Nationaal Lucht- en Ruimtevaartlaboratorium
National Aerospace Laboratory NLR

NLR TP 96220

A survey of the NARSIM C/S middleware

R.W.F.J. Michiels

DOCUMENT CONTROL SHEET

	ORIGINATOR'S REF.		SECURITY CLASS.
	TP 96220 U		**Unclassified**

ORIGINATOR
National Aerospace Laboratory NLR, Amsterdam, The Netherlands

TITLE
A survey of the NARSIM C/S middleware

PRESENTED AT

AUTHORS	DATE	pp	ref
R.W.F.J. Michiels	**960531**	**57**	**17**

DESCRIPTORS

Air traffic control
Applications programs (computers)
Architecture (computers)
Client server systems
Computer systems design
Distributed processing
Interprocessor communication
Modularity
Protocol (computers)
Simulators
Unix (operating system)

ABSTRACT
This document describes the Client/Server system architecture of NARSIM - the NLR Air Traffic control Research Simulator- focussing on the so-called middleware: the infrastructure that enables all the seperate parts of the simulator to interact with each other.
To show the full scope of the architecture, first a somewhat abstract architecture description is given, followed by more concrete technical implications of this architecture and the particular implementation chosen within NARSIM. Also some non-technical issues are discussed which are direct results from either the architecture or the specific implementation. Finally in the appendices a survey of the NARSIM C/S servers is given together with other system documentation, to give a better understanding of the environment in which the NARSIM C/S middleware operates.
The most distinctive feature of the NARSIM Client/Server architecture is probably the perfect symmetry between clients and servers. An application cannot be labeled statically as either server or client: only with respect to a certain dialogue can one application be a labeled as a server and the other as a client. Initial experience with the new architecture has shown that this perhaps seemingly arbitrary view of clients and servers has some profound positive consequences on the architecture as a whole, such as a very low degree of complexity and thus ease of understanding and maintenance.

Summary

This document describes the Client/Server system architecture of NARSIM—the NLR Air traffic control Research Simulator—focussing on the so-called *middleware*: the infrastructure that enables all the separate parts of the simulator to interact with each other.

To show the full scope of the architecture, first a somewhat abstract architecture description is given, followed by more concrete technical implications of this architecture and the particular implementation chosen within NARSIM. Also some non-technical issues are discussed which are direct results from either the architecture or the specific implementation. Finally in the appendices a survey of the NARSIM C/S servers is given together with other system documentation, to give a better understanding of the environment in which the NARSIM C/S middleware operates.

The most distinctive feature of the NARSIM Client/Server architecture is probably the perfect symmetry between clients and servers. An application cannot be labeled statically as either *server* or *client*: only with respect to a certain dialogue can one application be labeled as a server and the other as a client. Initial experience with the new architecture has shown that this perhaps seemingly arbitrary view of clients and servers has some profound positive consequences on the architecture as a whole, such as a very low degree of complexity and thus ease of understanding and maintenance.

Contents

(51 pages in total)

List of figures

Abbreviations

AAA	Amsterdam Advanced ATC (operational LVB system)
ACOD	Area conflict detection tool (server), see section A.1.2.1
API	Application program interface
ATC	Air-traffic control
ATCo	Air-traffic controller
ATM	Air-traffic management
ATN	Aeronautical telecommunications network
CAER	Computer Assistance for Enroute ATC
CMS	Common modular simulator, see Ref. 12
CORBA	Common object request broker architecture, see Ref. 11
CPDLC	Controller pilot datalink communication
CRDA	Converging runway display aid
CTAS	Centre TRACON automation system, see section A.1.2.5
DIS	Distributed interactive simulations, see also A.1.5.2
FIR	Flight information region
FPM	Flight position monitor (server), see section A.1.2.3
HMI	Human-machine interface (also MMI: man-machine interface)
HTML	Hypertext markup language, see Ref. 2
IFP	Initial flightplan (server), see section A.1.3.1
ILS	Instrument landing system
IPC	Inter-process communication
LAR	(Dutch) long range radar
LVB	Luchtverkeersbeveiliging (Dutch civil aviation authority)
NARSIM	NLR ATC research simulator
NCS	NARSIM Client/Server communication system
NLR	Nationaal Lucht- en Ruimtevaartlaboratorium (National Aerospace Laboratory)
PATN	PHARE Aeronautical telecommunications network.
PATs	PHARE advanced tools
PHARE	Programme for harmonised ATM research in Eurocontrol
PLAID	Programming language independent data tool, see Ref. 8
RADNET	Radar data network
RFS	Research flight simulator (at NLR)
RPC	Remote procedure call
SA/RT	Structured analyses for real-time systems (design standard)

SIMICA	Simulation of interactively controlled aircraft (server), see section A.1.4.1
SPL	System plan (server), see section A.1.3.2
SRT	System recording tape (server), see section A.1.4.3
SSR	Secondary surveillance radar
STCA	Short term conflict alerting tool, see section A.1.2.2
TCP/IP	Transport Control Protocol/Internet Protocol, see also figure 6
TP	Trajectory predictor (server), see section A.1.2.4
UTC	Universal co-ordinated time
IFR	Instrument flight rules
WWW	World wide web, see Ref. 1
X11	X Window System, Version 11
XDR	External data representation, see Ref. 15

1 A brief history of NARSIM

Before describing the ins and outs of the NARSIM C/S middleware this section gives a brief survey of the history of NARSIM, the NLR air traffic control research simulator. For more detailed information about NARSIM see also Ref. 10.

It all began in 1987 when in the course of a feasibility study the first ideas for an NLR air traffic control (ATC) research simulator were put on paper. These ideas described an air-traffic generator, pseudo-pilot interfaces, a ground system and human-machine interface (HMI) software for an air traffic controller (ATCo).

Not much later, the Dutch civil aviation authorities (LVB) contacted NLR to develop a short-term conflict alerting (STCA) algorithm—a safety net to assist air-traffic controllers—specifically for the (crowded) Dutch airspace. After developing a first off-line version of the STCA algorithm it was decided that a real-time environment was required for proper development and validation of the algorithm.

After the air-traffic controllers had worked with the real-time STCA algorithm the LVB also wanted to evaluate their medium-term conflict detection tool—a controller assistance tool (ACOD) with a time horizon of approximately 30 minutes—in a real-time environment. A trajectory predictor was developed mainly for this purpose, and a flight path monitor was added as well to create an environment in which the ACOD algorithm could be properly evaluated.

During all this time the ATCo HMI (Human-Machine Interface) was being used extensively to see if raster-scan displays were a viable alternative to the operational vector displays. Prototyping of the Amsterdam Advanced ATC (AAA) ATCo HMI was done using NARSIM, also including the interactions with ATM tools such as STCA and ACOD.

New operational procedures to use converging runways under instrument flight rules (IFR) conditions were evaluated using NARSIM generated air traffic, multiple pseudo-pilots, and two executive air traffic controllers. The use of a “sideview” for executive controllers to complement the default “God’s eye” radar view was tested using NARSIM and will be used in the AAA system.

Other projects that have used NARSIM include the Federal Aviation Authority’s datalink experiment in which the NLR’s Research Flight Simulator (RFS) and NARSIM were linked up to evaluate the use of digital datalink between pilots and controllers as a replacement for voice communications, and the Annette project linking international ATC simulators and flight simulators (Ref. 13).

NARSIM is currently also being used for in-house training of ATM engineers to teach the principles of operational air-traffic control. The future for NARSIM will focus on multi-site distributed simulations involving several European establishments, incorporating advanced controller assistance tools and the further development of a 4D air-traffic server.

Over the last few years, the NARSIM team has grown from only a few to more than a dozen people. Because it has always been attempted to build upon the results and applications from previous projects, by the end of 1994 the complexity of the system had increased beyond any single person's grasp.

2 NARSIM C/S predesign

The NARSIM software structure had thus become too complex and too rigid to support new developments in the field of air-traffic control research. To overcome these problems the NARSIM software had to be restructured.

The reasons for restructuring the NARSIM software architecture are manifold, but all boil down to the reduction of the system's complexity. Less complex systems are easier to understand, easier to expand and easier to maintain. In the old situation it could take almost a year to get to know the system well enough to integrate new tools, expansion had become nearly impossible, and most of the NARSIM budget was spent on maintenance. A new software architecture was required to solve these problems.

2.1 Problems with the old architecture

The old 'architecture' had slowly evolved by writing more and more programs which were glued together with a large block of shared memory. All the programs had read and write access to this shared memory and could thus communicate with each other. Where deemed necessary, other inter-process communication (IPC) mechanisms such as semaphores, signals, pipes, and sockets (to communicate with display programs running on different computers) where also used, albeit without a fixed strategy.

The problems with this architecture were that extensive use of shared memory did not require (or so it was thought) a formal specification of interfaces between programs. Errors in program *X* could easily result in anomalous behaviour in program *Y*. Also the multitude of other IPC mechanisms made interactions between programs very complex and error prone.

2.2 Analyses of other architectures

The high-level CMS (Common Modular Simulator) architecture as defined in Ref. 12 seems very appropriate to serve as a basis for NARSIM C/S and provides the additional bonus of being a somewhat de-facto standard for ATC simulators in Europe. This object-oriented client/server architecture, possibly extended with requirements and data dictionaries from the PATs (PHARE Advanced Tools) project seemed a good starting ground to solve our problems.

Of the other simulators that have been studied, some features of the CAER (Computer Assistance for Enroute ATC) system seemed very desirable. Especially the capabilities offered by the CAERbus—reading, writing and subscribing to data—are thought to be of great value. Also the

used data abstraction (a client does not need to know where the data resides, i.e. which server is managing it) is a very desirable feature.

The CAER architecture itself, however, lacking the flexibility of a true client/server architecture, is thought not to be flexible enough to serve as a basis for NARSIM C/S.

2.3 The selected client/server architecture

The selected client/server architecture as described in the next chapter is based on parts of several different real-time systems, including the ATC simulators described in the previous section. Other systems from which ideas have been “borrowed” are Amoeba (Refs. 16, 17), the Internet RPC protocol (Ref. 14), and the Common Object Request Broker Architecture CORBA (Ref. 11).

More information about the NARSIM C/S predesign can be found in the NARSIM C/S predesign document (Ref. 7).

3 NARSIM C/S: A new architecture

3.1 The Client/Server concept

The basic concept on which the new NARSIM architecture is based is the so-called client/server architecture. In this architecture the whole system is subdivided into several smaller functional units (*applications*, often also called *servers*) which communicate using well-defined APIs (Application Program Interfaces). Each server has a specific task and provides *services* so that other applications (in this context: clients) can use the results (output) of that server. To carry out its task, an applications (now acting as a client) may need to use other applications (servers). What makes the client/server architecture so valuable is that it is easier to manage several small parts individually than it is to manage the entire system as a whole.

Related to the client/server architecture is the data-driven execution model, in which applications (servers) are activated when a request comes in or when (any) new data becomes available. This execution model is implemented in a 'server loop', which repeatedly accepts new requests or other data, calls the appropriate application functions to do the work, and (optionally) sends a reply message.

A straightforward implementation of this model uses the common remote procedure call (RPC) mechanism. The RPC mechanism suits those situations where application *A* (the client) asks for some service from application *B* (the server) and waits for a reply. As a small extension of the usual RPC protocol, *asynchronous* RPC could be used in case no result is expected. In this case the client need not be suspended while the server processes the request.

In a real-time ATC simulator, however, the situation often arises where application *A* (the client) would like to ask application *B* (the server) to send back information more than once. An example of such a situation is when application *A* asks application *B* to send the current position of aircraft *X* whenever that position changes. To accommodate this sort of communication, the RPC mechanism can be extended with a *subscription* mechanism. Using subscriptions, clients can ask a server to send back data (e.g. data updates) asynchronously (e.g. after every update or every *n* seconds).

3.1.1 Available protocols

Summarizing the previous section, three slightly different protocols are available in the communication system.

The first protocol implements *remote procedure calls*: a client does a remote procedure call to a server, sending the parameters encoded in a message to the server. The server decodes the message,

invokes the remote function and sends the result, again encoded in a message, back to the client. The client which has been suspended while the server was doing its work now continues.

The second protocol is a variation on the first one, where the client is not suspended while the server carries out its work, and the server does not send back a reply. This is a case of an *asynchronous remote procedure call*. The advantage is increased performance through asynchronicity—one does not have to wait until the other is finished. The disadvantage is that its use is limited to those cases where no result (return value) is required.

The last protocol, *subscriptions*, is a combination of the former two. The client sends a request (a *subscription request*) to the server, but does not wait for an answer. The server receives the message and adds the client to a list of subscribers for a certain data item or event. Whenever the server modifies a data item or some other event occurs within the server that the client wants to know, the server sends back a reply (a *delivery*) to the subscription request. Subscriptions can thus be used a) when the client does not want to wait for a reply or b) when a single request may result in more than one reply.

From a message-passing point of view the three protocols only differ in the number of replies sent to a single request: a single reply for normal remote procedure calls, no reply for asynchronous remote procedure calls and zero or more replies when using subscriptions. From a programmer's point of view the differences are more subtle, in that remote procedure calls can easily be mapped onto programming language primitives (functions or subroutines), whereas this is not so straightforward with subscriptions. Section 3.2.2 describes just how subscription services are implemented.

3.1.2 Implementing a client/server mechanism

A common way to implement remote procedure calls is to use *stubs* to hide the details of the lower software layers from the application. These stubs can do message encoding and decoding and can route requests to the appropriate server.

Figure 1 shows how the application, stub and communication layer are related and what actions are required to implement a remote procedure call (to invoke a synchronous service). Section 3.5 gives a more detailed description of what the stubs look like.

The client calls a client stub (1) that encodes the request into a message. Next, the client's communication layer is called (2) to send the message to the server's communication layer—possibly on a remote computer (3). The server's communication layer will forward the message to the server stub (4) which decodes the request from the message and calls a server function that performs the

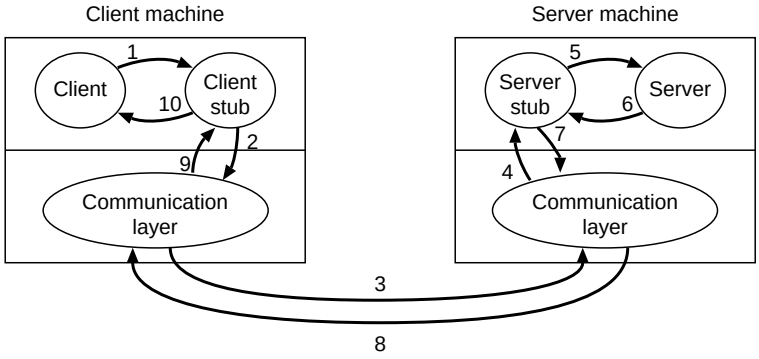

Fig. 1 Client/server communications

requested action (5). The reply is encoded into a message by the server stub (6), and this message is sent back through the communication layers (7, 8). The client stub decodes the message in a reply (9) which is returned to the client (10).

Note that for asynchronous services only steps 1 through 5 are required. For subscription services, steps 1 through 5 implement the subscription request and steps 6 through 10 implement a subscription delivery.

3.1.3 Performance

A frequently heard objection to the use of the client/server mechanism is its inefficiency in terms of speed and network use. Several optimizations exist to reduce the communication overhead introduced by a client/server mechanism. A practical optimization is to run multiple servers on a single computer, or to combine several servers in a single UNIX executable (a *package*). This removes the relatively expensive network transfer, as shown in figure 2a. A more drastic approach would be to remove the client and server stubs altogether and to link all clients and servers into a single process. Unfortunately this approach, as shown in figure 2b, can only be used in special cases (i.c. stand-alone fast-time simulations).

An entirely different approach to reducing network traffic uses *intelligent client stubs* for the implementation of subsequent invocations of idempotent services.[1] The first time a service is invoked, the normal procedure as described earlier is followed. The second time, however, the (intelligent!) client stub recognizes the request and—having remembered the previous reply to this request—

[1] A service is said to be idempotent when multiple invocations always return the same reply. The service with which an application can ask for the location of a beacon is idempotent under the assumption that the beacon is not moved during a simulation.

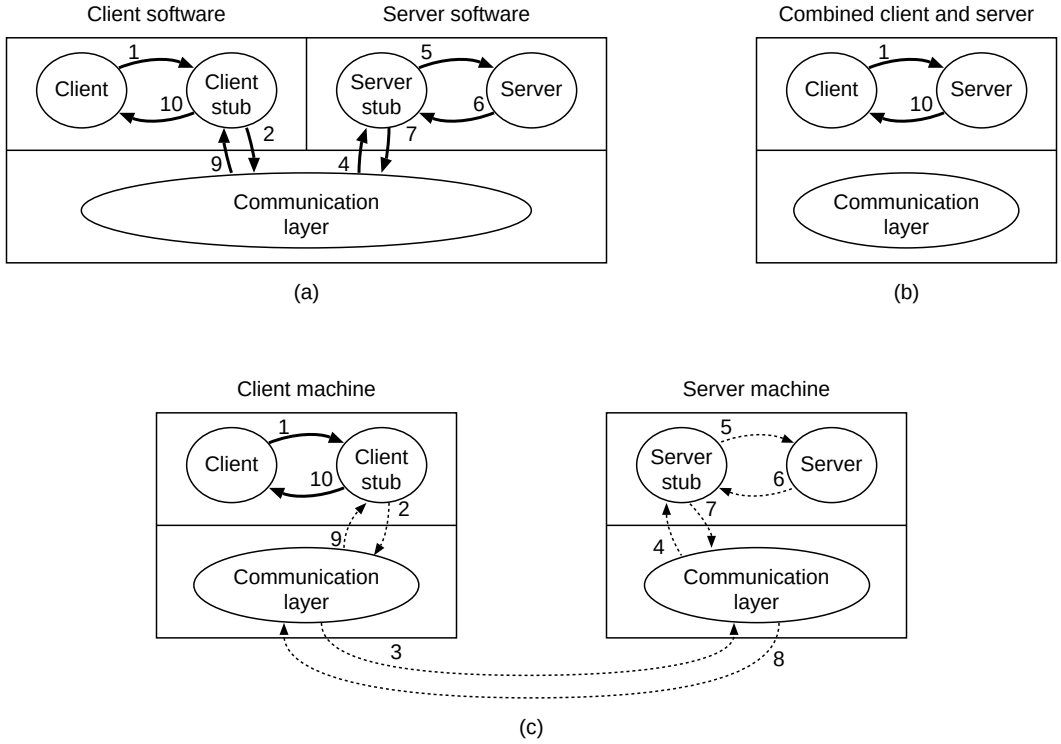

Fig. 2 Optimized client/server communication by (a) sharing the same computer; (b) directly linking clients and servers; (c) caching by intelligent clients.

immediately provides the answer. This whole process is shown in figure 2c. Intelligent client stubs can also be used for slowly changing data, as is for example the case with meteo data. The client stub itself must then subscribe to data updates to keep the local copy of the data up-to-date.

3.1.4 Addressing

When invoking a certain service, clients may want to specify a specific server to handle the service request. In the general case this is even required, as several servers can provide the same service: several ATCo HMI servers will usually provide identical services. A practical problem when sending synchronous requests to more than one server is the way replies are handled. Where the client expects a single reply to his request, several servers could send back possibly different replies.

The only case in which a client need not specify a server when invoking a service is when sending a subscription request. In this case the subscription request is broadcast to all servers providing the named service.[2] It is of course also possible to address a single server explicitly, e.g. when

[2]The communication system keeps track of all of these implicitly addressed subscription requests, so that whenever a new server that also provides this service is added to the simulation, the request is automatically sent to that new server

subscribing to a single object managed by a particular server.

3.2 Implementation of services

The previous sections have given a conceptual view of the actions involved in client/server interactions: *who* sends *what* to *whom*. This section explains *how*. It tells the same story, but this time from an application's point of view. It shows what applications must do to invoke services, and focuses on the flow of control.

Applications can invoke two basic types of services: regular services and subscription services. Both are explained in the next two sections.

3.2.1 Implementation of (regular) services

Regular services are implemented as subroutines or functions. A client can invoke a service by calling the subroutine or function as if it were dealing with a library routine, e.g.

```
GetSPL("KLM123", &spl);
```

Flow of control for regular services between a client and a server is shown in figure 3.

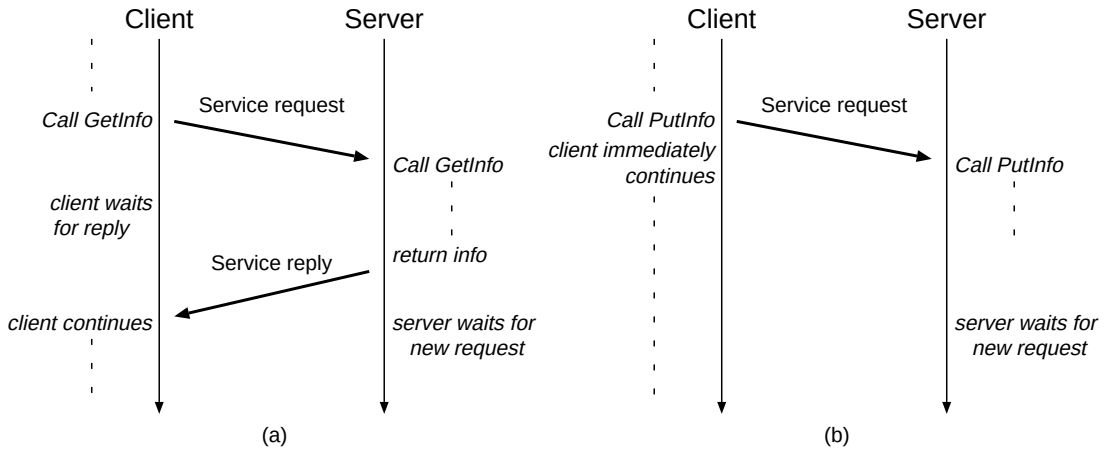

Fig. 3 Flow of control for (a) regular synchronous and (b) regular asynchronous services

For regular synchronous services ("normal RPCs"), the client sends a service request to a server, the server processes the request and sends back a reply to the waiting client. This is the general case depicted in the previous chapter. Regular asynchronous services only involve a single service request from a client to a server. The client immediately continues after sending the request and the server does not send back a reply.

---

as well.

3.2.2 Implementation of subscriptions

Using subscription services is more complex. To subscribe to a particular service, a client asks a server to send a message (*delivery*) when a certain event occurs (e.g. update of system plan for KLM123). Using subscription flags, the client specifies exactly what events it is interested in (e.g. only updates for a flight's EFL and XFL, the executive flight level and the exit flight level). Through a *callback* mechanism the client specifies a subroutine or function that is to be called when an event occurs. With the call

```
SubscribeSPLUpdates(&spl, EFL | XFL, spl_cb);
```

a server subscribes itself to updates of the specified SPL when either the flight's EFL or XFL change. The function `spl_cb` is installed as a callback function and will be called when an update is recevied. This function has two parameters: the updated system plan and flags indicating what fields have been modified (in this case `EFL`, `XFL` or both). When no flags are set, this indicates that the server has ended the subscription and that no more deliveries will be sent. This is usually due to termination of the item that was subscribed to.

Because of the callback mechanism it is not necessary to define which function processes what type of delivery statically. It is even possible to have several functions that process the same type of delivery, only for different objects. Deliveries of track *x* and deliveries of track *y* can be processed by two different (callback) functions.

Flow of control for subscriptions between a client and a server is shown in figure 4.

The client sends a subscription request to the server asking for a subscription to a specified item (e.g. system plan) for updates on some fields (e.g. EFL and XFL). The client does not wait for a reply but continues its work. The server receives this subscription request and records this request in its administration for later use.

When after some time a system plan is updated for which a client has sent a subscription request, the server sends a delivery back to that client. Within the client a callback function is invoked to process the delivery (e.g. on a system plan update, a client might want to update its screen). When later that same system plan is again updated, the last step of the above process is repeated.

See also section 4.1.2 for a more detailed description of how subscriptions can be used.

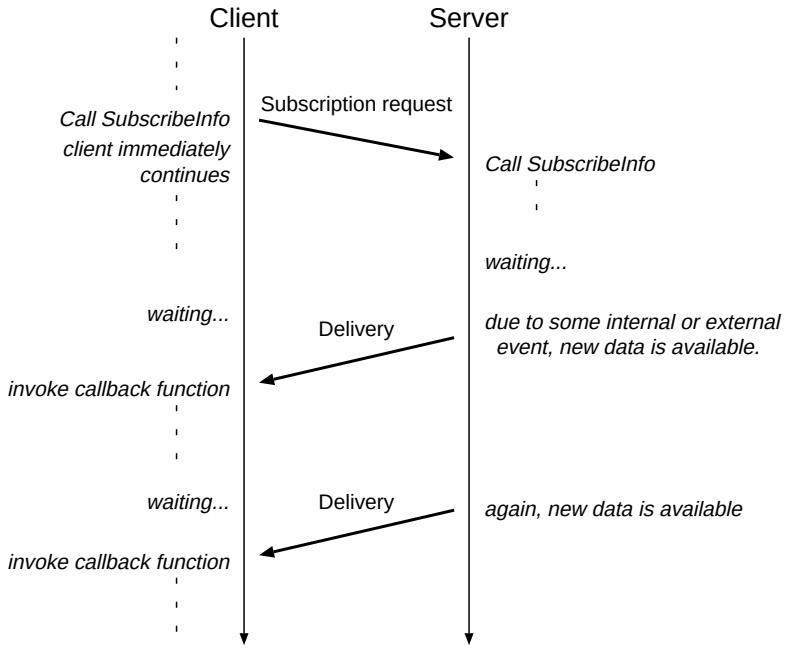

Fig. 4 Flow of control for subscription services

3.3 Interface specification language

Using the new NARSIM architecture, NARSIM consists of a set of applications that communicate transparently using a communication system. When an application is a server (i.e. it provides services), its interface can be defined by the services it provides. The services it uses from other servers are not essential to the definition of the application. For applications that only act as clients, no interfaces need to be defined: they use the interfaces implemented by the servers.

The services that define the interface of a server are described using a formal interface description, also known as an IDL (interface definition language). This description, which is usually part of an IDD (interface definition document), contains all data types used in the interfaces and a list of services that can be provided by each server. A tool has been built to generate client and server stubs from this description. Use of a formal interface description also supports the maintenance of a client/server system and the consistency of the interface's documentation and implementation.

The NARSIM C/S interface specification language is an extension of the XDR (External Data Representation) data description language and specifies the APIs of services. An interface specification file may contain

- XDR data specifications;
- service-set specifications;

- server specifications.

Ref. 15 contains a description of the basic XDR data specifications.

Service sets are used to group related services, usually services operating on the same type of object. A service-set specification contains the logical name of the service set, the service-set number and a list of services. A service is either asynchronous, synchronous or a subscription service, and is defined by a service number, the name of the service, the input parameter type and optionally a return type and an ellipsis (`...`) to indicate a subscription service.

A server specification contains the name of the server, the provided services—including a mapping of services to implementation functions—and the used services.

An example of a partial server specification including a service-set specification, and an XDR data descriptions is shown below.

```
struct Coord {
    restricted float lat ( -90.0: +90.0); /* latitude in degrees */
    restricted float lon (-180.0:+180.0); /* longitude in degrees */
};

services Foo = 3 {
    /* This service set contains a synchronous, asynchronous and subscription
     * service. The argument types can be any type, including basic types,
     * structures, unions, etc.
     */
    1: FooSyncService(float) -> char;
    2: FooAsyncService(int);
    3: FooSubscription(Coord) -> short ...;
};

server BAR {
    /* The BAR server implements some Foo services. It uses no other
     * services from other servers.
     */
    provides {
        General {
            SubscribeLogging = C: SubscribeLoggingHandler, /* lib */
            AbortServer      = C: BarAbort;
        };
        Foo {
            FooSyncService   = C:BarSync;
            FooSubscription  = C:BarSubscribe;
        };
    };
} = C: BarInit;
```

3.4 Hierarchical survey of servers and services

Figure 5 shows how simulations, packages, servers, service sets and services all fit into one picture. At the bottom level, a simulation is identified with an Internet host name and a simulation identification. All applications that want to take part in a simulation must use the same simulation identification to join the simulation. When restricting the scope to a single simulation, at the first levels there may be several packages (UNIX processes) containing several servers, and possible one or more stand-alone servers. The only difference with packages and servers is that a package contains several servers in a single UNIX process, so that communication between servers in a single package is more efficient than communication between stand-alone servers or servers in other packages.

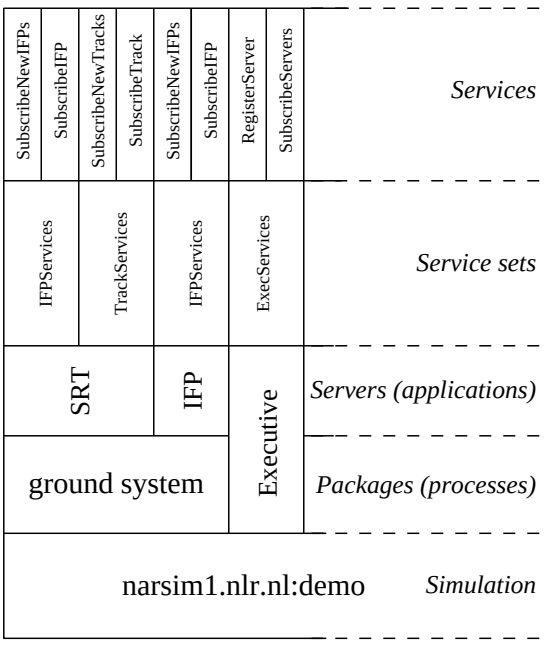

Fig. 5 Hierarchical survey of service

One level higher each server can provide several service sets. A service set is a logical group of services normally associated with a single object type. Using object oriented terminology, a service set can specify an object class. A server does not necessarily have to provide any service sets, although it is a bit strange to call it a server when it can never act as a server in any dialogue.[3]

[3] It would be better to use the term *application* when referring to servers outside the direct context of a dialogue. For historical reasons the term *server* is preferred unless this would cause confusion.

3.5 Structure of the communication system

Figure 6 shows the structure of the communication system and its position between the application code and the operating system (hence the name *middleware*). Only the application code and in exceptional cases the code for the intelligent client stubs has to be provided by the application programmer. All other components are either static or generated by the *ncsgen* program. The generated components in the list below are marked with an asterisk (*).

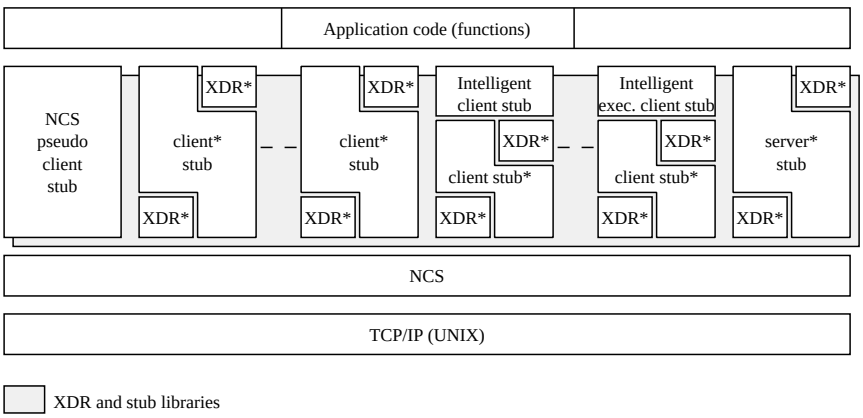

Fig. 6 Communications layered survey

application code (functions) Software that implements the capabilities offered by a server.

server stub* Partly generated and partly static code that marshals incoming requests to the proper application code.

client stubs* Generated code that makes remote servers accessible as if they were a local library by sending/receiving data to remote servers.

intelligent client stubs Software placed between the application and the client stub to improve the performance of the remote server by caching replies to previous requests.

XDR functions* Partly generated and partly static code to encode and decode data structures in a network standard data format.

NCS pseudo client stub A pseudo client stub that makes low-level capabilities of the communication system as described in section 4.8 accessible as if the communication system were a server.

NCS The low-level communication system, hiding all details with sockets and TCP from the rest of the communication system.

TCP/IP The transport level protocol implemented by the UNIX operating system.

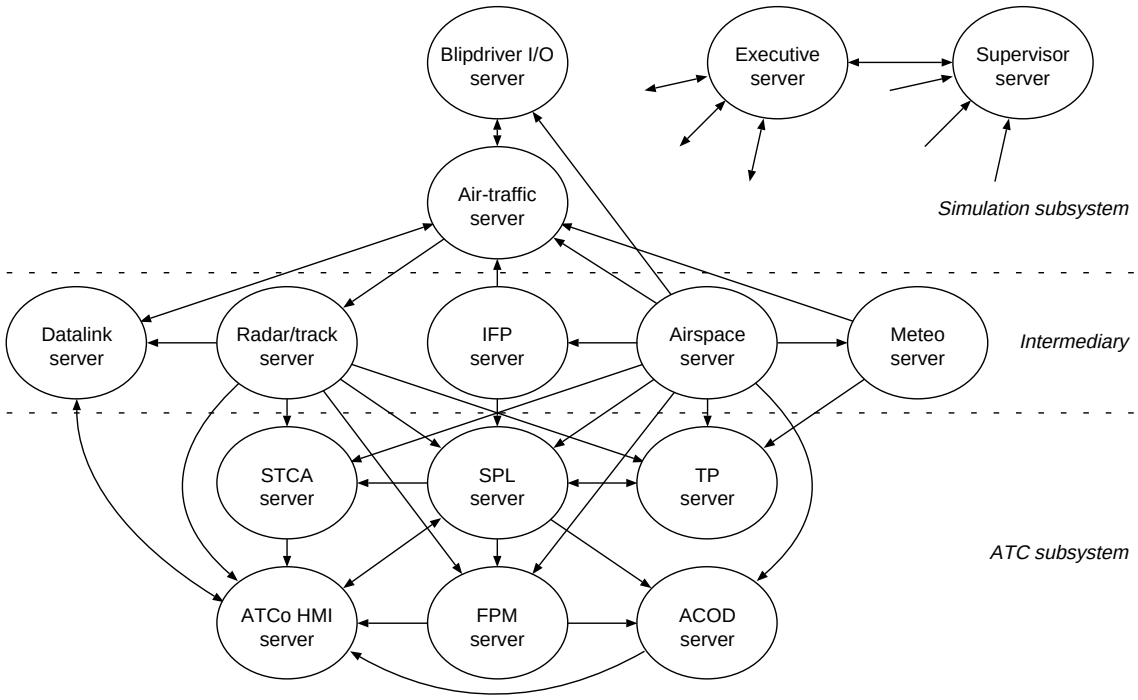

Fig. 7 Partial system subdivision.

3.6 System subdivision

An important part of the architecture is the actual division of the whole system into servers. Figure 7 shows a partial system subdivision, containing the most important servers. Arrows in this figure depict major dataflows in the NARSIM C/S system, where subscription requests are not considered to be "major dataflows." The arrows do not necessarily imply any dependencies between servers: the sender of a message is often not the initiator of a message exchange as most dataflows are deliveries to earlier subscription requests.

For most simulations only a subset of the given servers is used. This subset is selected by the experiment leader using the supervisor HMI as described in section 4.3.2. The Executive and Supervisor servers are required for all simulations and, as shown in figure 7, are connected to all and most servers respectively. See appendix A for a more complete list of servers.

3.7 Sample data flows

This section describes a relatively complex example of how the NARSIM C/S architecture is being used in NARSIM. Without going into the finest details, this example shows how the datalink server operates: where is gets its data from, how it communicates with others servers, etc. The partial

configuration of servers used in this example is given in figure 8.

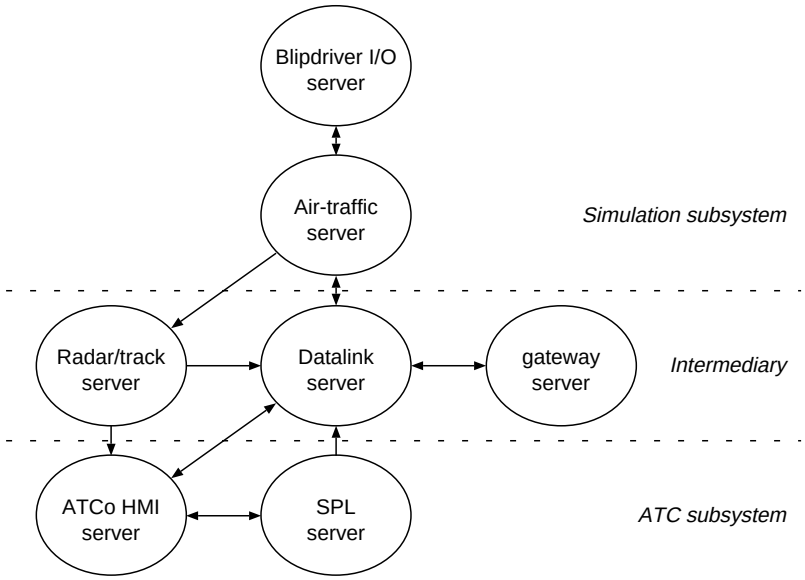

Fig. 8 Partial configuration of servers

In this example, the datalink server handles all digital information exchange between the ATCo HMI server and the pseudo-pilot HMI (in NARSIM named the *blipdriver*). The main function of the datalink server is to route the message to the correct pseudo-pilot or ATCo and to simulate realistic delays depending on the simulated network.

Immediately after start-up, the datalink server will use implicitly addressed subscriptions (see section 3.1.4) to subscribe itself to *downlinked* and *uplinked* datalink messages. The ATCo HMI server subscribes itself to *propagated downlinked messages* and the air-traffic server subscribes itself to *propagated uplinked messages*.

When an ATCo enters a command to be uplinked to a pseudo-pilot, it will use the delivery library (section A.2.2) to send the datalink message to the datalink server. Assuming, for argument's sake, that the datalink server should compute a delay as if the message were sent using SSR (Secondary Surveillance Radar) mode-S, it queries the SPL server to find out what radar track belongs to the destination aircraft. The datalink server then subscribes to track updates for that track. Because it uses the object identification of the track, it can automatically address the correct radar track server. As soon as the datalink server receives an update for that track (usually within a few seconds) it will deliver the delayed message as a *propagated uplinked message* to the air-traffic server, which has earlier subscribed itself to these kind of messages. The datalink server will also cancel its subscrip-

tion to further updates of the related track.

The interaction between the air-traffic server and the pseudo-pilot HMI server are rather straightforward. Messages downlinked from pseudo-pilot to ATCo are handled similarly to the message flow described in the previous paragraph. When the computed network delay does not depend on the observation of a track by a radar, e.g. when using satellite communications, the datalink server need not go through the find-and-subscribe-on-this-track procedure, but can simply use an internal function to compute the delay and then request a timeout at the appropriate time.

4 The NARSIM C/S middleware – technical aspects

Although the system's architecture as described in the previous chapter is of paramount importance to the success of the final system, the actual implementation of an architecture, including the ways in which the finer points are taken care of, decide whether or not the theoretical benefits of an architecture can in fact be achieved. This chapter describes the technical aspects of the NARSIM middleware from a user's point of view; the users being both the software developers and the experiment leader (supervisor) who monitors and controls the entire simulation.

It should be noted that the entire NARSIM C/S system has not been finished completely. Although the more basic functions have been operational since mid 1995, some parts are still in the implementation or test phase.

4.1 Object oriented approach

The client/server architecture by itself does not impose any kind of object oriented approach. Fortunately, almost any kind of (distributed) object oriented approach fits very well on top of a client/server based architecture.

Because all the elements and relations in an ATC simulator can be described very naturally using object oriented techniques, it has been decided to use this "object orientation" throughout NARSIM wherever possible.

4.1.1 Global object identification

Most servers could also be described as object managers. Servers can manage radar tracks, system plans, predicted conflicts, and even more dynamic objects like aircraft state vectors and timers. Within NARSIM it has been decided that all these objects should have object identifications that are unique within an entire simulation. Just as pointers can be used inside a single program to link several entities together, *object id's* can be used to achieve this goal when these entities are managed by different servers.

Typical use of these globally unique object id's can for example be found in a system plan (flightplan) which may contain references to aerodromes, waypoints, aircraft type and radar tracks. All these references can be implemented using object id's. When a client receives such a system plan and needs to know more about, say, the departure aerodrome, it can request more information about that object by invoking the server that manages the object (probably the airspace server in this example). The identification of the server that manages an object is contained in the object id.

4.1.2 Object creation/termination

For most object types, three kinds of operations are defined: object creation, object modification and object termination. When, for example, an initial flightplan is received by the system plan (SPL) server, the SPL server will create a new object (system plan). Whenever an ATCo gives a new clearance for the related aircraft, an update request is sent to the SPL server, and the SPL is updated. Finally when the aircraft is handed over to an adjacent ATM centre, the SPL is removed from the system.

In NARSIM such dynamic object management is essentially implemented by providing two services. In the case of the SPL servers, these services are named *SubscribeNewSPLs* and *SubscribeSPLUpdates*. Through use of the former service, clients can subscribe to creation of new SPLs. Whenever the SPL server creates a new SPL, the clients that have subscribed to the former service will receive the new SPL. Based on the contents of this SPL, a client may then decide to subscribe to updates of this SPL, using the latter of the two services.

With each delivered SPL update, clients also receive an indication of the fields that have changed with respect to the previous update. Object termination is signalled to the clients by sending an SPL update where none of the fields have changed. Consequently, using only the *SubscribeNewSPLs* service, clients do not receive a delivery when an SPL is removed from the system.

Some other object types for which this mechanism is used are radar tracks and simulated aircraft.

4.1.3 Selective update propagation

When a client subscribes to modifications of a certain object it may not be interested in *all* updates of that object. The SPL server, for instance, may subscribe to radar tracks to establish a link between an SPL and a track. To do so it only needs to know when a new track is created or deleted and when the SSR mode-A value changes. It does not need updates whenever the track position is updated.

To prevent unnecessary propagation of updates to all applications, a client can request to receive updates only when specific fields change. Sending the deliveries only to those servers that have expressed interest in the fields that have been modified, is usually delegated to the subscription library which is described in section 4.6.2.

4.2 Executive server

The Executive server is the first server to start. After its initialization the other servers may be started. Just after a new server is started, it must connect to the Executive server and will have to

use the *Exec services* to join the simulation. These Exec services are also used to inform clients about other available servers and services.

4.2.1 Connecting to the Executive server

The Executive server maintains a central administration of servers and services. To access this information, a server has to connect to the Executive server. The servers will have to know the exact location (*service access point*, or SAP) of the Executive server to make their connection. The function of the Executive server can be compared to that of a *nameserver* in other distributed environments.

4.2.2 Server and service administration

After start-up a server is not automatically involved in the simulation: it has no knowledge about servers and services available. The *Exec services* are defined to maintain and provide a central administration of the servers and services available. To update this administration, servers have to register themselves and the services they provide. Clients can use the information provided through the *Exec services* to find the appropriate servers and services they require.

4.2.3 Simulator states

Simulations controlled by the Executive server are stateless, i.e. a common state for all servers does not exist. As a result, servers may start at an arbitrary simulation moment. The sequence in the execution of an arbitrary server, as sketched in the previous sections, introduces some local server states. These states are directly coupled to the service invocations made by the server. The local server states are not used by the Executive server, but can be used to keep the experiment leader informed. The local states common to all servers are listed below.

START This state is entered directly after registering the server with the Executive server. It denotes that the server does exist, although it does not yet participate actively in the simulation.

READY A server enters this state when it has logged a READY message, following its successful initialization. Under normal circumstances a supervisor would not start a simulation before all servers have entered the READY state, although this is not a strict requirement.

EXIT A server is in state EXIT following the withdrawal of the server from a simulation, which is done implicitly whenever its connection with the Executive server is lost. The EXIT state can succeed every other state, and no state can succeed the EXIT state.

4.2.4 Time management

The Executive server maintains and distributes the three central (simulation) clocks. The first clock represents the given world-time synchronous with UTC, amongst others required for time aware-

ness during a frozen simulation. The second clock represents the simulation-time, starting at zero and running with a certain simulation speed, which may differ from the speed of the world-time. Simulation-time is continuously non-decreasing, it can be frozen, implying a state with a constant simulation-time. The third clock represents the experiment-time (simulated world-time), synchronous with the simulation-time, but with an arbitrary offset.

Initially the world-time is always known, the simulation-time is set to zero and the experiment-time can be set to a given initial experiment-time. The initial simulation speed (time rate) will be 0 (zero) which means that the simulation is frozen.

4.3 Graphical supervisor HMI

Probably the most visible point of the NARSIM C/S architecture is the supervisor HMI server. Through this server the experiment leader or supervisor can configure a simulation and monitor and control a running simulation.

An important design issue for the supervisor HMI server has been to include as little information as possible in the server itself about the rest of the simulation.[1] This has resulted in a supervisor HMI that has absolutely no static knowledge of other servers, except for a run-time configuration file that lists all names, executables and options of possible servers and packages that can be used to build a configuration. All other information about a simulation is obtained via the Executive server and through *general services* provided by other servers.

Fig. 9 Supervisor HMI main screen

When a supervisor HMI is added to a simulation, possibly when the simulation is already running,

[1] Perhaps this is one of the few occasions where George Orwell's *ignorance is strength* truly holds.

it subscribes with the Executive server to the list of active servers and the general services they provide. It can then ask each server for information, such as logging data.

Note that the fact that the supervisor can control the simulation does not necessarily imply that it also is the sole ruler of a simulation: other supervisors—even external supervisors not being a part of NARSIM—can monitor and control the simulation just as well.

4.3.1 Simulation monitoring and control

The main function of the supervisor HMI is, as its name implies, to enable supervision. The main screen[2] as shown in figure 9 presents the most important information about a simulation. The list of *Servers* shows which servers are, or have been, active during this simulation. The list of *Objects* shows which types of objects are currently being managed by one or more servers in this simulation. The *Control* panel shows the current simulation-time, experiment-time and the simulation speed. Buttons are available to stop either a single server or the entire simulation, freeze the simulation, start the simulation and to reduce or increase the simulation speed. The contents of the *Logging* frame is described in section 4.3.3.

4.3.2 Simulation configuration

When the supervisor HMI server is started it enters either one of two modes: it either joins a pre-configured (possibly already running) simulation, or it enters the configuration mode in which the experiment leader will be able to select and configure a set of servers for a simulation.

A typical configuration screen is shown in figure 10. Here the list of *Available Servers* shows what servers and packages are known to the supervisor HMI. The *Configuration* list shows what servers and packages have been selected for a certain simulation. The *Server Configuration* shows the current settings for the server under configuration; in this case the SRT server.

The *Configuration* list may show a single server more than once, as is the case for *ATCodis* (the ATCo HMI server) in figure 10. This means that two instances of the same server will be running in this simulation. The most practical case for this is when several ATCo HMI servers are required to control e.g. separate sectors. Each instance of a server can of course be configured independently.

Initially the options shown in the *Server Configuration* reflect the default option settings as specified in the supervisor configuration file. For the example shown, the configuration file contains the lines shown in figure 11. In fact this is the only information the supervisor HMI has about the SRT server.

[2] This is a sample screen used to demonstrate the capabilities of the supervisor HMI. It does not necessarily represent a realistic scenario.

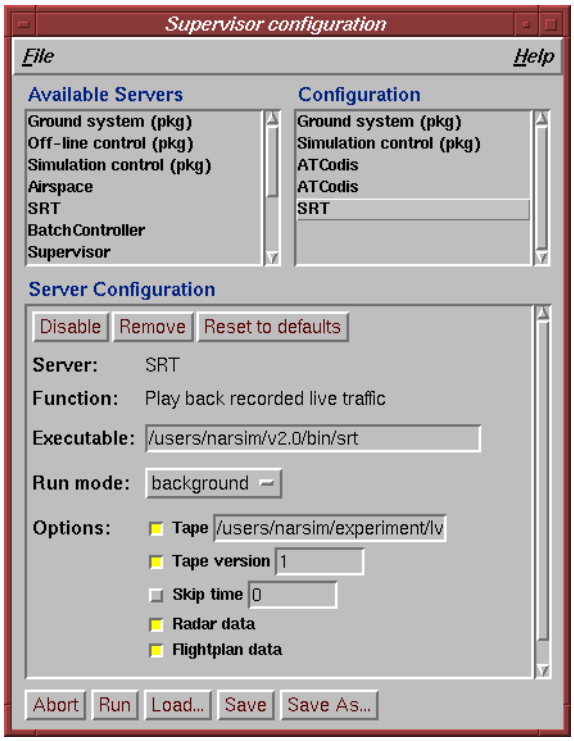

Fig. 10 Typical configuration screen where the SRT server is being configured

```
SRT { "Play back recorded live traffic"
  /users/narsim/v2.0/bin/srt background {
    {2 str  tape "Tape" /users/narsim/experiment/lvbtape...}
    {2 int  v    "Tape version" 1 }
    {0 int  t    "Skip time" 0 }
    {1 bool r    "Radar data" }
    {1 bool f    "Flightplan data" }
  }
}
```

Fig. 11 Partial supervisor configuration file

A complete configuration—selected servers and their options—can be saved to file for later use.

Even when the supervisor HMI has been used to configure a set of servers for a certain simulation this does not mean that no more other servers can be attached during the simulation.

4.3.3 Logging

The mechanism behind the logging capabilities is described in some detail in section 4.6.3. The important issue for the supervisor HMI is that all the servers in a simulation can send whatever logging data they want to the supervisor HMI server so that it can be displayed to a human supervisor. The *Logging* frame as shown in figure 9 shows a partial log of messages.

Each logged message has a message type. A few commonly used message types are *Debug*, *Info*, *Warning* and *Error*, though more types exist. The messages shown are all colour-coded so that the more important messages stand out between, for example, debugging information. The fields of the log messages shown in the example are the simulation-time, server and text field. Other fields, such as world-time, place in code and message type, can also be selected run-time.

When a single server is selected in the *Servers* list, the shown log messages will be restricted to those originating from the selected server. This makes it very easy to examine the output of a single server without having to wade through too many messages.

Note that in the example the callsign of the BAW440 is underlined in the logging message. This is part of the distributed data monitoring capability described in the next section.

4.3.4 Distributed data monitoring

Perhaps the most novel concept of the NARSIM supervisor HMI server is the way this server can assist both the experiment leader and software developers monitoring data in a distributed environment. As noted in section 4.1.1 most servers can be seen as object managers. This means that in a realistic simulation with a dozen servers there also are a dozen individual programs that each manage part of the total data.

To enable the supervisor HMI to present these fragments of data a hypertext based object browser is part of the supervisor HMI. Using one of the *general services*, clients can—and in fact *should*—provide a human-readable form of their internally managed data. The supervisor HMI can display this human-readable form as shown in figure 12. This figure shows three windows displaying a system plan, radar track and radar information. References to objects (see section 4.1.1) are shown as hyperlinks; by simply pointing the mouse over such a hyperlink and by pressing a mouse button does the supervisor HMI request the human-readable form of the data ‘under the link’ and displays this data in another window.

The supervisor HMI can send a request to a server to get information about a specific item, e.g. the system plan with callsign BAW440.[3] From here on a simple click on `LAR track` will bring up the second window shown in figure 12. Yet another click on `LAR` will show the third window with the radar specification.

Because the displayed data, as most other data, can be subject to change, the supervisor can sub-

[3] This is not shown in any figure, but implemented using a simple fill-out form similar to those found in the configuration screen. Readers familiar with the world-wide-web may know this as *Mosaic 2.0 forms*.

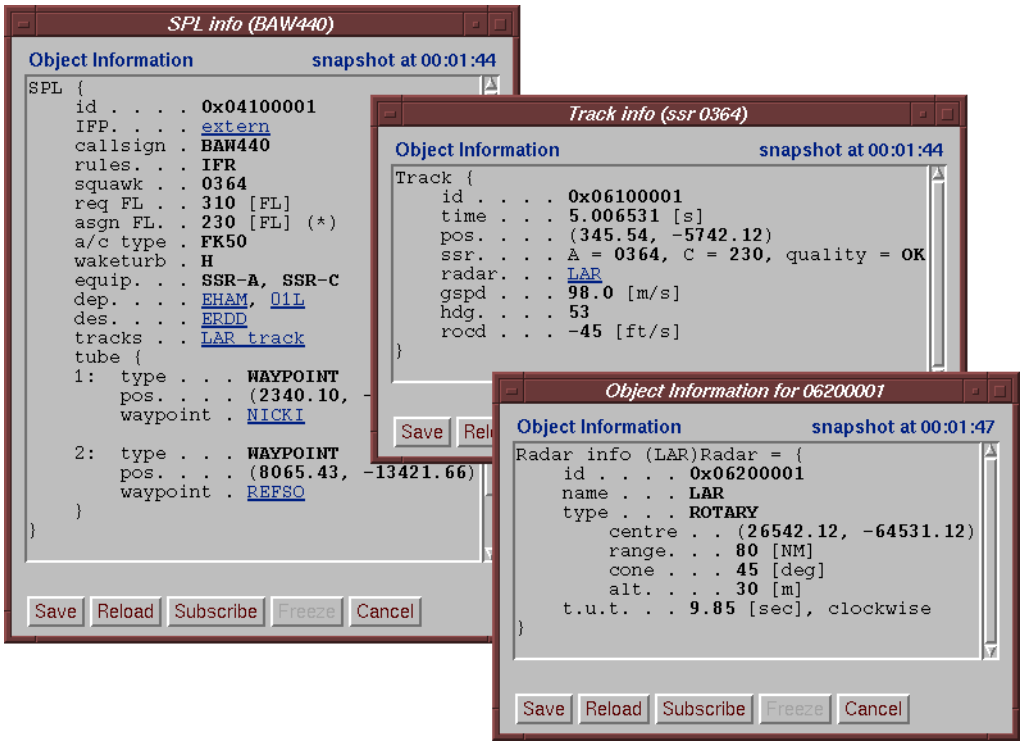

Fig. 12 Example of distributed data monitoring

scribe to updates of this displayed data. This way the integrity and consistency of the displayed data can be guaranteed.

The idea behind this distributed data monitoring is, of course, not really new. The world-wide-web (Ref. 1) has used this technique now for only a few years and has already proven itself extremely useful. To stay within the “proven concept” ideology, the human-readable form of the data which is provided by clients is formatted using the hypertext markup language HTML (Ref. 2).

4.4 Robustness

Even though the NARSIM C/S architecture is not intended for operational ATC systems, there is a strong demand for robustness in the form of graceful degradation. Because of the experimental nature of research simulators, the robustness of each of the applications cannot always be guaranteed. Instead, robustness has been built into the architecture.

The most obvious aspect is probably the absence of a global simulator state. It is not the case that all servers must first go through, say, a start-up phase, then wait until everybody is ready, then enter the next phase, etc. Each server, of course, may have its own phases, but these are not coordinated

with the phases of the other servers. A server can enter the simulation at any given time, find out what services are provided by the already present servers, based on this decide what services it can offer itself, register these services, and the server is running.

The asynchronous way in which server may be started is also found in the way servers may terminate, or leave a simulation. One way for a server to exit is on request of the Executive server (using an asynchronous general service). On receipt of such a request, the server process can simply exit[4] and the server is gone. When a server unexpectedly crashes, the Executive server is automatically notified because of the broken TCP/IP network link. The procedure, however, is exactly the same as for a normal server abort.

Practical use of these features can be found in the ability to add and remove HMI servers during a simulation. The graceful degradation implies that a single server crash does not necessarily halt the entire simulation—an especially attractive feature in complex distributed simulations.

4.5 Small Is Beautiful

One point which cannot be brought up often enough is that to keep a system easy to understand and easy to use it *must* be kept as small as reasonably possible. There certainly are some priorities as to what parts of the system must be kept small (a small interface is more important than a small module behind the interface), but “smallness” in itself is almost always a good thing.

In implementing the NARSIM C/S architecture, rigorous attempts have been made to prevent complexity wherever possible. When absolutely necessary, complex parts of the system have been isolated in black boxes (libraries) with small and simple interfaces to restrict the scope of this complexity.

Often-recurring code has also been placed into libraries, to increase uniformity and to reduce the overall complexity (complexity also increases with size).

4.6 Powerful libraries

As written in the previous section, more complex or often recurring parts of the system have been concentrated into libraries to reduce the system’s overall complexity. This section describes these libraries in some detail: the basic functions they perform and how they are used. See section A.2 for more details.

[4] Actually it is a bit more complex, as several servers may be contained in a single UNIX process. A call like *exit(0);* would also terminate the other servers in the same UNIX process.

4.6.1 Object management

Although it is desirable to have globally unique object id's in interfaces between applications, internally applications often prefer to use simple pointers or indices into arrays. The object manager provides the application with a few simple functions to establish a mapping between local references and the globally unique object id's. Furthermore when creating a new object, it is enforced that a reference to the server creating the object is included in the object id, so that other applications know which server to contact for more information. As an aid to the managing server, it is possible to label an object with a type field.

The object manager is used both by servers that manage objects and by clients that use these objects. All but a few servers in NARSIM make extensive use of the object manager.

4.6.2 Subscription management

To reduce communication overhead, all servers that manage dynamic data should provide this data to clients through subscription services, possibly next to regular synchronous services. Because almost all servers manage dynamic data, subscription management is a good thing to put in a library (see section 4.5).

The only essential trick performed by the subscription library is the administration of what clients have subscribed to what objects and under what conditions. With this information, the library can send deliveries to earlier subscription requests without any help from the managing server.

In case more complex behaviour is required for sending deliveries (e.g. when different clients want position updates using different coordinate systems) the subscription library still provides support for sending deliveries, although in this case it would require additional help from the managing server to "prepare" the data to fit the request.

4.6.3 Logging

For simple applications in a non-distributed environment, logging is not really an issue. Informative messages, warnings or errors can simply be written to the screen or to a log-file. In a distributed environment, however, such a solution no longer suffices: several applications may generate output near-simultaneously making it hard to interpret, and when a single experiment leader or supervisor needs to monitor the output of more than a dozen applications running on different computers it is highly undesirable to inspect the same number of independent log-files.

Within NARSIM this problem has been solved by defining a service set which contains *general*

services. These services are usually provided by all servers, although not all servers need to provide all general services. One of these general services is used to provide a centralized logging facility.

An important and perhaps at first a bit counter-intuitive design issue is that it is the servers themselves that provide the logging services, and not the server that actually logs the data. The supervisor HMI server, for example, subscribes to all servers that provide logging services using an implicitly addressed subscription services as described in section 3.1.4. The advantage of this role-reversal is that neither the applications that provide the data to be logged, nor the applications that can log the data need to know what their potential peers are: it is possible that there are several supervisor HMI servers that present log data to different human supervisors.

4.6.4 Option handling

Although in itself not a very complex issue, a separate library is available to handle different types of configuration options for each application. The main reason to put these functions in a library is to enforce all applications to use the same option mechanism and policy, thus providing a single consistent interface to the outside world—either an application or a human being who configures the applications within a simulation.

Because of this single option mechanism it is now straightforward to have an application, the supervisor HMI in the NARSIM case, select all the options for each server.

Another reason for providing an option library is that for performance reasons several servers can be combined in a single UNIX process (called a package). The option library makes sure that each server within a package can still parse its command-line arguments without distinguishing between a stand-alone server or a packaged server.

4.6.5 Hyperlinked data monitoring

To enable all servers to present their data in human-readable form to e.g. a supervisor as described in section 4.3.4, a small set of functions has been made available. The most cumbersome part, formatting the data into HTML, is taken care of by a library which is generated by the same program that generates the network encoding and decoding functions. This program also generates functions that convert all data types part of a service set definition—all data types being used in interface definitions—to HTML.

Using the functions provided by the library, a server can delegate almost all additional work required to implement the distributed monitoring services to this library. Only a few lines of code per server remain to inform the library of a server's (internal) datastructures.

4.7 Connecting to other simulators

When designing the NARSIM C/S system, it was already known that it had to be possible to interconnect NARSIM with other ATC simulators. Connections with flight simulators such as the NLR Research Flight Simulator (RFS) had already been realised in the past and should now be dealt with similar to connections with other ATC simulators.

Because it seemed unlikely that a single international standard for interconnecting different types of simulators would soon arise it was decided that the interconnection problem would be solved using gateways servers. These servers would convert NARSIM services to some external protocol and vice versa. Services to be converted include air-traffic and radar services for surveillance data, supervision commands (which means that NARSIM must be able to run “slaved” to an external simulator), datalink services and time control.

Currently only the DIS (Distributed Interactive Simulations, see Ref. 3) server is partly operational. It provides air-traffic services (computed aircraft positions) to the outside world, and makes incoming aircraft positions available as air-traffic services to e.g. the Radar/track server. The DIS server also relays simulation control commands (start, stop, freeze).

In the near future the datalink server will become operational and will then enable a link between NARSIM and, for example, a PATN. Also a server will be built to re-establish the NARSIM – RFS link.

4.8 UNIX integration

Because it is practically impossible to restrict all inter-process communication (IPC) to the client-server interactions as implied by the architecture, the possibility has been left open to integrate several other IPC mechanisms as offered by the UNIX operating system. These other mechanisms include communications over TCP sockets, UNIX pipes, UNIX devices, etc. and UNIX signals. An application can subscribe to any of these events via pseudo services provided by the communication system (the NCS pseudo client stub in figure 6). The same callback mechanism as for normal services is used to report events to the application.

The possibility to use IPC mechanisms other than normal client-server interactions is only intended for applications that are gateways to other simulators and applications that need to access devices (keyboard, X-terminal) to interact with the outside world. Abusing other IPC mechanisms to implement covert channels between specific applications removes most, if not all, of the advantages of the client/server architecture.

5 Organisational issues related to the NARSIM C/S system

Aside from the technical aspects of the NARSIM C/S architecture, there also are some organisational advantages. Although not directly adding extra capabilities, these advantages make the system as a whole more easy to use, manage, maintain and extend than a non-C/S architecture.

5.1 On-line documentation

All documentation written within the NARSIM C/S project has been written in HTML and is available on-line using the world-wide-web.[1] Documentation written before the availability of the world-wide-web is being converted to HTML as well, and new documentation related to NARSIM is also being written in HTML.

The current set of on-line documentation includes:

- Service-set descriptions for all available service-sets
- Server descriptions (purpose, provided and required services, options, etc.) for all servers
- Library descriptions for all libraries
- Global and detailed design documents for modules (re-)designed within the NARSIM C/S project. These include the communication system, and several servers, such as the Executive, SPL server, IFP server, etc.
- NARSIM related NLR reports such as the NARSIM C/S predesign document (Ref. 7), design and implementation standards, and the overall NARSIM C/S global and detailed design, software users manual, software test descriptions and software test plan
- Related documents not directly part of NARSIM, such as the XDR specification (Ref. 15), the standard data tool PLAID (Ref. 8), etc.

Especially the software-related documents make extensive use of hyperlinks to show relations between service-sets, servers, etc. Use of *server-side includes* makes it possible to include (portions of) actual code in the documentation without duplications. Document version management has been implemented through hyperlinks to older versions in the document change log. Ref. 9 describes the experience of using the world-wide-web in projects like NARSIM.

[1] Unfortunately due to security reasons the documentation is not accessible outside the `nlr.nl` domain.

6 Conclusions

Due to the used client/server architecture it has been possible to subdivide the entire NARSIM simulator into 20 or so relatively independent subsystems. The complete set of formal interface specifications describing all the service sets and thus all the interfaces between the servers is less than 30 pages of text. A typical server uses only a small fraction of the available services, typically four or five out of the currently 17 service-sets, making the actual interfaces very thin and thus satisfying the modularity requirement.

As each single server contains at most a few thousand lines of code, it is simple enough so that a single person can fully understand any single server.[1] In combination with the thin interfaces this makes the system truly modular and thus easy to maintain and extend.

Before the introduction of the client/server based system, most NARSIM applications could also communicate with each other, albeit without a general philosophy. Rewriting the old ad-hoc communications code using the new client/server architecture increased the capabilities and flexibility, while at the same time reducing the total amount of code by more than 10,000 lines.

Having restructured most NARSIM modules as servers has proven the validity of the symmetric client/server concept. Not only is this symmetry a clean concept, practice shows that its use results in straightforward, easy to understand code.

Within the NARSIM C/S system all inter-process communication is achieved by means of client/server interactions. Also communications with external entities (keyboard, windows system, other simulators) has been encapsulated by the middleware.

Finally it has been possible to restrict the alleged overhead introduced by a client/server architecture (or by any other abstract interface) to an acceptable level. Running real-time simulations, the overhead is less than a few percent of the available CPU and network resources. Realistic simulations involving more than 100 aircraft and several controller assistance tools can still run many times faster than real-time.

[1] The only exception being perhaps the ATCo HMI server which is really too complex for its own good. This is partly caused by the external requirement that the capabilities of the ATCo HMI server must be almost identical to the display software built for the operational AAA (Amsterdam Advanced ATC) system.

7 References

1. Tim Berners-Lee et al.: *The World-Wide Web*, Communications of the ACM, August 1994/Vol. 38, No. 8
2. Tim Berners-Lee, *Hypertext Markup Language (HTML): A Representation of Textual Information and MetaInformation for Retrieval and Interchange*, Internet Engineering Task Force, June 1993.
3. IEEE Standard for Information Technology, *Protocols for Distributed Interactive Simulation Applications*, IEEE Computer Society, IEEE Std 1278-1993
4. K.A. Borggreve, *Point-mass aircraft model for the NLR ATC Research Simulator (NARSIM)*, VG-90-014, National Aerospace Laboratory, Amsterdam, The Netherlands, June 1990.
5. W. den Braven, *Plotgenerator Design for the NLR ATC Research Simulator (NARSIM)*, VG-91-001, National Aerospace Laboratory, Amsterdam, The Netherlands, September, 1998.
6. M.A.C.C. van den Meijdenberg, *The Trajectory Predictor for ground system applications with NARSIM*, VL-94-017, National Aerospace Laboratory, Amsterdam, The Netherlands, December 1994.
7. R.W.F.J. Michiels, et. al., *A new software architecture for the NLR ATC research simulator NARSIM*, TR 95075, National Aerospace Laboratory, Amsterdam, The Netherlands, January 1995.
8. R.W.F.J. Michiels, *PLAID: A Programming Language Independent Data Tool*, TR 95159, National Aerospace Laboratory, Amsterdam, The Netherlands, March 1995.
9. R.W.F.J. Michiels, *Use of the World-Wide-Web in NLR projects, two case studies: Annette and NARSIM C/S*, IR-95-019, National Aerospace Laboratory, Amsterdam, The Netherlands, July 1995.
10. NARSIM Team, *The NARSIM Home Page*, http://www.nlr.nl/NARSIM.html
11. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, OMG Document Number 91.12.1, Revision 1.1.
12. PHARE, *Common Modular Simulator: Architecture And Application Programming Interfaces Of PARADISE, V2R0*, August 1993.
13. H.A. Reijers et. al., *Annette: A Feasibility Study for an Air Traffic Management Research Network—Work Accomplished*, National Aerospace Laboratory, Amsterdam, The Netherlands, January 1996.
14. Sun Microsystems, *RPC: Remote Procedure Call Protocol Specification*, RFC1050, April 1988.
15. Sun Microsystems, *XDR: External Data Representation Standard*, RFC1014, June 1987.

16. A.S. Tanenbaum, M.F. Kaashoek, and H.E. Bal, *Parallel Programming Using Shared Objects and Broadcasting*, IEEE Computer vol. 25, pp. 10-19, Aug. 1992.
17. *The Amoeba Distributed Operating System*, http://www.am.cs.vu.nl/

Appendices

A Survey of NARSIM C/S servers and libraries

This appendix gives a survey of most of the available servers and libraries that make up the NARSIM C/S system. Where useful some technical details are given such as programming language, code size and available documentation. As written in section 4.5, rigorous attempts have been made to prevent unnecessary complexity. This also includes trying to keep the code as small as possible, without sacrificing clarity: the reader may choose to use the *code size* to determine complexity and the *purpose* to determine capabilities, but should never do this the other way around! Compared to the number of non-comments lines, the total amount of code is on average two to three times as large.

A.1 Servers

To get a better idea of the environment for which the NARSIM C/S architecture is intended, the following sections give a survey of the servers that "use" this architecture. For each server some basic information is given describing amongst others its purpose. The related servers listed for each server represent relations between servers for a typical simulation. These relations do no necessarily hold for other simulation configurations, nor do they imply any static dependencies between these servers. The fact that server *X* usually sends data to server *Y* does not even imply that server *X* and server *Y* known each other: most dataflows are a result of implicitly addressed subscriptions (see section 3.1.4) where server *X* just happens to supply data that server *Y* subscribed to.

A.1.1 Simulation management

Simulation management in NARSIM is handled by two servers: the Executive server and one or more supervisor HMI servers.

A.1.1.1 Executive server

Purpose The Executive server's task is to manage the simulation. It provides server (e.g. name, network-address) and service information (e.g. providers) to the other servers. The Executive server also controls time and its update rate (speed) during a simulation.

Code and size C, 1200 non-comment lines

Documentation On-line interface description, internal design documentation and functional description are available.

Related servers The Supervisor server may be used for human interaction in simulation control. All servers need to register themselves with the Executive server on start-up.

A.1.1.2 Supervisor HMI server

Purpose The supervisor HMI server enables an experiment leader, other human supervisors and software developers to monitor and control the progress of a simulation. The supervisor HMI can also be used to "configure a simulation" by selecting a set of servers and setting options for each selected server. See section 4.3 for more information.

Code and size Tcl/Tk, 1000 non-comment lines; C, 250 non-comment lines. (*under development*)

Documentation All the capabilities of the supervisor HMI are documented using the built-in hypertext browser. The capabilities and internal design are also described as part of the on-line NARSIM documentation.

Related servers The supervisor HMI server gets its main information about the other servers from the Executive server. Other information such as logging is obtained from all servers.

A.1.2 ATM tools

A.1.2.1 Area conflict detection (ACOD)

Purpose ACOD is a medium term planning conflict detection tool. It uses planning data and actual radar data to determine whether the minimum distance between two aircraft will infringe the separation criteria. ACOD compares eligible flights as soon as they are identified and expected to cross the Dutch FIR. Each flight consists of waypoints at which ETO's (estimated time over) are given. These waypoints form segments which ACOD uses to perform its calculations.

ACOD has been evaluated for application in the new Amsterdam Advanced ATC-system (AAA) and will be operational in 1997.

Code and size Fortran, 4500 non-comment lines.

Documentation In addition to the services and server descriptions, complete specifications using SA/RT exist. These also are the specifications used for the operational AAA system.

Related servers The ACOD server requires an SPL server, and usually sends out conflict information to the ATCo HMI server. The SPLs must contain accurate trajectories, usually coming from either the TP server or the SRT server.

A.1.2.2 Short term conflict alerting (STCA)

Purpose STCA uses data from a radar server to calculate whether aircraft pairs will infringe a given separation criteria in the near future. STCA compares a flight which is under control of ACC (area control centre—Amsterdam's en-route air-traffic control) each time its track is updated with all other flights in order to detect conflicts between pairs of aircraft.

STCA has been evaluated for application in the new Amsterdam Advanced ATC-system (AAA) and will be operational in 1997.

Code and size Fortran, 3200 non-comment lines.

Documentation In addition to the services and server descriptions, complete specifications using SA/RT exist. These also are the specifications used for the operational AAA system.

Related servers STCA works mostly on radar tracks. Some information from the SPL server is required to use the executive flight level (EFL) and to determine which ATCo has a flight under control.

A.1.2.3 Flight path monitoring (FPM)

Purpose The purpose of FPM is two-fold. Firstly it should reduce the workload of air traffic controllers by removing the burden of continuously having to check each flight for compliance with its planned route. Secondly the introduction of automated tools like ACOD requires that the flightplan data is up-to-date and accurately represents the actual situation to function properly.

Code and size Fortran, 500 non-comment lines

Documentation In addition to the services and server descriptions, complete specifications using SA/RT exist. These specifications, written by the Dutch civil aviation authority, are also used for the operational AAA system.

Related servers FPM requires an SPL server and radar tracks to compare the actual and the planned route. Deviations are usually sent to ATCo HMI servers and to ACOD.

A.1.2.4 Trajectory predictor (TP)

Purpose Based on an aircraft's flightplan, flight progress, current position and meteo data, TP computes the expected 4D-trajectory. This trajectory can be fed back into the SPL to be used by other tools, or can be used as input for a flight-path conformance monitor.

Code and size Fortran, 2500 non-comment lines.

Documentation In addition to an on-line functional description, TP has been fully documented in Ref. 6.

Related servers TP gets its main information from the SPL server and uses the last observed radar track to compute an up-to-date trajectory. The meteo server is used to account for meteorological influences such as wind direction and speed. The SPL server has an option which enables auto-feedback of computed trajectories into SPLs.

A.1.2.5 Centre TRACON automation system (CTAS) gateway

Purpose The CTAS server provides a connection between NARSIM C/S and the CTAS software. This servers makes NARSIM information such as flightplans, weather data, and radar data available to CTAS and provides the datalink facility to air traffic controllers. The CTAS

server will run on Sun IPX stations under SunView, as it cannot be run on X11 displays. The CTAS server which is currently in the design phase will be implemented within the (already existing) communications manager of CTAS.

Code and size *under development*

Related servers The CTAS server uses the SPL server to enter tactical clearances. Radar tracks are used for 4D approach sequencing and monitoring. The datalink server is used to send CPDLC messages to the pseudo-pilots (via the air-traffic server).

A.1.3 Databases

Most servers in NARSIM maintain some sort of database, so technically all server could be categorized as being database servers. The servers listed in the following sections, however, seem to fit the description of database servers better than others: their main function being to maintain and/or distribute data on request to other servers.

A.1.3.1 Initial flightplan server (IFP)

Purpose The task of the Initial Flightplan server is to distribute Initial Flightplans to—in particular—the SPL server and the Air Traffic server. It is possible to deliver Flightplans to these servers that are slightly different, thereby introducing slight inconsistencies that force Air Traffic Controllers to take action.

Code and size C, 1000 lines

Documentation On-line interface description, internal design documentation and functional description are available.

Related servers Initial Flightplans are used by the System Flightplan and Air Traffic servers.

A.1.3.2 System plan server (SPL)

Purpose The System Plan server maintains and distributes system plan information. Clients can be notified of newly created System Flightplans and can then decide whether they wish to be notified of changes to them. To reduce the number of deliveries, clients can specify exactly which changes they want to be notified of. The System Flightplan server also performs some actions to maintain the consistency of System Flightplans, e.g. checking whether the correct radar tracks are linked to the Flightplan, and performing flight progress monitoring.

Code and size C, 1800 lines

Performance The SPL server can service hundreds of requests per second, although in practice during real-time simulation less than ten requests per second are sent or received by the SPL server.

Documentation On-line interface description, internal design documentation and functional description are available.

Related servers The IFP server provides Initial Flightplans to the SPL server. System Flightplans are used by the ATCo display server and all ATM tools, such as ACOD and FPM. The SPL server uses radar tracks to perform consistency checks.

A.1.3.3 Airspace server

Purpose The Airspace server maintains a database of airspace objects. These include waypoints, airfields, runways etc. The Airspace server uses intelligent client stubs to drastically reduce the number of requests made to it.

Code and size C, 2200 lines

Performance Through use of intelligent client stubs the airspace server can service well over 100,000 requests per second.

Related servers Many servers including the initial flightplan server, system plan server, air-traffic server, ATCo HMI use the Airspace server in some way.

A.1.3.4 Meteo server

Purpose The Meteo server being fully compatible with the CMS meteo server provides 4D actual meteo data, and nowcast and forecast meteo data based on grid-meteo. Meteo data includes wind speed and direction and air pressure, temperature and density.

Documentation In addition to the service and server descriptions, on-line user-requirements and design documents and a software user manual are available.

Code and size (*under development*)

Performance Through use of intelligent client stubs the meteo server can service up to 100,000 requests per second.

Related servers Servers using the meteo server are the air-traffic server for actual meteo data and TP for forecast meteo data.

A.1.4 Air traffic generation

The NARSIM air-traffic server, called *SIMICA*, and its pseudo-pilot HMI have been split into two servers. This enables the possibility that several pseudo-pilot HMI servers can be used to control the aircraft simulated by a single air-traffic server. Note that pseudo-pilot HMI servers can be added and removed during a simulation, e.g. when the load on a pseudo-pilot is found to be too high.

A.1.4.1 SIMICA (air traffic server)

Purpose SIMICA contains several flight control, aircraft performance and navigational models. The requirements of the experiments determine the accuracy of the used models. A complete 4D guidance experiment, for example, requires more advanced aircraft performance and flight control models than an experiment used for the evaluation of en-route traffic.

The navigational models contain horizontal navigation (area navigation and conventional navigation), vertical navigation, interception of radials, heading, ILS, etc., standard routes, holding procedures, 3D navigation, etc. An advanced 4D guidance is being developed at the moment.

The aircraft performance model that is used depends on the purpose of the experiment. NARSIM supports both a simple parameterised aircraft model as two point mass aircraft models.

Code and size Fortran, 7500 non-comment lines. The code is planned to be redesigned and rewritten in C.

Documentation Almost a dozen reports have been written about (parts of) the NARSIM air-traffic server (see for example Refs. 5, 4). A detailed description of the interface is available on-line.

Performance SIMICA can currently handle up to 100 aircraft, resulting in a computer load of only a few percent on a modern workstation.

Related servers The pseudo-pilot HMI, datalink, meteo and airspace servers are used. Output is usually processed by the Radar/track server.

A.1.4.2 Pseudo-pilot (blipdriver) HMI server

Purpose The Blipdriver server forms the user-interface between the Air-Traffic server and the pseudo-pilots. It shows the status of the aircraft that the pseudo-pilot has under control, and it enables the pseudo-pilot to enter commands that control the behaviour of those aircraft.

Code and size Fortran, 2500 non-comment lines

Documentation On-line functional description and command description are available.

Related servers The Air Traffic server uses the Blipdriver IO services to control aircraft.

A.1.4.3 System Recording Tape (SRT) server

Purpose The SRT server is used to play back recorded live traffic from the Dutch civil aviation authority (LVB). These recordings contain radar tracks, initial flightplans, manual and automatic flightplan updates and predicted trajectories. These data are provided to the rest of the system in the exact same way as they would normally be provided by other servers. NLR currently has more than 25 live-traffic recording with an average length of more than 2 hours.

Code and size C, 1500 non-comment lines

Related servers The types of data provided by the SRT server would normally be provided by the Radar/track server, Initial Flightplan server, TP server and ATCo HMI server.

A.1.5 Miscellaneous

A.1.5.1 ATCo display server

Purpose The ATCo display server displays relevant information for the air traffic controllers. Depending on the position (function) of the air traffic controller this can be one or several from the following: airspace maps, flightplan tables, plan view of en-route or approach/departure radar tracks, track labels in various layouts, vertical view of stack traffic, CRDA 'ghost' traffic, weather information, area-conflict information, short-term-conflict information or flight path deviations. Some of this information has been added to the display program for prototyping and evaluating the LVB AAA system.

The display software can be used on an X11 display, or with a Metheus 3720 or Raytheon DCX display controller. The controller can make inputs using either a rollball or a mouse and a keyboard or one of several function keyboards. Furthermore the controller can use one or two TIDs (touch input devices) having a button/page layout which can be easily adapted to a specific task.

Documentation A description of the services and a functional overview is available on-line. Some additional documentation is available on paper.

Performance The performance of the display server depends on the specific hardware used. On an X display of 1024×1280 resolution roughly 200 tracks can be displayed while maintaining good respons times. On a 2048×2048 display the number of tracks is somewhat lower, depending on the display controller and the quantity of additional information on screen.

Related servers Information of the SPL and Radar/track servers is used to create the basic air traffic picture. Furthermore the STCA, ACOD and FPM server can be used to show conflict and warning information.

A.1.5.2 DIS (Distributed Interactive Simulation) server

Purpose The purpose of the DIS server is to interact with other (local or remote) simulation applications and exchange data in such a way that all simulators can participate in one global (real-time) exercise.

The DIS infrastructure provides interface standards, communications architectures and management structures necessary for linking all kinds of simulators into one distributed simulation exercise. Data exchange between simulators has been defined by DIS using Protocol Data Units (PDU's). Although DIS initially has not been designed for linking ATC simu-

lators, it is a well defined standard that can be adapted for inter-site ATC simulation. Extra capabilities can be added in the form of free definable Data PDU's (e.g. Flightplan updates) The DIS server is still in its prototyping phase where network requirements, simulation management and handover problems will be investigated.

Code and size C, 2200 non-comment lines (partial implementation)

Documentation The official protocol specification (Ref. 3) and on-line server description and design documentation is available.

Related servers The DIS server provides almost the same capabilities as the air traffic server, because it is essentially a remote traffic server.

A.1.5.3 Radar/track server

Purpose The Radar/track server 'converts' positions from simulated aircraft, obtained from e.g. an air-traffic server into observed tracks. When several servers provide air-traffic services, all observed aircraft will be combined into a single stream of tracks. This server can currently simulated two types of radars. Simulating the NUL-radar, the Radar/track server immediately converts any incoming aircraft positions to a tracks which is then propagated to clients having subscribe to radar tracks. When simulating a rotary radar, a delay is computed based on the position of the aircraft, the position of the radar beam and their respective movements. Only when the simulated rotary radar could in real life have observed the aircraft (i.e. when crossing its beam) is the (extrapolated) position converted to a radar track and delivered to clients.

Code and size C, 800 non-comment lines.

Documentation On-line internal design documentation and functional description are available.

Performance The Radar/track server is computationally very efficient. Hundreds of aircraft can be handled without any noticeable load.

Related servers Input from air-traffic servers (actually: all servers providing air-traffic services) is used as input. Radar tracks are delivered to many clients such as the SPL server, ATM tools, ATCo HMI server, datalink server, etc. In a distributed simulation the DIS server will also provide air-traffic for traffic generated outside NARSIM.

A.1.5.4 RADNET server

Purpose The RADNET server reads and decodes ASTERIX formatted RADNET data to be used as radar data in simulations. The external interface of this server is the same as those of other radar data providers.

Code and size Fortran, 2800 non-comment lines.

Related servers RADNET data can be used by all servers that require radar services.

A.1.5.5 Datalink server

Purpose The Datalink server simulates a simple ATN with applications for controller-pilot interactions (CPDLC) and to downlink aircraft parameters (DAP). It can also be used as an transparent interface to other research or operational ATNs.

Code and size (*under development*)

Related servers Servers providing radar tracks may be used to calculate a SSR mode-S delay. Typical datalink users are the air-traffic and ATCo HMI servers.

A.1.5.6 RT (radio-telephony) server

Purpose The RT server which is planned for late 1996 will handle all voice communications between ATCo HMI servers and pseudo-pilot HMI servers. Configuration will be done dynamically through provided services.

Code and size (*under development*)

Related servers The RT server will be used by the ATCo HMI servers and pseudo-pilot HMI servers. The ATCo HMI can draw DF (direction finder) lines to indicate which pseudo-pilot is talking on a frequency. For distributed simulations a link with the DIS server is foreseen.

A.2 Libraries

A.2.1 Object management

The purpose of the Object library is to maintain administrations of objects of a certain type, to be exchanged between applications. Each object of a certain type needs to have a unique identification to avoid mixing up two objects of this type.

Applications creating objects, called *owners* of these objects, will use an Object manager, an instance of the Object administration, for each object type it provides. When this application creates an object of a certain type, it asks its corresponding Object manager for a unique object identifier (object id). This object id can be sent to interested applications along with the actual object.

Applications using objects not owned by them, can use an Object manager for each object type used by them. They use these Object managers to maintain an administration of mappings of (global) object id's to local references (an index or pointer into the administration of the application). Receiving an update of a global object, the Object manager can retrieve the local object reference of the object, which can be used by the application to update its local database.

An object id is uniquely identified by the following three items:

- Identifier of the owner, 'responsible' for the object. This may be required to obtain more detailed information from an object (to get more information about an object, one has to ask

the owner that provided the object in the first place). It is also useful for removal of all objects of a terminated owner.

- The type of the object. Types are unique per owner and are typically used to distinguish between the objects of equal owner, but different type.
- The object sequence number. These numbers are unique per owner/type tuple, start at 1 and are incremented whenever a new object is created by an owner. Sequence numbers are never reused within a simulation.

Within the NARSIM C/S system, the owner identifier will be the server id of the server providing the objects.

A.2.2 Subscription management

Subscription management is implemented in the Delivery library. The purpose of this library is to handle distribution of subscription deliveries at the server's side.

A server may use the Delivery library to store subscription requests of clients. Whenever an event gives data to be delivered, the Delivery library can deliver the data to the subscribers. Simple subscription services require the server to install a Delivery manager, an instance of the Delivery administration. Subscription services allowing the client to subscribe on separate objects, will need a Delivery manager per object, e.g. one for each system plan.

The subscription administration of the Delivery manager is based on the subscription id parameter of all the subscription services. Also, the Delivery manager maintains event flags for each subscription made. The event flags are a specification of events defined by the client as "interesting". The Delivery manager can use these flags to select a subset of clients interested in the event causing the delivery.

A.2.3 Logging

Probably the smallest library is the one that enables all applications to log their output data without having to worry about subscriptions etc. Already in the formal server specification the programmer can direct all incoming requests for logging data (see section 4.6.3 for a description of the logging mechanism) to this library. The only interface directly used by the applications is a single function[1] that sends data to be logged to those servers that have subscribed to this server's logging data.

[1] In C it is actually a macro, which makes it possible also to report the exact place in the code from where the log message is sent

A.2.4 Option handling

The purpose of the Option library is to parse (command line) options and settings for stand-alone servers and servers in packages alike. It is also possible to process options obtained externally (e.g. from the Executive server) as if they were command line options.

The Option library provides a set of functions to access several types of command line options: booleans, integers, strings and sequences of strings. It is possible to set any of these options either directly (by hand) or indirectly through use of the supervisor configuration screens as described in section 4.3.2.

B Used industry standards

Wherever possible, industry standards have been used in the NARSIM C/S system. Sometimes standards have been carefully studied, only to find that they did not suit the needs of a real-time ATC simulator. Even from those standards, however, the useful ideas have been reused in custom built software. The list of industry standards found in the current NARSIM C/S system is:

UNIX All NARSIM C/S software runs under the UNIX operating system. Not too many servers, however, rely on this, as low-level communication and file access is most often done using libraries that hide the UNIX interface from the applications.

TCP/IP The low-level communication layer uses the Berkeley socket interface to the TCP/IP layer. All NARSIM C/S inter-process communications is done using TCP/IP.

ANSI-C/C++ Large parts of NARSIM are written in ANSI/C. The entire communications layer and most of the client/server specific software is written in C, although interface to other languages are also available. No special C++ class libraries have been written yet, but the object-oriented design of the current libraries allows easy transformation when more C++ code will be used in NARSIM.

Fortran77 Historically much software has been written in Fortran77. Because it is often not efficient to translate large portions of often complex code to another language (compared to maintaining interfaces to different languages) Fortran77 remains to be used for computationally intensive servers such as ACOD and STCA.

Ada Because much international code is written in Ada, it is essential that it is possible to accommodate Ada code in NARSIM. Lack of standardisation for Ada regarding its interface to other programming languages and UNIX facilities has kept Ada from being used in the NARSIM core system.

Tcl/Tk For building user interfaces in a complex real-time environment Tcl/Tk (pronounced as *tickle tee-kay*) is almost ideally suited. The entire supervisor server (except for some small stubs) has been written in Tcl/Tk. The possibility to implement an HTML based hypertext browser in a few hundred lines of code shows the power of the Tcl language and the Tk library.

It is currently being investigated if the pseudo-pilot HMI should be rewritten in Tcl/Tk also.

X11R5 All HMI's—ATCo, pseudo-pilot, and supervisor—use the X Window System to display graphical information. It is not expected to shift to any other system in the near future.

Motif Currently the Motif look-and-feel is only used in the supervisor server. Use of Motif in a more operational environment is a point of very strong debate: the "motives" for using Motif that apply to HMI's in general do not always apply in time-critical environments such as air-traffic control.

XDR All data sent through a network from client to server is XDR-encoded. Functions to encode and decode local data structures into XDR are automatically generated by NARSIM C/S.

HTML When converting local data structures to a human-readable form, mostly for debugging and supervision purposes, the hypertext markup language HTML is used.