

# HYDRA: High-Speed Simulation Architecture for Precision Spacecraft Formation Simulation

Bryan J. Martin\* and Garrett A. Sohl†  
*Jet Propulsion Laboratory, Pasadena, CA*

HYDRA— the Hierarchical Distributed Reconfigurable Architecture — is a scalable simulation architecture that provides flexibility and ease-of-use which take advantage of modern computation and communication hardware. It also provides the ability to implement distributed- or workstation-based simulations and high-fidelity real-time simulation from a common core. Originally designed to serve as a research platform for examining fundamental challenges in formation flying simulation for future space missions, it is also finding use in other missions and applications, all of which can take advantage of the underlying Object-Oriented structure to easily produce distributed simulations.

HYDRA automates the process of connecting disparate simulation components (HYDRA Clients) through a client-server architecture that uses high-level descriptions of data associated with each client to find and forge desirable connections (HYDRA Services) at run time. Services communicate through the use of Connectors, which abstract messaging to provide single-interface access to any desired communication protocol, such as from shared-memory message passing or TCP/IP to ACE<sup>1</sup> and CORBA.<sup>2</sup> HYDRA shares many features with the HLA,<sup>3</sup> although providing more flexibility in connectivity services and behavior overriding.

## INTRODUCTION

HYDRA is a lightweight, client-server architecture that has a publish-subscribe paradigm for inter-client communications, designed to enable high-speed distributed and heterogeneous simulation (see Figure 1). It provides for complete automation of client connection and communication without programmer intervention, and also provides automation of timing services and synchronization, either when connected to a real-time OS, or at lower fidelity (and higher jitter) if running in a normal OS. The architecture is designed to be overridden as needed without breaking the paradigm.

HYDRA grew from research performed in the distributed simulation of formation flying missions. Our past experience showed that there was a need for a lightweight client-server architecture that provided automated and managed connectivity, but didn't perform a large efficiency trade-off in favor of flexibility. Using the ability of C++ to produce in-line template code we have created a distributed architecture in the

vein of HLA. We have incorporated description-based externalization similar to CORBA's, however heavy use of compile-time implementation has increase the overall efficiency of HYDRA as compared with run-time coding schemes.

Clients themselves are fully automated, with most behaviors overridable or replaceable as needed. All client behaviors can be performed by calling a single "busy" function that is provided by the base class, resulting in callbacks to derived classes as needed. Secure communication is enabled by point-to-point connections between clients to reduce communication collisions, and data is wrapped in a binary, hierarchical packet format that permits efficient run-time data validation as well as an increased ability to catch routing and similar errors.

HYDRA is still early in its life. Although the architecture design has been static for some time, new functionality continues to be added. Even so, it has been demonstrated as a useful tool in the research environment for performing heterogeneous, distributed and single-hosted simulation, and will soon be in use in a variety of flight testbeds for space missions and technology development.

HYDRA is designed to support fully asynchronous initialization. One common problem in distributed, multi-component simulations is that the order in which components start is critical, as many applications are not robust if the starting or initialization order is violated. (This is a recurring problem: networking issues,

\*Senior Staff Engineer, Engineering and Science Directorate, Avionic Systems and Technology Division, Autonomy and Control Section, Simulation and Verification Group.

†Staff Engineer, Engineering and Science Directorate, Avionic Systems and Technology Division, Autonomy and Control Section, Simulation and Verification Group.

Copyright © 2003 by the American Institute of Aeronautics and Astronautics, Inc. The U.S. Government has a royalty-free license to exercise all rights under the copyright claimed herein for Governmental Purposes. All other rights are reserved by the copyright owner.

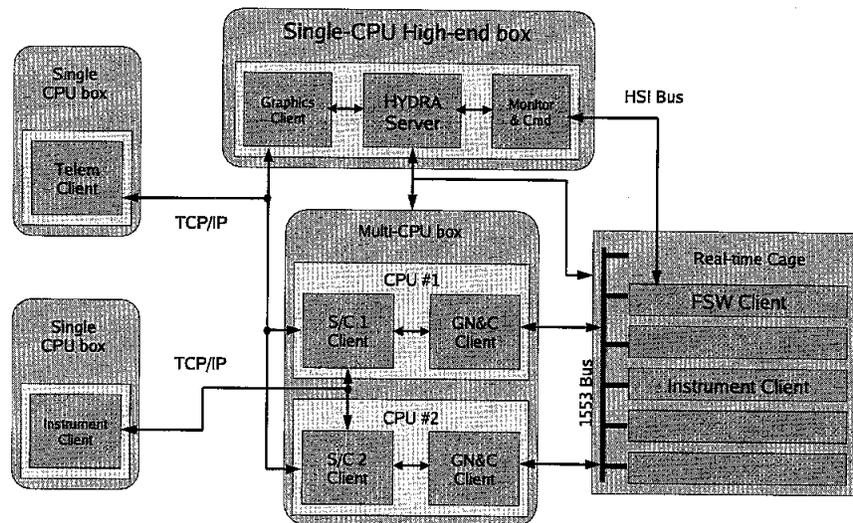


Fig. 1 HYDRA Example Block Diagram

server speeds, and differences in OS speeds can all lead to significant, essentially random delays in application start-up times.) HYDRA's design uses two main mechanisms to avoid this pitfall. First, all clients are essentially state machines at start-up, awaiting connections and commands through various channels. Normally this means that even a single connection is sufficient to access the functionality of a client, however in some cases a client cannot produce valid data until some subset of the connections (called *Services*) are active. These key *Services* are marked at instantiation with the *Critical* flag, indicating that the client will not begin full functionality until all of these connections have been completed successfully.

The following sections discuss the main elements of using HYDRA to enable distributed, heterogeneous simulation. The last section discusses a specific application of HYDRA to a formation-flying research problem,<sup>4</sup> and gives a brief discussion of future directions for the architecture.

### DESIGN PRINCIPLES

In approaching creating literate programming principles,<sup>5</sup> it is desirable in many ways to write very little code. That is *not* a general statement supporting the use of minimalist, write-only languages<sup>1</sup>. Rather, it means that when read, literate code should clearly express the intent of the programmer, not the intent of the code. Hundreds of lines of auto-generated code in which are buried five lines of manually generated code that actually does the work is a good example of how to make code maintenance and re-use diffi-

<sup>1</sup>Perl, for instance

cult. Using an object-oriented paradigm, that same example would result in a program which had close to five lines of implementation with what would effectively been auto-generated code hidden away in parent classes. Documentation is embedded directly into all files where pertinent, and assembled automatically using the Doxygen package.<sup>6</sup>

Although the codification of a design methodology can easily take many pages to describe, it is much better to find a few key principles that guide the development and techniques that are the basis of any architecture. Although special cases and peripheral concepts are not found within the following list of key design features, it serves better than an exhaustive document with the same intention:<sup>7</sup>

**1. Don't make the user deal with common tasks: let the user focus on what makes new elements unique**

A 'Hide, but override' philosophy leads to the creation of default behaviors that the user can replace on those rare occasions where necessary, without significant time penalty.

**2. Design for efficiency**

- Make success the fastest path
- Get the code out of the way of the process
- Don't sacrifice run-time efficiency to improve one-shot event timing, such as initialization

**3. Assume a distributed model first, then enable efficient single-processor use**

A single-use, monolithic application will always be faster than a generalized application that performs the

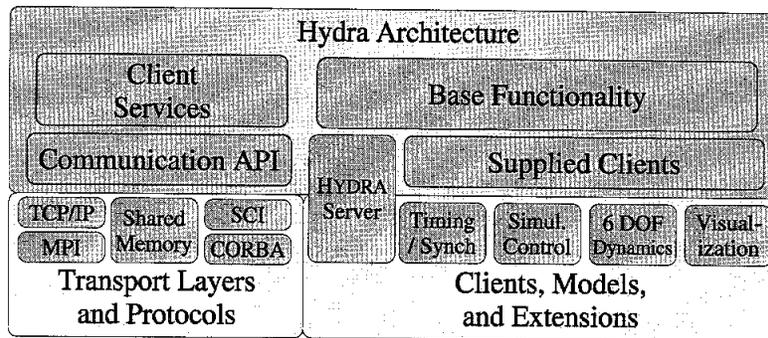


Fig. 2 HYDRA Architecture

same task. This is not a statement of failure on the part of generalized applications, but it points out that generality costs time. Designing with these two ends in mind, however, it is possible to create efficient architectures with a lower level of generality, that still satisfy all practical needs for simulation.

**4. Take advantage of standard template library (STL) routines**

The STL is well tested and efficient, and the STLs' basis in templates results in run-time efficiency. Hand-coded algorithms are theoretically more efficient, but the time spent researching, coding, debugging, and characterizing them offsets these gains in most cases.

**5. Keep components properly partitioned whenever possible, especially in the core**

Proper design partitioning saves time in the long run, and increases utility, which opposes the normal assumption of expediency.

**6. Don't use it just because it exists / Keep It Simple**

Keep the number of external package dependencies in the core areas at an absolute minimum. Avoid changing internal interfaces to facilitate external package use. Glue layers are also expensive to maintain, so it is far better to use inter-client interfacing schemes that require no extra maintenance. (This has the added benefit of making external packages network enabled.)

A layered, inherited approach makes it possible to understand the overall structure by understanding complete components, and then understanding their interactions.

In order to satisfy the requirements of low latency, high speed, and flexibility simultaneously, the architecture must support a success-first principle. That is, failures must be caught, yet doing so must not slow down the cases that succeed. One common result of this approach is that errors are caught and handled at a much higher level than might otherwise be the case,

and simply passed through at the lower levels. The C++ exception handling mechanism is superb at fulfilling this goal, as intermediate layers do not have to handle failures resulting from lower level calls.

**LAYERED DESIGN**

HYDRA provides a layered architecture using several techniques: templating, virtual methods, and message interception. These techniques enable a high degree of flexibility without a large run-time penalty, and in fact are the major method by which we satisfy 1, 2, 4 and 5 from our design guidelines. Templating and virtual methods are both required to make proper use of the C++ *Standard Template Library*, which is used extensively in HYDRA.

Templating satisfies our design principles in two ways: by reducing the amount of code required to implement new features (such as data types support, communication packetization and validation) and by reducing the errors typically experienced by creating and maintaining multiple, similar methods. Templating enables additional layering in HYDRA by allowing, for instance, the implementation of a message packetization hierarchy without regards to data type.

Heavy use of virtual methods in the base class functionality (See Figure 2) gives HYDRA a large number of default, inherited behaviors that are overridden by client applications as needed. The message processing method, for example, can be overridden at any level, with unprocessed messages passed to the next level. Only the messages unique to a given client need appear in the code for that client. This produces readable, compact code for each client application which can rely on existing code for default behaviors. Client applications can also choose to override these default behaviors and not rely on existing behaviors. This satisfies the 'Hide, but override' philosophy of the first design principle and gives each client total control over how each message is processed.

These design layers help satisfy the HYDRA design

principles as outlined in the previous section. Normally, keeping a strict design separation between low-level communication API (Section FORMING CONNECTIONS), high-level connection description (Section COMMUNICATION), and data description (Section SENDING INFORMATION) can add latency to run-time communication due to penalties caused by generalized code. With templating and careful design, HYDRA avoids these latencies while providing separable components.

In the next section, we will describe the components of the main HYDRA *Communication* layer: forming connections, sending information over connections, and event callbacks.

### COMMUNICATION

As outlined in the introduction, HYDRA employs a client-server architecture in which clients register with a server application at initialization. Connectivity between clients is described by *Services*, which are communicated to the server when an initial connection with the server is formed. Both types of applications share a common abstraction layer for inter-process message serialization and archiving (*ArchiveOperator*), transmission and receipt. They also share a pure virtual event callback class (*DBNotifier*), which calls optional user defined methods when various run-time events are triggered, and defaults to lower-level methods if the user does not define them. These commonalities allow a server application to operate just like a client application in many ways, including the ability to have multiple server applications (where one server may function as a client application for a second server). The server does have some additional responsibilities which differentiate it from a standard client application. These responsibilities include identifying, commanding and potentially tracking connections between registered client applications.

#### FORMING CONNECTIONS

The abstraction layer used by both client and server applications for message transmission and receipt was written in order to provide high-speed, low-latency communication without loss of run-time flexibility. A modular *Connector* class is used to abstract the communication transport layer (TCP/IP, Scali, MPI, shared memory, serial bus, etc.) between clients to allow a variety of transport mechanisms without requiring a-priori knowledge of which is to be used. The *Connector* class is designed to detect problems such as the termination of one of the connected client applications and relay that information via callback methods. Disconnection is handled through client notification at

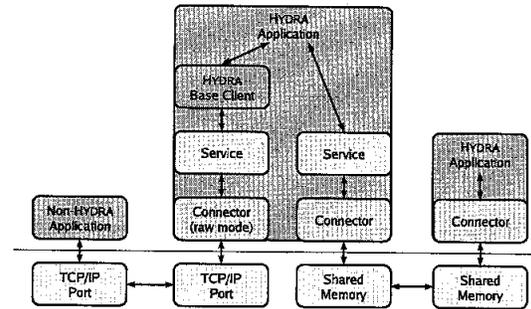


Fig. 3 Bypassing Layers of HYDRA

the base client level, and abnormal disconnection (due to the interruption of a client thread by the user, for instance) is handled cleanly by the remaining client just as normal disconnection. Due to its wide availability on distributed systems, we nominally use a *Connector* class based on TCP/IP.

The *Connector* class not only manages connections, but additionally provides automatic externalization of outgoing data (via the *ArchiveOperator* class), and caching and assembly of incoming data.

Connections between clients are configured at run time by the HYDRA server application. Each client application describes the information it is willing to provide and accept from other clients at a high level. The *Service* class provides a high level encapsulation of this information. As each new application is run in the HYDRA environment, it forms a connection with a server application, which proceeds to dynamically create a permanent connection with the client. At any time after this connection is formed, the client application provides the server with information about its *Services*. Based upon that information, the server decides which clients need to establish connections and what transport layers and protocols are available for this to occur (shared memory if the processes share a backplane, TCP/IP if nothing faster is available, etc). Once the server identifies that two clients are offering compatible services and a transport medium is available, it sends a message to each client with the details required to create a *Connector* and which *Service* is communicating over that connector. Each *Connector* is point-to-point and allows direct communication between clients. *Services* may be either point-to-point or *broadcast*, where broadcast services communicate over several *Connectors* to as many clients as required.

One major advantage of using point-to-point for all inter-client communications is that it can reduce communication bottleneck based on transport architecture. Additionally, in secure environments each client can encrypt communications based on local keys, so that messages passed between two clients are only

readable by those clients even if intercepted<sup>2</sup>.

Another advantage of this design is that applications may interact with HYDRA clients at almost any level. Figure 3 shows several applications (including applications which are not HYDRA clients) using different levels of functionality provided by the architecture. While a normal HYDRA client never deals directly with setting up *Services* or *Connectors*, it is possible to directly specify the *Connector* for a service, or do away with the service completely and hard-code a connector between two applications, still making use of the data validation and assembly inherent in the *Packet* class which will be described in the next section. In fact, if a connector is put in *raw*<sup>3</sup> mode it no longer performs even these functions, allowing connection to non-HYDRA applications. Note that in the diagram, the central application still makes use of HYDRA's capabilities to manage and monitor the connection to the non-HYDRA application.

#### SENDING INFORMATION

HYDRA provides a *Packet* class for serializing and validating data for transmission between applications over a *Connector*. In a homogeneous environment, serialization may be skipped in favor of communication using raw binary data. Since HYDRA does not know a-priori whether the environment is heterogeneous, it has an abstraction layer used to delay that decision until the connection is formed. This data abstraction uses the *ArchiveOperator* class, which provides a single virtual method in the base *Packet* class for insertion and extraction of data. The XTLL<sup>8</sup> package, a template-based layer on top of XDR, is used to serialize *Packet* data into a binary form for transmission over a heterogeneous network.

The *Packet* class also provides limited, expandable fault protection and detection schemes to insure successful delivery of information. These mechanisms are also built to allow new packet types to be derived from existing types with a minimum of effort. Data is stored in *Packets* hierarchically, with the base packet containing three pieces of information: the identity of the base packet class (later used for version checking), the total length of the incoming or outgoing packet, and the identity of the last packet in the inheritance chain (the *final ID*). (See Figure 4)

<sup>2</sup>In fact, this can be implemented in a straightforward manner if each client uses *public-key cryptography*, public domain packages are available to enable this, including GnuPG: <http://www.gnupg.org>

<sup>3</sup>In *raw* mode, packetizing, externalization, and validation are not performed, and the size of incoming data is unknown. Typically, in *raw* mode the receiving application must perform these functions based on a-priori knowledge of the external, non-HYDRA application

| Serializable Object |             |
|---------------------|-------------|
| ID (Packet)         | Packet      |
| ._length            |             |
| ._finalPacketID     |             |
| ID (SomePacket)     | SomePacket  |
| ._intData           |             |
| ._floatData         |             |
| ._doubleData        | OtherPacket |
| ID (OtherPacket)    |             |
| ._numData           |             |
| ._data[0]           |             |
| ._data[1]           |             |
| ._data[2]           |             |

Fig. 4 Packing and Unpacking of Data in HYDRA.

A number of basic *Packets* are provided, and the user can also create new packet types based on the required application. Creating a new packet type requires minimal effort since the needed serialization and fault protection methods are inherited from the base *Packet* class. While the *Packet* class provides very useful functionality, some applications may require lower level communication. HYDRA allows an application to bypass packet communication and send raw data directly over a *Connector* if required. This allows a client to communicate with an outside processes which does not understand the packet structure. However, sending raw data does not allow for using the fault protection provided by the *Packet* objects.

One major advantage HYDRA has over DIS (Distributed Interactive Simulation, IEEE Std 1278.2-1995), is that it uses point-to-point communications wherever possible, much like CORBA does. Unlike CORBA, however, HYDRA supports very low latency communication.

#### EVENT CALLBACKS

Callback methods are used by HYDRA to notify an application about a variety of events, including communication events. Each event callback has a default behavior which can be overridden at several levels to provided needed functionality, and event handling behavior can be defined on a per client, per *Service*, and per *Connector* level. Figure 5 shows how a poll command generated by a HYDRA application results in a callback message which can be interpreted at various levels of abstraction. The application code, HYDRA client code (or server code if the application is a server) and HYDRA base code are each given an opportunity to process the message callback before passing the message to the next abstraction layer. If these three levels fail to process the callback successfully,

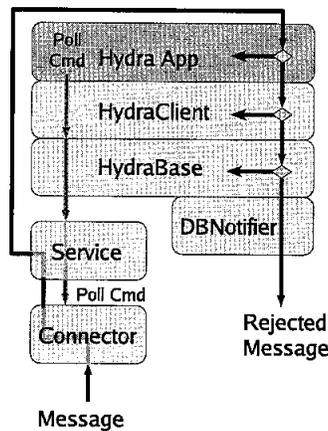


Fig. 5 Routing of Messages in HYDRA.

the DBNotifier class provides a final layer of error logging and the message is rejected.

The most common of these message callbacks is notification of the receipt of a Packet sent from another application. Each Packet header includes information about the type of information it contains. Based on the Packet type, the Connector it was received over and the Service associated with that Connector, a HYDRA application can process the message or pass the message to the parent client code. Creating a new client application can be as simple as creating code to deal with specific Packet types of interest and letting the parent handle all other Packets. Other event triggered callbacks include the creation of a new Connector, notification that a Connector has successfully established point-to-point communication with another client and notification that a connection has closed. One special Packet hierarchy and the callback methods that deal with it, the time stamped TimePacket, are discussed in section on time dependent clients.

#### TRACKING HEALTH OF CONNECTIONS

The relative health and status of all connections is tracked automatically by a reporting mechanism between the client and the server, where this repository of information is held. Status messages from the clients are sent only when status changes do reduce traffic on the communication channels that would be caused by a polling model. Each service request of each client is tracked and timestamped as it proceeds from first notification to the server of a request, through connection and eventual termination. This time history can be archived and reviewed later for performance or validation purposes. This information may be used in a variety of ways, from automatically graphing the current layout of the simulation to detecting

failed connection attempts or non-responsive clients. Currently, the data is used by a special-purpose client to automatically generate a set of web pages that give insight into the health, status, and eventually throughput of each service connection, sorted by client, service, or as an aggregate.

#### TIME-DEPENDENT CLIENTS

In a simulation environment, some of the client applications involve propagating the state of a dynamic system forward in time using a numerical integrator. These client applications include the concept of simulated time and require special consideration when communicating with other time-dependent clients. Synchronization is used to insure that messages passed between time-dependent clients are received and processed at the correct simulation time. Without a synchronization method, one client's integrator might advance time past the time needed to correctly process a message from a second client.

#### TIME-BASED PACKETS

The TimePacket class and its children, which are derived from the Packet class discussed in SENDING INFORMATION, are used by time-dependent clients to include temporal information in inter-client communications. The TimePacket class includes four separate time tags which define the simulated time at which the message is generated, transmitted, received and processed. These time tags allow the modeling of time delays in the communication between time-based client applications, and can be modified as appropriate by the model (client) at either end of the Service. Normally, the sender sets the generated and transmitted time tags, and the receiver sets the received and processed time tags. These time tags can be based on a local clients concept of time, which may need to differ from the universal integration time. This is useful for tracking and modeling skew and drift between spacecraft clocks in a multi-spacecraft formation. Figure 6 shows how two spacecraft's local time (LT) may differ from the universal time (UT). The red line shows spacecraft B transmitting a message to spacecraft A and how the local time on each spacecraft is different at the time this message is sent.

Upon receipt of a TimePacket the base client class automatically inserts the message in a cache, sorted based on the simulation time at which the message must be processed. HYDRA buffers the message in this cache until the integrator has reached the correct simulation time for processing the message. Once this happens HYDRA pauses the integration, generates a callback event for the TimePacket message, and then resumes integration. In this way all inter-client mes-

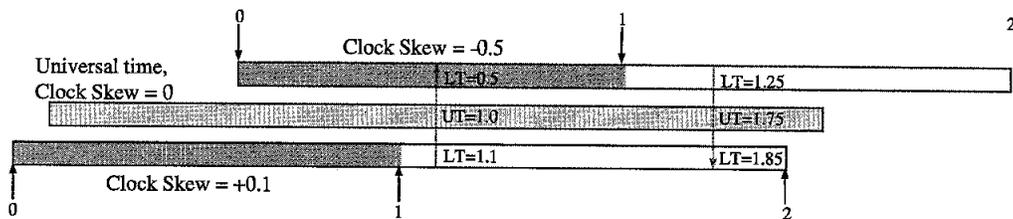


Fig. 6 Multiple Clock Skew Simulation in HYDRA.

sages can be processed by the client at the correct time, taking into account relative clock skew, transmission delays, processing delays, and without inappropriate temporal quantization caused by integration intervals.

#### PERIODIC SYNCHRONIZATION

Communication between time-dependent clients typically occurs at a periodic simulation rate. As the simulation time for each client is advanced via numeric integration, messages are sent to other clients based on the current simulation time. In a formation flying scenario, these periodic messages include inter-spacecraft communication, formation controller commands, relative sensor measurements, and the transmission of physical medium such as light signals. For the correct generation, processing, and coordination of these messages, the integration time of all clients must be synchronized before messages can be generated and processed. This synchronization prevents clients from advancing their local integrators until all clients have processed time tagged messages and completed integration cycles.

In order to provide synchronization of independent, time-based simulation clients, HYDRA offers a periodic *TimeTick* client with a broadcast service. Time-dependent clients register a *TimeTick Service* in order to receive synchronization messages at a simulation rate specified by the client. The separate *TimeTick* client then manages the potentially different rates of all connected clients in order to generate global synchronization commands at appropriate simulation times. Upon receipt of a *TimeTick* message, each client is commanded to advance its integrator to the next synchronization time, stopping off along the way (as described in TIME-BASED PACKETS) in order to process any buffered *TimePackets*. This activity is completely transparent to the client code. As each client finishes integrating forward to the next synchronization time, it sends a completion message to the *TimeTick* client. Once the *TimeTick* client has received a completion message from all connected clients, it waits for the next appropriate interval and generates the next *TimeTick* message. In a non real-time environment, these intervals are typically generated as soon as all synchronization replies are received.

At any point during this process, a client may transmit a *TimePacket*-derived message across a *Service*. As described in TIME-BASED PACKETS, these packets include information about the simulation time at which they are generated by the client. However, the current simulation time of a second client is non-deterministic. The only guaranty is that the second client has not integrated past the next synchronization time. This means that inter-client, time-based messages are only guaranteed to arrive at the synchronization boundaries. When a *TimePacket* is received, the second client will use the synchronization time and any modeled time-delays to place the message in the cache for processing during the next commanded time advancement by the *TimeTick* client. Since messages are only guaranteed to be delivered at the synchronization boundaries, clients often employ implicit message receipt blocking to prevent message receipt during time advancement to the next synchronization time. This implicit blocking prevents non-deterministic behavior of the overall simulation. Figure 7 shows how synchronization and communication/queuing of time tagged (TT) and un-time tagged packets (UTT) interact with time-dependent clients.

#### APPLICATION AND RESULTS

JPL has a significant research effort in the area of control and analysis for formation flying missions.<sup>9-11</sup> Advanced algorithms have been performance-validated on several generations of software-based formation flying testbeds, which are an invaluable tool for performing research activities in this area. However, past attempts to generate testbeds have been based on monolithic (non-distributed) simulation technologies (either self-written or based in COTS tools like Mathworks' Simulink), and past experience shows why this is challenging for a monolithic simulation. A previous two-spacecraft simulation with optics models and flexible-body dynamics required more than 60 minutes of CPU time on a Sun Ultra 10 computer to generate 30 seconds of simulated time in Simulink.

Another example testbed used kinematic models alone with perfect actuation to simulate five spacecraft simultaneously for Monte-Carlo analysis, and was able to achieve approximately 25 test cases per hour for ap-

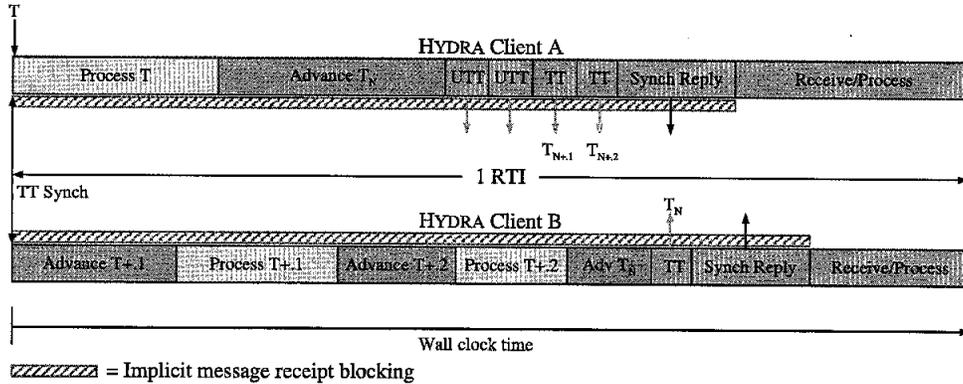


Fig. 7 Client Timeline and Synchronization.

proximately 1600 seconds of simulation per test case. This testbed was involved in testing the formation initialization problem,<sup>4</sup> wherein a multiple-spacecraft reset has occurred and each spacecraft is tasked with finding the others (an open research problem). In order to validate the new algorithm, many test cases were required and it was not feasible to generate them in a practical length of time using the existing testbed. Since HYDRA had been designed with exactly this application in mind, it proved a natural environment to both provide needed data to a research project, as well as validate the design and approach of the architecture. The next section will discuss the HYDRA clients used in the implementation of this testbed.

#### METHODOLOGY

Each spacecraft is simulated using a time-based, self integrating HYDRA client (ScClient). This client uses a derivative of NASA's DARTS<sup>12</sup> multi-body dynamics package, and a variable step integrator for state propagation.<sup>13</sup> The dynamic and kinematic configuration of the spacecraft is obtained from a spacecraft definition file at run time and allows a single, generalized client to simulate different spacecraft types. For this application, each spacecraft client includes simplified thruster and truth-based inertial sensor models. ScClient allows for a spacecraft to be initialized at an arbitrary position and velocity for use in Monte-Carlo.

A single HYDRA client is used to simulate the global physics of the AFF sensors on each spacecraft (AFFClient). Each AFF sensor provides relative sensor data (range and bearing) for spacecraft in its field of view, provided an AFF "lock" is achieved, which occurs when two AFF sensors face each other and are within a specified field-of-view cone. In order to determine if AFF lock exists, each spacecraft needs to know the position and orientation of all other spacecraft. A centralized AFFClient is used to reduce this

communication overhead. Each ScClient sends its current state to the AFFClient, rather than broadcast the state to all spacecraft clients. The AFFClient then determines which individual spacecraft have achieved AFF lock and sends each one of them the correct relative sensor data.

A HYDRA client (FormationInitClient) was created to implement a general version of the formation initialization algorithm outlined in.<sup>4</sup> This algorithm commands spacecraft maneuvers based on the relative sensor reported by each spacecraft and an internally estimated state for each spacecraft that has been seen at least once. If the algorithm is successful, all spacecraft will achieve zero relative velocity. Once relative velocity is nulled, a second algorithm (not simulated in this example) could be used to bring the spacecraft into a desired relative position and orientation. Testing the initialization algorithms for different numbers of spacecraft is very easy using the HYDRA framework since both the AFFClient and the FormationInitClient work for any number of connected spacecraft. The number of spacecraft being simulated is determined at run-time based on the number of ScClient applications that are executed.

The final client used in this demonstration application is a *TimeTick* client as discussed in PERIODIC SYNCHRONIZATION. This client provides synchronization between the distributed spacecraft clients.

#### RESULTS

Figure 8 shows the aggregate data for 50,000 simulation results the 5 spacecraft initialization case. The figure displays a histogram of the time (using 60 second bins) required for all spacecraft to null their relative velocities based on the given initialization algorithm. Various stages of the initialization algorithm are denoted on the figure. A large percentage (11%) of the simulations finish in the first 60 seconds of simulated

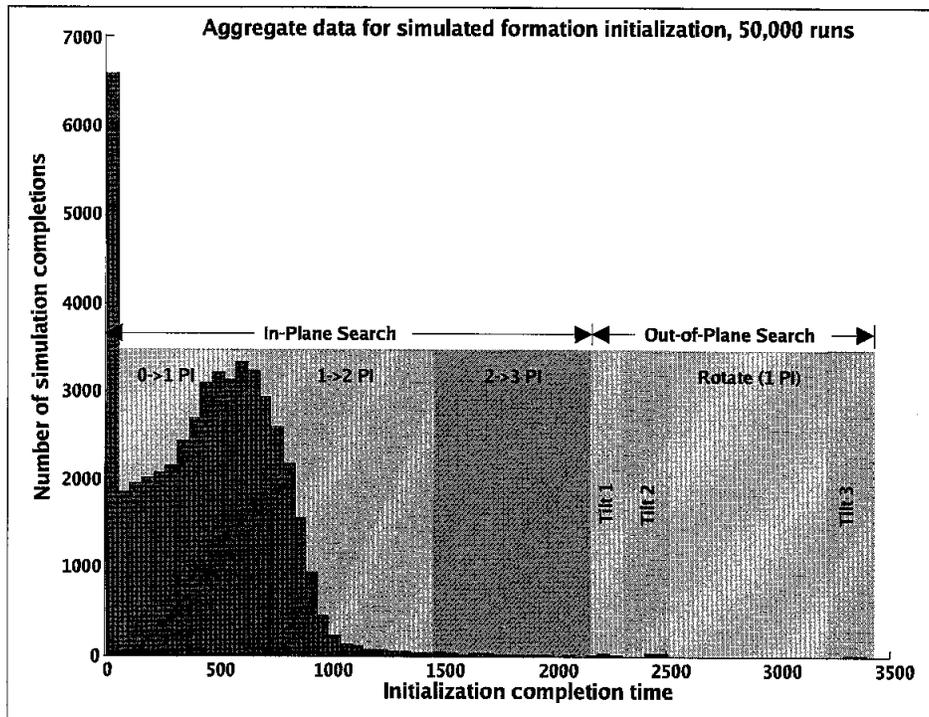


Fig. 8 Monte-Carlo Simulation Results

time. These are the result of starting the simulation with all 5 spacecraft in a positions which allow nearly instant AFF lock of the entire formation. Only a very small percentage ( $\approx 0.1\%$ ) of simulations required the out-of-plane search phase of the algorithm (See<sup>4</sup> for complete details and analysis of the algorithm). The average time for formation initialization was approximately 475 seconds of simulated time, which required about 2.5 seconds of CPU time.

50,000 test cases using the HYDRA framework required approximately 35 hours of computation time on a small COTS cluster computer (twelve 1.8 GHz CPUs). This same set of data would have taken about six weeks<sup>4</sup> to compute using the old monolithic simulation, which also had far less fidelity.

### FUTURE WORK

In the next year, we will apply the HYDRA framework to several current missions and research projects at JPL, including real-time flight software testbeds for the Space Interferometer Mission (SIM), and the Formation Algorithm Simulation Testbed (FAST). HYDRA will continue to gain new breadth and depth in connection automation and auto-negotiation, benefit

<sup>4</sup>The new, distributed, dynamic simulation is approximately 20 times faster than the older, kinematic-only simulator.

from the implementation of new multi-rate capabilities currently under research, and the expansion of the existing graphical, web-based interface to include not only complete status monitoring of the simulation and its components, but also control of the simulation and real-time data display and analysis. In addition, automatic archiving and time tagging of all transmitted data is planned, with an interface that allows easy browsing, searching, and graphing of historical data from a web interface.

### ACKNOWLEDGMENTS

The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Funding was provided by the Distributed Spacecraft Technology program. The authors thank Fred Hadaegh for his continuing support of research associated with formation flying simulation.

### REFERENCES

- <sup>1</sup>Schmidt, D. C., "ACE: and Object-Oriented Framework for Developing Distributed Applications," *Proceedings of the 6th USENIX C++ Technical Conference*, USENIX Association, Cambridge, Massachusetts, April 1994.
- <sup>2</sup>The Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 2.6*, <http://www.omg.org>, December 2001.

<sup>3</sup>U.S. Department of Defense, *High Level Architecture Interface Specification, Version 1.3, draft 1*, <http://www.dmsomil/projects/hla>, April 1998.

<sup>4</sup>Scharf, D. P., Ploen, S. R., Hadaegh, F. Y., Keim, J. A., and Phan, L. H., "Initialization of Distributed Spacecraft Formations," *Submitted*, 2003.

<sup>5</sup>Knuth, D. E., "Literate Programming," *The Computer Journal*, Vol. 27, No. 2, May 1984, pp. 97-111.

<sup>6</sup>van Heesch, D., "Doxygen," <http://www.doxygen.org>, 2003.

<sup>7</sup>Locke, Weinberger, Searls, and Levine, "Cluetrain Manifesto," <http://www.everything2.com/index.pl?node=Cluetrain%20Manifesto>, 1999.

<sup>8</sup>Pereira, J. O., "eXternalization Template Library," Tech. rep., Departamento de Informática, Universidade do Minho, <http://xtl.sourceforge.net/>, 1999.

<sup>9</sup>Scharf, D. P., Hadaegh, F. Y., and Kang, B. H., "A survey of spacecraft formation flying guidance," *Proc. CNES Int. Symp. on Formation Flying Missions and Technologies*, Toulouse, France, October 2002.

<sup>10</sup>"Collision avoidance guidance for formation-flying applications," *Proc. of the AIAA Guidance, Navigation, and Control Conference*, Montreal, Canada, October 2001, pp. paper AIAA-01-4088.

<sup>11</sup>Hadaegh, F. Y., Ahmend, A., and Shields, J., "Precision guidance and control for formation flying spacecraft," *Proc. CNES Int. Symp. on Formation Flying Missions and Technologies*, Toulouse, France, October 2002.

<sup>12</sup>Jain, A. and Man, G., "Real-Time Simulation of the Cassini Spacecraft Using DARTS: Functional Capabilities and the Spatial Algebra Algorithm," *5th Annual Conference of Aerospace Computational Control*, Aug. 1992.

<sup>13</sup>Cohen, S. and Hindmarsh, A., "CVODE, a Stiff/Nonstiff ODE Solver in C," *Computers in Physics*, Vol. 10, No. 2, March-April 1996, pp. 138-43.