (scenario A). At run time, ISS-Conductor generates a number of log files. The MAs generate a reasoning log file, which contains all the queries to the reasoning kernel, and the time cost for each query. We study the performance using these log files.

### 4.5.1 Overall observations on the action reasoning

From the perspective of the reasoning engine, all the facts of the Capability and Story and the rules for execution control are represented as Prolog terms. Table 4.2 shows a number of terms in the instances of Consumer and Producer.

The story is forced to execute in a distributed, hierarchical and centralised execution paradigm respectively. Each paradigm has been executed 20 times. From the experiment, we observed the following; first, the federates of all the roles (Actor and
4.5 Performance for action reasoning and story execution

Figure 4.20: Scenario A (involved roles: Producer A, Consumer A and Consumer B). Using the publish and subscribe mechanism, a data object can be simultaneously consumed by multiple consumers.

Figure 4.21: Scenario B (involved roles: Producer A, Producer B, Consumer A). SbT7 and SbT8 are two critical transitions.

Conductor are two federates, and there are in total 12 federates) can successfully join and resign from the story federation. Second, all the Conductors can correctly find the actions for their Actor (by comparing the activity sequences with the scenario marking-graph). And finally, the roles in each scenario can successfully adapt their
Implementation and performance analysis

Figure 4.22: Scenario C (involved roles: ProducerA, ConsumerA, ProducerC and ConsumerC).

Figure 4.23: Topologies of the routing spaces of the involved roles. The roles that are not involved in the scenario set their routing spaces using the away from the others profile.

routing spaces, non-involved roles do not receive state information from the scenario, and involved roles can adapt their scenario in the correct way.

4.5.2 Overhead of the reasoning kernel

The overhead of the reasoning kernel is defined as the delays for processing the events that are received from the task interpreter, e.g. for reporting observations and for
4.5 Performance for action reasoning and story execution

Table 4.2: Facts and rules in the knowledge base of the Consumer and Producer components. The story facts and reasoning rules are consistent for all the component instances.

<table>
<thead>
<tr>
<th></th>
<th>Capability facts</th>
<th>Story facts</th>
<th>Reasoning rules</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Consumer</strong></td>
<td>20</td>
<td>154</td>
<td>380</td>
</tr>
<tr>
<td><strong>Producer</strong></td>
<td>28</td>
<td>154</td>
<td>380</td>
</tr>
</tbody>
</table>

querying new actions. The events for all three scenarios are traced in Producer A; the total numbers and the time cost for each processing are shown in Fig. 4.24.

<table>
<thead>
<tr>
<th>Execution paradigm</th>
<th>Centralised</th>
<th>Distributed</th>
<th>Hierarchical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total events</td>
<td>6720(±2%)</td>
<td>5100(±2%)</td>
<td>4200(±2%)</td>
</tr>
<tr>
<td>Processing cost</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(seconds)</td>
<td>Min</td>
<td>Average</td>
<td>Max</td>
</tr>
<tr>
<td></td>
<td>&lt;0.0001</td>
<td>0.002</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td></td>
<td>(0.003)</td>
<td>(0.003)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.24: The total number of events and the time cost for processing an event.

In the centralised model, the co-ordinator has to search activities for all the other peers and maintain the states for them; therefore the total number of events is much higher than the distributed and hierarchical paradigms. In the hierarchical paradigm, irrelevant events are filtered, the total number of events is less than for the other two.

From the results, we can see the maximum delay for querying is less than 0.05 seconds, and the average cost is about 0.002 seconds which is comparable to the latency of the object distribution in ComAs (0.002 seconds). From this point, we can say that the reasoning kernel does not necessarily introduce a bottleneck of the system performance.

4.5.3 Reasoning complexity and delay

In this section, we briefly analyse the complexity of reasoning procedures in an MA and figure out the upper bound of the delays. In general, the reasoning engine accepts two types of requests from the task interpreter, see the architecture of MAs in Fig. 2.4: asserting observations to the world model, and querying activities for the Actor. The delay for the first type of operations is inherent to the implementation of the Prolog reasoning engine; the update of the dynamic database in the Prolog reasoning engine causes the overhead. The delay for the second type is related to the description of a scenario net, capability, and the number of involved roles. Since the operation for searching a doable action is frequently invoked in the second type for control run-time behaviour, we use the computing complexity of this operation to analyse the reasoning delay.
According to the algorithm we discussed in chapter three, searching a doable activity takes two steps. It first finds an enabled transition from a scenario net, and then finds a doable action from the capability for the transition. To find an enabled transition from a scenario net, the reasoning engine needs to traverse the transitions in a scenario net, and to scan the places in the pre set of a transition, therefore the complexity is $O(N_t \cdot N_p \cdot (N_{pg} + N_{tg}))$. $N_p$ and $N_t$ denote the number of places and transitions, $N_{pg}$ and $N_{tg}$ denote the maximum number of guide expressions in places and relation links using, and $N_r$ denote the number of roles. Finding a doable action in the capability can be two cases. If the action in the transition is doable return the action, the searching cost is merely for finding the fact from the dynamic database and evaluate the expression, if it has one, therefore the complexity is $O(N_a \cdot N_{ag})$, in which $N_a$ and $N_t$ respectively denote the number of activities and transitions in the capability, and $N_{ag}$ denotes the maximum number of guide expressions in links. Otherwise the reasoning engine has to find another action in the capability which leads a path to the action; in Prolog, using a recursive algorithm to find the path between two nodes in a graph, the complexity is $O(P_T \cdot N_{ag})$, $T$ is the possible intermediate actions in the path. In distributed and hierarchical paradigms, each role does all the search procedures; for the centralised paradigm, the co-ordinator not only searches actions for itself but also the enabled transitions for all the other involved roles, and the other peers only searches doable activities. We can consider the number of guide expressions as a constant, and the evaluation of the guide expressions is mostly inherent to the Prolog reasoning engine. The complexities of searching procedure are thus depicted in Table 4.3.

<table>
<thead>
<tr>
<th>Execution paradigm</th>
<th>The best case</th>
<th>The worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralised paradigm</td>
<td>$O(N_r \cdot N_p \cdot N_t) + O(N_a)$</td>
<td>$O(N_r \cdot N_p \cdot N_t) + O(P_T \cdot N_{ag})$</td>
</tr>
<tr>
<td>Co-ordinator</td>
<td>$O(N_a)$</td>
<td>$O(N_a)$</td>
</tr>
<tr>
<td>Peers</td>
<td></td>
<td>$O(N_a)$</td>
</tr>
<tr>
<td>Distributed paradigm</td>
<td>$O(N_p \cdot N_r) + O(N_a)$</td>
<td>$O(N_p \cdot N_r) + O(P_T \cdot N_{ag})$</td>
</tr>
<tr>
<td>Hierarchical paradigm</td>
<td>$O(N_p \cdot N_r) + O(N_a)$</td>
<td>$O(N_p \cdot N_r) + O(P_T \cdot N_{ag})$</td>
</tr>
</tbody>
</table>

Using Prolog to implement the high level searching strategies, the programming itself is easier and flexible, but the price is the low efficiency of searching itself. In the analysis, we do not include the assumption on the order of clauses and other tricks, because most of them require knowledge on the capability and scenario net. From the analysis, we see that the performance can be principally improved in a number of ways. First, a sufficient detail scenario net makes the searching cost for doable action close to the best case; a small $T$ means the reasoning engine does not need to find many intermediate activities. Second, a complex scenario net can be divided into a number of smaller sub nets, so that both $N_t$ and $N_p$ are small.
4.54 Brief comparison between execution paradigms

We study the execution paradigms by comparing their execution time for each scenario. Since Producer A is the responsible role for all the scenarios, it's time costs for each scenario when using different paradigm are compared. Fig. 4.25 shows the mean of the 20 executions.

![Bar chart showing execution time for different paradigms]

**Figure 4.25:** The time cost by Producer A for each scenario in different execution paradigms. The error bars show the standard deviations.

From the results, we clearly see that the execution paradigms do influence the performance of the execution. When the scenario (scenario B) contains critical transitions, the centralised paradigm has a better performance than the other two. When the scenario (scenario C) has a dependency tree which has more than two layers, the hierarchical paradigm can get a better performance. An important reason for that is that when the dependency tree is deep the centralised paradigm does not exploit the parallelisation between the loosely dependent roles. It also introduces a load-balancing problem in the central co-ordinator; the central co-ordinator has to do many more queries than the other two paradigms. That is also the reason that when the number of roles is small as in scenario A, the centralised paradigm can achieve a better performance than the other two.

Between the distributed and hierarchical paradigm, the performance difference is not remarkable when the dependency tree is only two layers, as in scenario A and B. We can clearly see that the hierarchical paradigm reduced the number of messages in scenario C, as shown in Fig. 4.26.

4.5.5 Summary

From the experiments, we can say the current implementation of the reasoning kernel can correctly realise the interpretation of a story, and the MAs can successfully control the system execution. The system performance is influenced by the design of the scenario and the selection of the execution paradigm. The reasoning kernel has a comparable latency to the ComAs, and is not necessarily a killer for the system performance.
4.6 Discussion and conclusions

4.6.1 Evaluation

In this chapter, we have discussed the implementation of Communication Agents and Module Agents and have studied the associated performance issues. In ComAs, the services provided by the RTI of HLA are used for the underlying data communication. In the MAs, the control intelligence for activity control and decision making is realised using a logic language (Prolog). Its reasoning engine supports the solution searching and maintenance of dynamic databases.

An original goal of ISS-Conductor is to provide an architecture which can efficiently interconnect simulation and visualisation programs, and can support the rapid prototyping of interactive simulation systems. We have not deployed ISS-Conductor in real cases of simulation and visualisation systems. Yet, from the experimental results presented in 4.4 and 4.5, we can evaluate the development from following aspects:

1. We have implemented the agent framework discussed in chapter two. By constructing the interaction story of the test case, we see that using the ISS-Conductor components, the system behaviour can be adapted at the story level and does not demand modifications to the component kernels. The logic of the system behaviour can be adapted at the story level.

2. We have realised the story execution mechanisms discussed in chapter three. In 4.5, we have executed the scenarios in distributed, hierarchical and centralised modes. From the experiments, we see that the current implementation of the reasoning kernel can correctly realise the interpretation of a story, and the MAs can successfully control the system execution.

3. In the experiments, we have also studied a number of performance characteristics of the implementation. The Communication Agents add limited overhead on the data transmission; in general, the latency of transferring messages or small size objects is acceptable for soft real-time interaction. As we have mentioned, the DoD's RTI does not support hard real-time distributed simulations [67].
comparable throughput to pure TCP sockets when transmitting large size objects. Of course, the transmission can be optimised at application level; in the next chapter we will show how parallel data producers improve the transmission delay.

From the experimental results, we can say that the implementation of ISS-Conductor fulfils the basic design requirements.

### 4.6.2 Conclusions

The discussion leads following conclusions:

1. High Level Architecture (HLA) provides a flexible interface for implementing ComAs. The RTI services provide a standard way for accessing and updating shared objects, and for adapting the multicast group between the roles. The RTI services can support the real-time interaction between ISS modules.

2. Separating the control of the run-time interactions from the functionality of the system modules improves the reusability of the constituent system components and the adaptability of the overall system behaviour.

3. By benchmarking the Module Agents, we see that the reasoning delay of Prolog is comparable to the communication latency in the test case. By analysing the reasoning complexity of the implementation, we can see the performance is dependent on the complexity of scenario net, and the scenario net can be simplified by dividing into smaller sub nets. Therefore, the reasoning is not necessarily the bottleneck for the system performance.

In the next chapter, we will use a test case to demonstrate the main features of ISS-Conductor.