Proceedings of the International Conference on Robotics and Automation, IEEE, May 1994

The Network Data Delivery Service: Real-Time Data Connectivity for Distributed Control Applications

Gerardo Pardo-Castellote*
Stanford Aerospace Robotics Laboratory
pardo@sun-valley.stanford.edu

Stan Schneider†
Real-Time Innovations, Inc.
stan@rti.com

Abstract

The Network Data Delivery Service (NDDS) is a novel network data-sharing system. NDDS builds on the model of information producers (sources) and consumers (sinks). Producers generate data at their own discretion, unaware of prospective consumers. Consumers "subscribe" to data-updates without concern for who is producing them. The routing protocol is connectionless and nearly "stateless," thus network reconfigurations, node failures, etc. are handled naturally. This scheme is particularly effective in systems (such as distributed control systems) where information is of a repetitive nature.

NDDS provides support for multiple producers, reliable data-delivery, consumer update guarantees, notifications vs. polling for updates, dynamic binding of producers and consumers, distributed queries, user-defined data types etc. NDDS is integrated into the ControlShell *real-time framework and is being used in several robotic applications as an effective means of information sharing between sensor systems, robot controllers, planners, graphical user interfaces, simulators etc.*

1 Introduction

Many control systems are naturally distributed. This is due to the fact that often they are composed of several physically distributed modules: sensor, command, control and monitoring modules. In order to achieve a common task, these modules need to share timely information. Robotic systems are a prime example of such distributed control systems.

These information sharing needs are common to many other application environments such as databases, distributed computing, parallel computing, transaction systems, etc. However, distributed control applications have some unique requirements and characteristics:

- Data transactions are often time-critical. For control purposes data must get from its source to its destination with minimum delay.
- Computations are often data-driven, that is, triggered by the arrival of new data. For instance, a collision-avoidance plan may need to be re-evaluated as soon as a new obstacle is detected.
- A significant portion of the data flow is repetitive in nature. This is true of sensor readings, motor commands etc. For this type of data, data loss is often not critical: sending data is an idempotent operation and new updates just replace old values. Considerable overhead can be avoided by using a data transfer paradigm that exploits these facts.
- There are often multiple sources of (what may be considered) the same data item. For example, a robot command might be generated by a planner module as well as a tele-operation module. Similarly there can be many data consumers. A robot and a simulator are both sinks of "command-data." The network of data producers and consumers may not be known in advance and may change dynamically.
- Data requirements are ubiquitous and unpredictable. It is often difficult to know what data will be required by other modules. For instance, force-level measurements—normally used only by a low-level controller—may be required by a sophisticated high-level task planner in the future. The architecture should support these types of data flow. Thus, vital data should be accessible throughout the system.
- Most data flow can and should be anonymous. Producers of the sensor readings can usually be

*Department of Electrical Engineering, Stanford University
†Real-Time Innovations, 954 Aster, Sunnyvale, CA 94086, USA, (408) 720-8312

unaware of who is reading them. Consumers may not care about the origin of the data they use. Hiding this information increases modularity by allowing the data sources and sinks to change transparently.

The *Network Data Delivery Service* (NDDS) has been developed to address these unique needs. NDDS provides transparent network connectivity and data ubiquity to a set of processes possibly running in different machines. NDDS allows distributed processes to share data and event information without concern for the actual physical location and architecture of their peers[1]. NDDS allows its "clients" to share data in two ways: subscriptions and one-time queries.

NDDS supports "subscriptions" as a fundamental means of communication. In the application context described, subscriptions have some fundamental advantages over other information sharing models (such as client-server or shared-memory). Subscriptions cut in half the data latency over query/response type models and it allows synchronization on the latest available information as soon as it is produced.

NDDS supports multiple information sources (producers) and users (consumers). It provides clear semantics for multiple-producer conflict resolution, provides support for and guarantees multiple update rates (as specified by the consumers). NDDS's implementation is nearly "stateless" and internally uses decaying state to ensure inherently robust communication.

NDDS is integrated into the *ControlShell* real-time programming framework [5, 11] and is being been used in several applications including the control of a two-armed robotic system [9], an underwater vehicle [14], and a self contained, two-armed space robot originally described in [13].

This paper describes the philosophies behind the NDDS system, and details an example application of a dual-arm robotic system capable of planning and executing complex actions under the control of an interactive graphical user-interface.

2 Relation to other research

There is a large body of literature covering information sharing in distributed computer systems [1, 2]. Systems that support some of the most popular schemes, such as shared-memory, have been developed for robotic applications [6]. The shared memory abstraction is intuitive and well understood. However, it is hard to implement efficiently in a distributed environment without special hardware [15], doesn't scale well to large systems and does not take advantage of the characteristics of Distributed Control Systems (DCS) that were listed in the introduction.

In the last few years, several data-sharing systems that take advantage of special characteristics of DCS have been developed: MBARI's Data Manager [7], CMU's TCX [4], Rice University's TelRIP [16] and Sparta's ARTSE [12] all offer network-transparent connectivity across different platforms and support *subscriptions* as means of communication where multiple consumers can get updates from a single producer. Of these, only the Data Manager provides support for multiple (consumer-specified) update rates. And only TelRIP supports multiple producers of a single data item. None of the above architectures combine the above facilities with NDDS's fully-distributed, symmetrical implementation (absence of privileged nodes) nor use a re-startable handshake-free stateless protocol.

3 The NDDS communication model

The NDDS system builds on the model of information producers (sources) and consumers (sinks).

Producers register a set of data instances that they will produce, unaware of prospective consumers and "produce" the data at their own discretion. Consumers "subscribe" to updates of any data instances they require without concern for who is producing them. In this sense the NDDS is a "subscription-based" model. The use of subscriptions drastically reduces the overhead required by a client-server architecture; Occasional subscription requests, at low bandwidth, replace numerous high-bandwidth client requests. Latency is also reduced, as the outgoing request message time is eliminated.

NDDS identifies data instances by name (their *NDDS name*). The scope of this name extends to all the tasks sharing data through NDDS. Two instances with the same NDDS name are viewed by NDDS as different updates of the *same* data instance and are otherwise indistinguishable to the client. If two data instances must be distinguished by any NDDS client, they must be given a different NDDS name.

NDDS requires all data instances to be of a known type. NDDS has some built in types (such as strings and arrays) but most data flow consists of user-defined types. Creating a new *NDDS type* involves binding a new type-name with the functions that will allow NDDS to manipulate instances of that type.

[1] NDDS is being used to communicate between Sun, HP and DEC workstations as well as VME-based real-time processors running the VxWorks operating system.

Function	Action
NddsRegisterProducer	Register and specify producer parameters
NddsProduce	Add an instance produced by the producer
NddsSampleProducer	Take a snap-shot of all the instances produced by the producer. For immediate producers also send updates to all consumers.
NddsRegisterConsumer	Register and specify consumer parameters
NddsSubscribeTo	Add a subscription to a consumer. Specify a call-back routine to be called on updates
NddsReceiveUpdates	Poll the consumer for updates. Will result on call-back routines being called when applicable. Required only of polled consumers.
NddsQueryInstance	Issue a one-time query.

Table 1: **Functional interface to produce, consume and query data.**

Producing data involves three phases: Registering (declaring) a producer, declaring the instances the producer will produce and sampling the producer. Receiving data updates also involves three phases: Registering (declaring) a consumer, declaring the instances that the consumer subscribes to along which the action to be taken and lastly receiving the updates. The last phase is only required for polled consumers.

NDDS treats producers and consumers symmetrically. Each node maintains the information required to establish communications. Producers inform prospective consumers of the data they produce. Consumers use this information to either subscribe to data or issue one-time queries. Table 1 lists the steps involved in becoming a producer or consumer of data.

3.1 Producer characteristics

A producer can be compared to a multi-channel Sample-and-Hold. It is associated with a set of object instances (similar to the signal channels) that get sampled synchronously. Sampling a producer takes a sample of the values of each data item the producer has associated with it. The data is either immediately distributed (for *immediate* producers) or saved for later distribution (*delayed* producers).

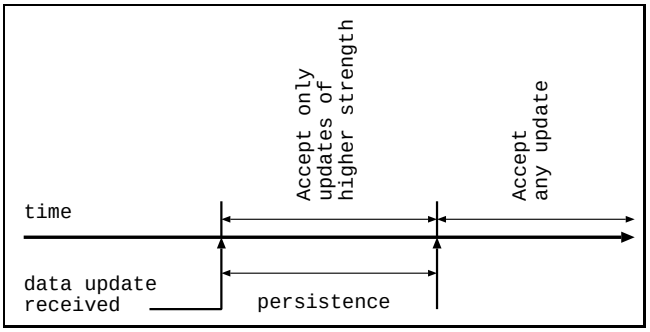

Figure 1: **Multiple producer conflict resolution.**

NDDS resolves the multiple-producer conflict by characterizing each producer with two properties: the producer's strength *and its* persistence. *When a data update is received for some object instance, it is accepted if either the producer's strength is greater (or equal) to that of the producer of the last update for that instance or, the time elapsed since the last update was received exceeds the persistence of the producer of the last update. In essence the strength is like a priority and the persistence is the duration for which the priority is valid.*

A producer is characterized by three parameters: *production rate*, *strength* and *persistence*. The strength and persistence parameters are used to resolve multiple-producer conflicts. Their meaning is illustrated in Figure 1. A producer's data is used while it is the strongest source that hasn't exceeded its persistence. Typically a producer that will generate data updates every period of length T, will set its persistence to some time T_p where $T_p > T$. Thus, while that producer is functional, it will take precedence over any producers of less strength. Should the producer stop distributing its data (willingly or due to a failure), other producers will take over after T_p elapses. This mechanism establishes an inherently robust, *quasi-stateless* communications channel between the strongest producer of an instance and all the consumers of that instance.

3.2 Consumer characteristics

Consumers are *notification based*. They subscribe to a set of instances (identified by their NDDS name) by providing *call-back functions* for each instance they subscribe to. When a data update for a subscription arrives, the call-back function of every consumer is called with the data as a parameter.

Two consumer models are currently supported: *immediate* and *polled*. An immediate consumer will be

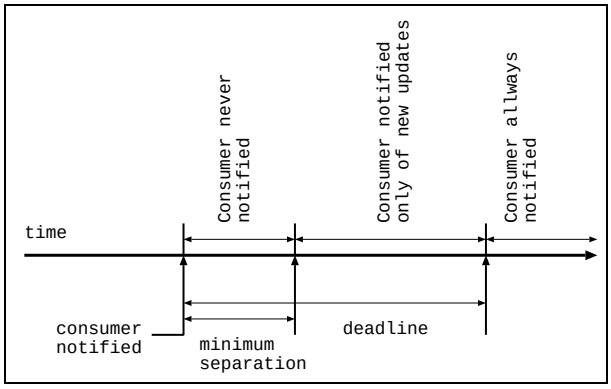

Figure 2: **Consumer notification rate.**

The NDDS characterizes consumers requests for periodic updates with two properties: the consumer's minimum separation *time and its* deadline. *Once the consumer is called with an update for an object instance, it is guaranteed not to be notified again of the same instance for at least the minimum separation time. The deadline is a the maximum time the consumer is willing to wait for a new update. Even if new updates haven't arrived, the call-back routine will be called when the deadline expires.*

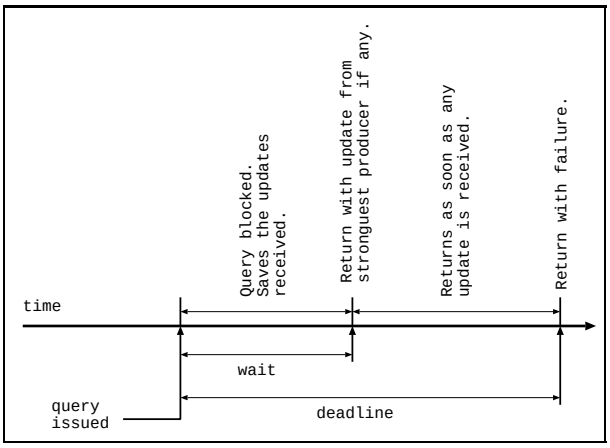

Figure 3: **One-Time Query Parameters.**

A one-time query specifies two parameters: the wait *time and the* deadline. *A query will block for at least wait time during which the data arriving from the producer with higher strength is saved. At the end of the wait time the query returns if any data has been received. Otherwise, query remains blocked until either an update comes or the deadline expires (whichever comes first).*

called back as soon as the data update arrives. A polled consumer will not be called back until it itself "polls" for updates.

Consumers are characterized by two parameters, the minimum separation and the deadline (see figure 2). These parameters are used to regulate consumer update rates. Consumers are guaranteed updates no sooner than the minimum separation time and no later than the deadline.

Typically the minimum separation protects the consumer against producers that are too fast whereas the deadline provides a guaranteed call-back time that can be used to take appropriate action (the expiration of the deadline typically indicates lack of producers or communications failure).

3.3 One-time queries

A client task may issue one-time queries for specific NDDS data instances.

Queries are blocking calls. Aside from specifying the name and type of the NDDS data instance, a query contains two parameters: the *wait* and *deadline* illustrated in figure 3. These parameters regulate the tradeoff between returning as soon as data becomes available and waiting for "better" data. The use of these parameters make the latency of this call predictable, allowing its use from real-time application code. Typically the wait is set to be long enough to account for communication delays from all data producers to the consumer. The deadline provides a guaranteed call-back time in case no responses arrive. Setting a wait time to zero causes the first response to be accepted.

3.4 Reliable updates

By default NDDS provides unreliable, unacknowledged data updates from producers to consumers. While this gives the most throughput with minimum overhead, it may result in occasional loss of updates or out-of-order arrival. For sensory-type data, having the latest data as soon as it becomes available ,is usually more important than occasionally missing an update. Other kinds of information, such as commands, often present in distributed control systems, would be better served with a reliable protocol.

NDDS supports reliable updates. A producer may specify any one of its productions to be delivered reliably. Reliable updates get grouped together in special packets that are individually acknowledged. The producer is notified if the update isn't acknowledged by a specified deadline and/or if another reliable pro-

duction is attempted before the last one was acknowledged. This guarantees in-order update delivery at the expense of reduced update bandwidth. A window size larger than one may be specified to compensate for longer communication delays so that multiple reliable updates can be sent before any acknowledgement is received.

4 Implementation

NDDS is symmetrically distributed, that is, there are no "special" or "privileged" nodes nor name servers. All NDDS nodes are functionally identical and each node maintains its own copy of the NDDS database and contains the helper processes necessary to implement the communication model described above.

NDDS uses UDP/IP [10, 3] as a means of communication. To allow communications between computers with different data formats the External Data Representation (XDR) [8] is used.

4.1 Architectural overview

An NDDS *node* is composed of one or more NDDS client processes (each with its respective NDDS Server Daemon) a copy of the NDDS database and three daemon (helper) processes that maintain the database and implement the NDDS communication model described above. See figure 4.

The user task becomes an NDDS client by linking to the NDDS library. Each NDDS client process spawns a private NDDS Server Daemon process that will assist in establishing the subscriptions and informing the peer nodes of the user productions. There is at most one NDDS node per address space so in operating systems that support shared memory threads (for example VxWorks), several NDDS client processes may belong to the same node (sharing the same copy of the NDDS database and helper daemons[2]).

The following is a functional description of the different daemons:

- **NDDS Forwarding Daemon** (NFD). There is one NFD per network host. All the Request Receiver Daemons running on the host register with the NFD. Production notifications and subscription requests received by the NFD daemon are immediately forwarded to all the Request Receiver Daemon(s) running on the host.

[2]In operating systems that don't support shared memory threads, such as Unix, the helper daemons are not independent tasks but rather are installed as signal handlers.

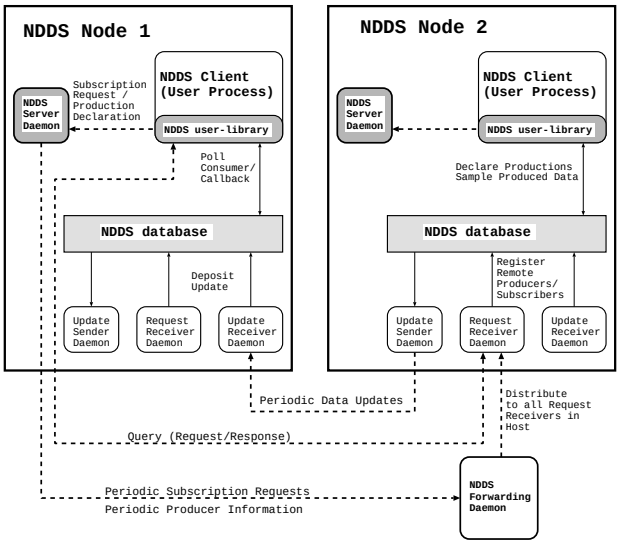

Figure 4: **Communication between NDDS nodes.**

A NDDS node *is composed of one or more NDDS* clients *and three helper daemons that share a local copy of the NDDS database. Each NDDS* client *has a private NDDS Server Daemon that informs other NDDS* nodes *of the productions and subscriptions of the client. The three helper daemons are responsible for maintaining the NDDS database, sending updates to remote subscribers, receiving updates and servicing queries. There is one NDDS Forwarding Daemon per Network host.*

- **NDDS Server Daemon** (NSD). Each NDDS client (user-task) spawns its private NSD. The NSD is responsible for periodically informing the all the other NDDS nodes of both the subscription requests and the productions of the NDDS client.
- **Request Receiver Daemon** (RRD). There is one RRD per NDDS node. The RRD is responsible for maintaining the remote subscriptions and productions in the NDDS database. Stale productions and subscription requests are aged and eventually dropped by the RRD. This daemon is also responsible for replying to one-time *queries* from other NDDS nodes.
- **Update Sender Daemon** (USD). There is one USD per NDDS node. The USD is responsible for sending the updates of locally produced data items to the subscribers in other NDDS nodes. This daemon also ensures that the the timing parameters requested by the consumer are met.
- **Update Receiver Daemon** (URD). There is

one URD per NDDS node. The URD is responsible for receiving updates for the local subscriptions of the nodes. The URD solves multiple-producer conflicts and, in the case of *immediate* consumers, executes the callback routine(s) installed for that data item. The URD also ensures that the timing parameters requested by the each consumer in the node are met.

4.2 Data management overview

The NDDS database is replicated and maintained on each NDDS *node* by three helper daemons (the Request Receiver Daemon, Update Sender Daemon and Update Receiver Daemon). The database stores and cross references producers and the data they produce, consumers and the data they consume, remote productions, subscriptions requested by both the NDDS clients in both the local and remote NDDS nodes, etc.

Consistency between databases across different NDDS nodes is not necessary and requires no special effort. Temporary inconsistencies between databases may result on subscription requests (or queries) not reaching all the producers of a given data item and, as a consequence, different nodes may get data from different producers. A similar situation may result from the data loss due to communication failure. At worst this will be a transient situation that arises only if there are multiple producers of the same data.

All information about remote NDDS nodes is aged and is eventually erased unless it is refreshed. The NDDS Server Daemon associated with each NDDS *client* is responsible for the periodic refreshment of the information that concerns that NDDS *client* . This mechanism is inherently robust to remote node failures, communication dropouts and network partitioning. Furthermore, it requires no special recovery mechanisms.

5 Applications

This section describes how a distributed robotic application exploits NDDS unique facilities to build a modular expandable system that integrates planning into a two-armed robotic work-cell [9]. The system is composed of four major components: user interface, planner, the dual-arm robot control-and-sensor system, and on-line simulator. The graphical user interface provides high-level user direction. The motion planner generates complete on-line plans to carry out these directives, specifying both single and dual-armed motion and manipulation. Combined with the robot control and real-time vision, the system is capable of performing object acquisition from a moving conveyor belt as well as reacting to environmental changes on-line.

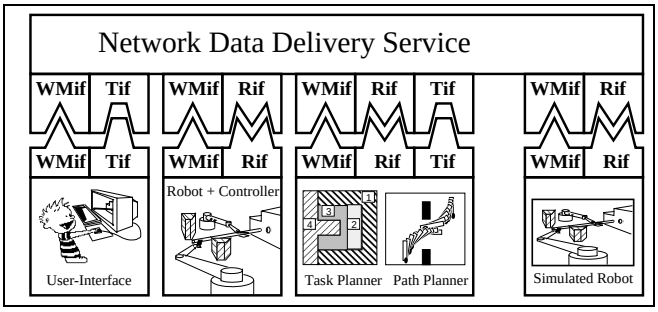

Figure 5: **Application Example: Task-Level control of a Two-Armed robot.**

This example shows the main application modules. Each module communicates using one or more of the three interfaces: The World Model interface (WMif), the Robot Interface (Rif) and the Task Interface (Tif). These modules are physically distributed. All the interfaces are built using the Network Data Delivery Service (NDDS). NDDS plays the role of a bus providing the necessary module interconnections.

The use of NDDS as the underlying communication mechanism provides unique benefits to this application without requiring any special programming.

- The different modules can be distributed across different computer systems (with different processor architectures and operating systems)[3].
- Several copies of the *same* module can be run concurrently. For example, the graphical user interface module can be run on several workstations. This allows multiple users to monitor the system and permits *simultaneous* interaction with the robot system.
- The graphical simulator module can *masquerade* as the real robot and allow independent testing of the remaining modules in the system. Any time the real robot goes on-line, its productions override those of the simulator[4] and all the remaining modules are now connected to the real robot.

[3] This experiment has modules running on DEC Workstations, Sun Workstations and several VME-based real-time processors.

[4] Thanks to NDDS's multiple-producer conflict resolution mechanism.

- Should the planner be unable to generate adequate plans for specific situations due to limitations or malfunctions, a teleoperation module (under development) can override planner commands and take control of the robot.

- The modules can be started in any order. Future modules (such as the teleoperation-module mentioned above) can be *dynamically* added to the system even if it is already in operation or deployed[5].

6 Conclusions

This paper has presented NDDS, a unique data-sharing scheme that allows programs distributed on a computer network to share data and event information unaware of the location of their peers. These facilities provide fundamental new capabilities to distributed control systems that use NDDS as the underlying communication mechanism. This paper has also discussed an application that uses NDDS to communicate between different modules that integrate planning into a two-armed robotic work-cell. Several other applications are cited in the paper.

Acknowledgements

NDDS is being jointly developed by Stanford University and Real-Time Innovations, Inc. It is currently supported under ARPA's Domain-Specific Software Architectures (DSSA) program. The authors wish to expressly thank Dr. R. H. Cannon, Jr. for his guidance and leadership. The authors would also like to thank the many developers at Stanford and MBARI for their support and insight.

References

[1] Henri E. Bal, Jenniffer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computer systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[2] R. S. Chin and S. T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1):91–123, March 1991.

[3] Douglas E. Comer. *Internetworkning with TCP/IP*, volume 1-3. Prentice Hall, Englewood Cliffs, N.J., 2 edition, 1991-92.

[4] Chris Fedor. TCX task communications. School of computer science / robotics institute report, Carnegie Mellon University, 1993.

[5] Real-Time Innovations Inc. *ControlShell: Object Oriented Framework for Real-Time Software User's Manual*. 954 Aster, Sunnyvale, California 94086, 4.2 edition, August 1993.

[6] J. P. Jones, P. L. Butler, S. E. Johnston, and T. G. Heywood. Hetero helix: synchronous and asynchronous control systems in heterogeneous distributed networks. *Robotics and Autonomous Systems*, 10(2-3):85–99, 1992.

[7] MBARI (Monterrey Bay Aquarium Research Institute. Data manager user's guide. Internal Documentation, 1991.

[8] Sun Microsystems. Xdr: External data representation standard. RFC 1014, June 1987.

[9] Gerardo Pardo-Castellote, Tsai-Yen Li, Yoshihito Koga, Robert H. Cannon Jr., Jean-Claude Latombe, and Stan Schneider. Experimental integration of planning in a distributed control system. In *Preprints of the Third International Symposium on Experimantal Robotics*, Kyoto Japan, October 1993.

[10] J. Postel. User datagram protocol. RFC 768, June 1980.

[11] S. A. Schneider, V. W. Chen, and G. Pardo-Castellote. Controlshell: A real-time software framework. In *Proceedings of the AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service and Space*, Houston, TX, March 1994. AIAA, AIAA.

[12] Sparta, Inc., 7926 Jones Branch Drive, McLean, VA 22102. *ARTSE product literature.*

[13] M. A. Ullman. *Experiments in Autonomous Navigation and Control of Multi-Manipulator Free-Flying Space Robots.* PhD thesis, Stanford University, Stanford, CA 94305, March 1993. Also published as SUDAAR 630.

[14] Howard H. Wang, Richard L. Marks, Stephen M. Rock, and Michael J. Lee. Task-based control architecture for an untethered, unmanned submersible. In *Proceedings of the 8th Annual Symposium of Unmanned Untethered Submersible Technology*, pages 137–147. Marine Systems Engineering Laboratory, Northeastern University, September 1993.

[15] T. Williams. Fiber network supports distributed real-time systems. *Computer Design*, 29(17):60–62, September 1990. dd.

[16] J. D. Wise and Larry Ciscon. *TelRIP Distributed Applications Environment Operating Manual.* Rice University, Houston Texas, 1992. Technical Report 9103.

[5] This facility may prove crucial for other current applications such as the undersea vehicles.