RoboTalk: Controlling Arms, Bases and Androids through a Single Motion Interface

A.Y. Yang[*] H. Gonzalez-Banos[†] V. Ng-Thow-Hing[†] J.E. Davis[†]

[*]Coord. Science Lab, U. of Illinois
1308 West Main
Urbana, IL 61801
yangyang@uiuc.edu

[†]Honda Research Institute, USA
800 California St. 300
Mountain View, CA 94041
{hhg,victorng,jdavis}@honda-ri.com

***Abstract*— Despite several successful humanoid robot projects from both industry and academia, generic motion interfaces for higher-level applications are still absent. Direct robot driver access proves to be either very difficult due to the complexity of humanoid robots, very unstable due to constant robot hardware upgrade and re-design, or inaccessible due to proprietary software and hardware. Motion interfaces do exist, but these are either hardware-specific designs, or generic interfaces that support very simple robots (non-humanoids). Thus, this paper introduces RoboTalk, a new motion interface for controlling robots. From the ground up our design model considers three factors: mechanism-independence to abstract the hardware from higher-level applications, a versatile network support mechanism to enable both remote and local motion control, and an easy-to-manage driver interface to facilitate the incorporation of features by hardware developers. The interface is based on a motion specification that supports a wide range of robotic mechanisms, from mobile bases such as a Pioneer 2 to humanoid robots. The specification allows us to construct interfaces from basic blocks, such as wheeled bases, robot arms and legs. We have tested and implemented our approach on the Honda ASIMO robot and a Pioneer 2 mobile robot.**

I. Introduction

In the last decade or so, a handful of humanoid robots have been successfully developed by both industry and academia. Some of the famous examples include ASIMO from Honda [1], Qrio from Sony [2], and Open Pino [3] from Japan Science and Technology Agency. But there is still a lack of a common control interface for humanoid robots to interact with higher-level applications. Researchers within development teams must write their own driver interfaces to specific robots, modifying these programs repeatedly as the hardware evolves. This is inefficient because common algorithms are constantly reimplemented in different hardware.

Android development can greatly benefit from software development trends in non-humanoid robotics. In fact, general interfaces for non-humanoid robots do exist. The example of Player [4], [5] comes to mind. The Player Project supports a wide range of robotic platforms thanks to the collective effort of many developers. However, the interfaces found in Player tend to support very simple mechanisms from the point of view of motion. Moreover, available interfaces (within or without Player) are usually *direct-drive* approaches: a client issues a command, robot executes command, control is returned to client, and client issues a new command. Under this approach motions are executed frame by frame, and the inter-frame timings are lost. This is often un-desirable in humanoid robots (e.g., during a dance sequence). Direct-drive is not about running network communications in blocking vs. non-blocking mode. It is about the lack of efficient buffering mechanisms to allow motion playback in the presence of latency and bandwidth limitations in network communications.

A. Overview

This paper introduces RoboTalk, a novel interface for general robot motion control. Our motion interface has the ability to communicate with a wide range of humanoid or mobile robots, and provides a unified user interface for higher-level applications. Our key motivation was to ensure cross-robot, cross-network, and cross-OS compatibility. To achieve this, we divided our development cycle into three phases:

The first design phase was to develop a motion specification to address cross-robot compatibility. This specification is a collection of descriptors about the robot geometric model, joint and link configuration, kinematic constraints, motion states and supported control commands. Although robots have different mechanical capabilities, the specification is general enough to describe the motions of a wide variety of robots.

The second design phase dealt with the network communication model: transmission protocol, client/server interaction mechanisms, support for multi-client connections, reply caches, etc. RoboTalk is based on the concept of *frames*, where a frame is a set of motion descriptors with the same time stamp. This is akin to a video stream where a frame is a set of image pixels sampled at the same instant. Buffering mechanisms in RoboTalk deal with network congestion by preserving inter-frame timings in the command stream at the expense of latency.

The final design stage was the selection of libraries that could be easily ported across platforms. To achieve cross-OS portability we avoided the use of proprietary software tools or libraries. Our choice of programming language was C++, but RoboTalk's specification itself is language independent. So far we have compiled and run RoboTalk clients and servers in machines running Linux kernels 2.4 and 2.6, Mac OSX, and Windows XP running Cygwin.

RoboTalk is not a complete robot architecture. For instance, it does not cover sensor interfaces (except for measurements of kinodynamic variables), nor does it provide an environment for implementing real-time controllers (although it can be used to interface with a controller in real-time). RoboTalk is just a motion interface, but one designed to easily complement other robot architectures.

B. Organization

This paper is organized as follows: Section II surveys previous work that is related to our research. Section III describes in detail our motion specification and communication model, while Section IV describes our implementation. Section V explains the four command modes supported by RoboTalk: *direct*, *delay*, *playback*, and *broadcast* modes. Finally, Section VI describes a prototype interface connected to a Honda humanoid robot and a Pioneer 2 equipped with a Sony EVI pan-tilt camera. A single client program issues motion sequences to both devices across the network. The motion sequence plays back with the same inter-frame timings due to RoboTalk's buffering mechanism.

II. Previous Work

RoboTalk serves as a specification for robust communication of robot configuration information and motion commands between a high-level control application and the robot hardware. Previous teleoperation systems have been designed to cope with the problem of scheduling, and sending motion commands to a robot over a communication link. The *Robonaut* [6] architecture allows high-level commands to be converted by modules called subautonomies into low-level motor commands. The Athena software development model [7] for Mars rovers features command sequencing to schedule and execute commands sent from the control application to the rover. RoboTalk allows control applications to be written in a manner that can be easily reused or reconfigured at runtime for changing robot platforms with a common command interface. There exist several public and commercial software projects for interfacing with robots. Some examples are the following:

ARIA (ActivMedia Robotics Interface Application) is an application programming interface developed under the object-oriented model [8]. ARIA communicates with the robot via a client/server relationship through serial or TCP/IP connections, but it lacks buffering mechanisms. It is designed to interface only with ActivMedia's mobile bases.

OROCOS (formerly Open Robot Control Software, now Open *Real-time* Control Software) is a free software project that focuses in real-time control [9]. It follows a component-based design, where each component transmits their complete state in a single call. OROCOS is ideally suited for feedback control systems, but it is not an interface for higher-level descriptions (such as joint and link configuration and geometry). Neither is OROCOS intended for motion playback under network limitations.

OAP (Open Automaton Project) focuses on hardware implementation. The goal of OAP is to provide inexpensive designs for building robots [10]. The Open Pino Project is a GNU project designed to control commercial Pino robots [3]. Thus, it lacks the potential to be a general platform to access other humanoids, mobile robots or human models.

Open HRP (Open Architecture Humanoid Robotics Platform) provides an abstraction layer between the hardware details of the robot and its controller systems [11]. The same controller can be used on both real and simulated versions of the robot. RoboTalk adopts the same philosophy of hardware abstraction, but expands the class of robots to non-humanoids and provides different modes of network communication independently from the target robot.

A general robot control platform is Player [4], [5]. While it was designed to provide a control interface for general mobile robots and sensors, it currently has the following limitations:

1) Player server is designed to directly access the robot/sensor drivers, which is not a good option for black-box components or changing hardware.
2) Player lacks the ability to define and access individual joints, links and control frames on a humanoid or a kinematics simulator. It can be argued that this capability can be added to Player, but this is far from a trivial addition. (In comparison, in our system it is almost trivial to redefine a pan-tilt camera as a head link on top a Pioneer 2 mobile robot.)
3) The Player server cannot issue multiple commands to different motors in a robot at the same time. In androids, it is important to have the ability to issue simultaneous commands because it allows the possibility of whole body coordination.
4) There is no mechanism in Player to customize Panic responses to prevent disastrous outcomes on different types of expensive robots. This modification again is not trivial, because the communication model must understand (as opposed to just communicate) the concept of Panic in order to preempt other actions.
5) The communication model between server and client in Player was designed in blocking mode, meaning the client cannot process a new command before it receives an acknowledgment about the previous command. This is equivalent to direct mode in our system (Section V-A).
6) Finally, there are no command buffers or return caches built within Player. Thus, a motion sequence sent over long distances will not replay in the same way when it is sent locally.

In the following sections, we will explain in detail the design and implementation of RoboTalk.

III. RoboTalk Design

Figure 1 illustrates the structure of the interface architecture, which adopts a specification-centered design. A general robot specification standard is at the center of the interface as a format to describe the configuration of an arbitrary robot and its control commands. The server and client implement a mechanism to query and update the specification values through the execution of remote functions. Also, the server

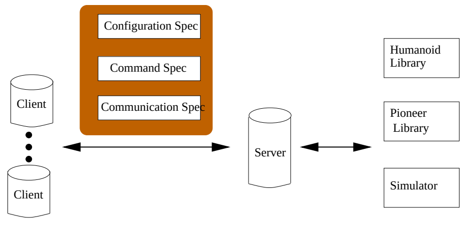


Fig. 1. Interface design model.

has the freedom to link a list of driver modules for different robots and simulators within the same core implementation.

A. Motion Specification Overview

Having a specification standard at the center of RoboTalk's architecture immediately provides four benefits:

1) The client sees and understands a unified set of motion descriptors. As long as the descriptors can be encoded and decoded, the client operation is independent from the server implementation.
2) Likewise, behind the server, the driver has implementation freedom. This implementation is independent from the client, and the driver development is free to reinterpret the specification for individual robots.
3) The client side avoids the use of proprietary information about the robot driver. Only the capabilities of the robot are disclosed by the descriptors.
4) The client and server can use different OS and programming languages, selecting those that better serve their individual needs.

As an example of the above, consider the operations underpinning a GOTO(x, y, θ) locomotion command. In a humanoid robot this is implemented through the leg drivers, which execute a walking gait preserving the robot's upright balance. Yet, in a Pioneer 2 robot the GOTO command is a call to far simpler wheel drivers since there are no balance issues. And for a PUMA arm mounted on a table the GOTO command is non-existent, in which RoboTalk dictates that "N/A" is returned to the client. Throughout the above range of examples the client issues the same GOTO function. The result varies across different robots, but the syntax remains the same.

RoboTalk's specification consists of three levels. The first level is the set of *robot configuration* descriptors. It describes the kinematics model, the states and constraints of the body, joints and links of a robot, and the control frames supported by the robot. The second level defines the set of *robot commands* and their arguments. These commands query and update the configuration descriptors. The third level defines a *communication* protocol. This describes the format by which client and server interact across a network.

B. Robot Configuration Descriptors

When an application written with the RoboTalk specification establishes communication with an unknown robot, a request for its physical constraints and kinematic joint configuration can be made. RoboTalk uses a fixed-length configuration

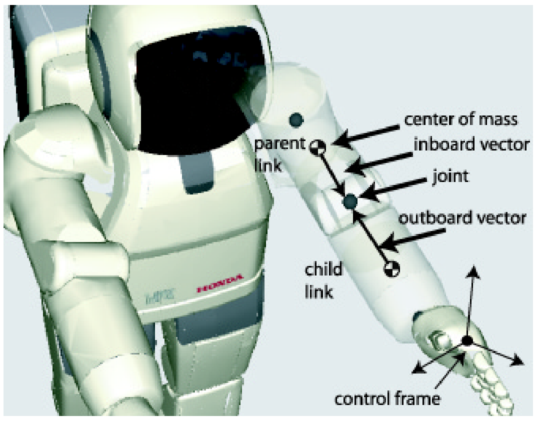


Fig. 2. Robot configuration of links and joints

header containing all global parameters of the robot to load the header efficiently into pre-defined data structures. The header contains three sections to register offset indices for joints, links, and control frames in the robot. Joints are stored in a single memory block whose size is registered in the header, and each joint is cross-referenced with the links joined by it. Links and control frames are also organized in this way. Queries about the robot's configuration are made from the information stored in this header.

Our configuration descriptors assume that the robot can be represented as a set of rigid links connected by a hierarchical tree of joints. Each link has a single parent joint and a joint shared with a child link. There exists at least one single root link with no parent whose position and orientation are defined with respect to the world frame of reference. For simple non-humanoid robots, this may be the only link of the robot. Similar hierarchical robot representations have been adopted by the Open HRP humanoid robotics software platforms [11] and the H-ANIM humanoid model specifications [12]. This representation is sufficient to describe a wide variety of branched, articulated chain mechanisms.

Our robot descriptors consist of *links*, *joints* and *control frames* (see Figure 2). Each of these entities can be referenced by a name, allowing intuitive labels or common naming schemes to be used (eg., leftArm, torso, rightLeg, head). Links contain important parameters such as the mass, center of mass, and inertia tensors of the body segment. These mass parameters are required for dynamic simulation and can be useful for the design of appropriate controllers.

A joint stores the local rigid body transformation of a link with respect to its parent. The transformation is defined by a combination of constant parameters and variable degrees of freedom (dofs) of the joint. The constant portion consists of the joint's position described in both the parent's (inboard) and child's (outboard) body-fixed coordinate system with respect to each link's center of mass (refer to Figure 2). In addition, we specify a constant *rest matrix* representing the joint's local transformation when all its dofs are zero. This defines the robot configuration in its rest (or home) state.

The dofs of the joint make up the kinematic state of the robot's configuration with a maximum of six dofs per joint for a fully unconstrained rigid transformation on a link. Each degree of freedom is specified as being either prismatic or revolute with a related axis vector, allowing any combination of up to six of these transformations to produce the most common robotic joint types such as pin, universal, gimbal and prismatic joints. Joint limits defined for each dof can further constrain movement.

The last descriptor is the control frame which has a position and orientation in the local frame of its parent link. In contrast to joints, a control frame is fixed to the link and does not exhibit independent motions from the link. The primary use of control frames is to specify the location and orientation of end-effectors or task frames used for manipulation. Control frames provide an important interface between the robot's generalized coordinates and the task at hand.

C. Robot Commands and Communication Protocol

The command specification defines the format of robot commands and their arguments. Command arguments are always expressed in MKS units when they represent physical quantities. Controlling a robot joint depends (of course) on whether an actuator is present at the joint, and the type of servo command (set-point, speed or force) accepted by an actuator is a function of its implementation. RoboTalk associates a servo-command type to every joint and control frame to indicate the motion commands supported. A joint or control-frame may support more than one type of servo command.

We defined seven types of signals to return function command status. There are three types of error signals: PANIC, ERROR and INTERRUPTED. The signal INTERRUPTED occurs when a legitimate action is halted prematurely due to an error. The other four signals are: BUSY, SUCCESS, MODIFIED and NA. Signal BUSY is returned if the robot is yet to finish a previous request that cannot be interrupted. SUCCESS is returned to indicate successful completion. If the driver has limited support of a command, it will return MODIFIED (e.g., a holonomic command executing in a non-holonomic robot). However, signal NA is returned if the robot does not support this command type (e.g., leg commands sent to a Pioneer 2 robot).

The communication protocol describes the format to send multiple commands in one network package. This protocol sits above conventional network protocols, such as TCP, UDP, DDS (data-distribution service), etc. RoboTalk's protocol supports four command modes: direct, delay, playback, and broadcast. Each package contains a fixed-length header and a payload. The header contains a panic flag, the type of the payload, and the size of the payload.

When the panic flag is set the server automatically puts the robot in panic condition (if supported by the hardware). Panic conditions are set by the package header, not its payload. The reason behind this will be clear in the next section.

Network packages contains one of two types of payload. These are either queries about the robot's status, or commands to move the robot. We restrict payloads to be either type, but not both: i.e., payload types cannot be mixed. This allows us to easily distinguish "read" from "write" packages.

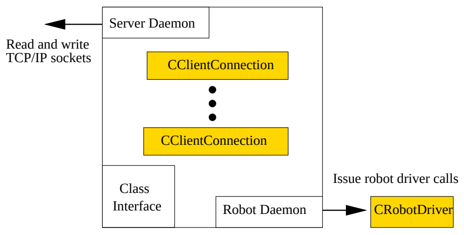


Fig. 3. CServer class structure.

The values within a package payload must be stored in network-byte-order integer format to ensure compatibility across systems. This is to deal with byte ordering differences among little-endian and big-endian processors. Thus, command parameters and other configuration descriptors are expressed in integers. Our client and server libraries convert the MKS units into integer representations.

IV. RoboTalk Implementation

Our first implementation for both the server and client followed the object-oriented paradigm and was written in C++ language. Thus, we use C++ examples during our discussion. Nevertheless, RoboTalk has no language requirement. Yet, the implementation must follow the specs and support RoboTalk's command modes (explained in the next section).

A. Server Implementation

Figure 3 illustrates the structure of the basic C++ server class: CServer. Within CServer, a second class – CClientConnection– manages the services to individual client connections, such as network I/O buffering, command mode scheduling, panic detection, etc.

CClientConnection class provides methods to buffer and encode/decode the packages for one client connection. A CClientConnection array extends these methods to multiple clients. Two independent threads encode/decode packages and process the buffers in round robin over all client instances: the ServerDaemon thread and the RobotDaemon thread.

ServerDaemon is responsible for adding/deleting client instances. It calls the ::Read() function to read a package sent by a client, and parses the commands contained in the payload. These commands are then processed by RobotDaemon.

Commands are not executed in their arrival order: instead, commands are stored in a *command queue* and scheduled according to their starting time. Thus, a STOP command scheduled to start in 40 sec. will be executed by the RobotDaemon after a GOTO command scheduled to start in 20 sec., even if the STOP message arrives before the GOTO. That is, the starting time of a command constitutes its reverse priority. A command with priority of 0 will be executed immediately. Commands with same priority are executed in their arrival order.

Query commands always have priority 0, but motion commands usually do not. A set of motion commands can be

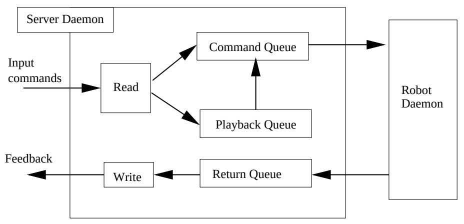


Fig. 4. CClientConnection class structure.

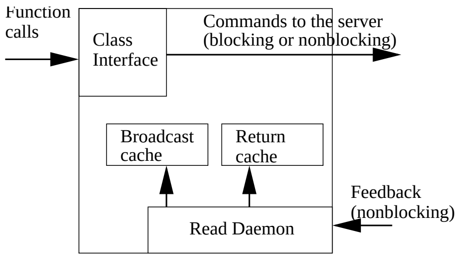


Fig. 5. CRobotClient class structure.

queued, resulting in better motion execution in the presence of network delays. PANIC signals, however, are never stored in the command queue, and are instead executed immediately by ServerDaemon. Commands from other clients are blocked until the panic signal is reset.

Return signals generated by robot commands are placed in a *return queue* by the RobotDaemon, and scheduled according to their issuance time. This queue is processed by the ServerDaemon in the same way the RobotDaemon processes the command queue, calling the ::Write() function to send replies back to the client. A third queue, the *playback queue*, is used to play a motion sequence of finite duration. This will be explained in Section V.

There is one CClientConnection instance for each client connection. Therefore, each a client has its own set of command, playback and return queues.

B. Client Implementation

The structure for a client is relatively simple. Figure 5 shows our implementation of the CRobotClient class.

CRobotClient sends network packages to the server in either blocking or non-blocking mode depending on the command mode selected (see next section). In blocking mode, the client execution stops when a command is issued to the server, and execution resumes when the client receives an acknowledgment. Execution is not interrupted in non-blocking mode.

Motion commands are executed by the server based on their priority, and commands in turn have different completion spans. Therefore, in non-blocking operation, we cannot assume that the server replies to the client in the same order as the original motion commands were sent. To address this problem, an independent thread –ReadDaemon– stores server replies in a *return cache* implemented as a hash map.

The hash map behaves as a dictionary data structure. The element keys are the original command ID's for which the server generated a response. Thus, in non-blocking mode, the client can periodically query the return cache to verify if the response to a particular command has arrived. The hash map is implemented with the hash multi-map defined in the C++ Standard Template Library.

V. COMMAND MODES

In this section, we describe the four command modes supported by RoboTalk. These modes are implemented by the three priority queues on the server's end for each client connection, and the hash map on the client's end.

A. Direct Mode

Direct mode operates as follows: 1) The network communication between a client and a server runs on blocking mode; 2) if the robot is not in Panic state, the server processes all control and query commands immediately after these are received; and 3) a client function call returns only after the client receives an execution acknowledgement. A command may optionally delay its execution with a non-zero starting time.

As shown in Figure 6, the first step is to synchronize the clock between the client and the server. This synchronization occurs immediately after the connection between client and server is first established.

Afterward, the client starts to accept calls to control the robot. When a function is called, the client creates a network package containing (possibly) multiple commands to be executed by the robot. Commands are tagged, and the whole package is time-stamped and sent to the server. The client execution halts inside the function call until a result package for this function arrives to client's return cache.[1]

On the server side, the package is placed into the command cache. The RobotDaemon thread continuously checks if the timestamp of the command with highest priority has expired. Moreover, all control commands with priority 0 will be executed by the RobotDaemon immediately after they are received by the server. Thus, the transmission and execution of queries and other priority-0 commands follow a *hand-shake* model while in direct mode.

For Panic signals, a special flag is set in the network package header. This flag indicates that the command is not to be stored in the command cache. Instead, the robot panic state is activated as soon as the package header is parsed. All caches are flushed, and the server rejects any new commands until the panic flag is reset.

B. Delay Mode

Delay mode specifies that: 1) The network communication runs on TCP non-blocking mode; 2) function calls return immediately after a command package is sent; 3) the server uses buffering to compensate possible network congestion; and 4) the amount of buffering is specified by the client as a time delay in the execution of commands.

Figure 7 shows the implementation diagram of delay mode. This diagram is similar to the one describing direct mode,

[1]A time-out mechanism can be added to deal with communication breakdowns or server crashes.

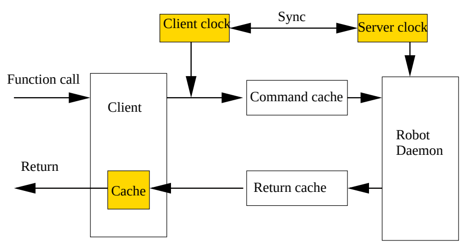


Fig. 6. Direct mode diagram.

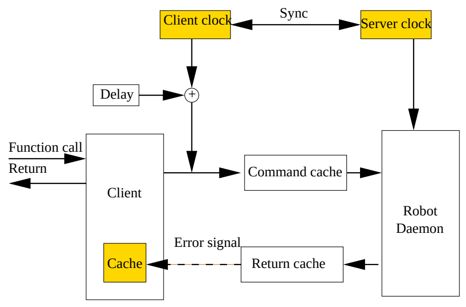


Fig. 7. Delay mode diagram.

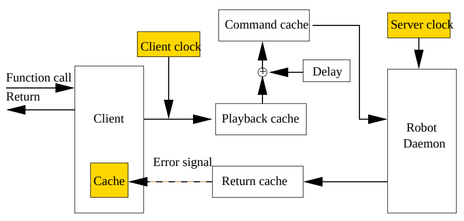


Fig. 8. Playback mode diagram.

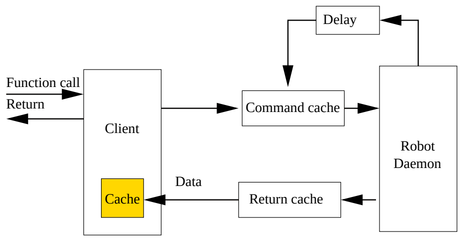


Fig. 9. Broadcast mode diagram.

except for an additional mechanism that shifts the client's clock forward. That is, the client synchronizes its clock to the server's local time plus the specified time delay. The delay is henceforth included in all packages.

On the server side, the contents of the command cache have a time offset with respect to the server's clock. The RobotDaemon will not execute the client commands until the time delay elapses, and these are instead queued in the command cache. The client can thus specify a long delay in order to play a difficult motion sequence through a congested network connection. The sequence is delayed, but its inter-frame timings are preserved because the command cache is acting a buffer.

It is possible to run delay mode with a time delay of 0. This looks similar to direct mode, but client-server transactions do not obey the hand-shake model.

C. Playback Mode

Playback mode is an improvement over delay mode when a motion sequence is finite. The server automatically decides how much to delay the sequence based on the observed network congestion.

It is not necessary to synchronize the server and client clocks because only the inter-frame timings are important in this mode. Instead, playback mode requires that the duration of the motion sequence is known to the client, and that motion frames are (roughly) evenly-spaced in time.

The server stores the received packages into the playback cache (another priority queue). The server calculates the ratio of the average time between commands to the average time between package arrivals after enough packages are stored in the playback cache. This ratio is used to estimate the time delay required by the playback sequence.

After the server estimates the necessary time delay, it adds this value to the package timestamps, and shifts all stored packages and future packages back into the command cache. The motion sequence is thus delayed and buffered by the server. Meanwhile, the client runs in non-blocking mode.

D. Broadcast Mode

Broadcast mode is designed to periodically query the robot state with a single request. The server handles broadcasting commands with a special query that periodically re-inserts a delayed copy of itself into the command cache.

Figure 9 illustrates the mechanism behind broadcast mode. The client issues a broadcast request with a specified sampling time. A query about the robot state is inserted into the command cache. Once this query is executed the state information is sent to the client's return cache. However, a copy of the query is created and reinserted into the command cache with the sampling time added to its priority. Thus, the query command regenerates itself for future execution by RobotDaemon without the client repeating the query. This regeneration stops once the client issues a broadcast cancellation. The client, of course, runs in non-blocking mode.

Table I shows the summary of the four command modes.

VI. Experiments

This section explains our experiments using RoboTalk with a Pioneer 2 robot and a Honda ASIMO humanoid. Several video demo sequences are available to our readers at *http://perception.csl.uiuc.edu/demos/RoboTalk/* .

A Sony EVI camera was mounted on top the Pioneer to act as a head. This camera has motorized pan and tilt actions. The Pioneer base became link 0 in our specification, while the camera became link 1. The links are connected with a single 2-dof revolute (actuated) joint.

We implemented CServer for the Pioneer 2 and ASIMO using POSIX threads and socket protocols under Linux and

TABLE I

Mode	Communications	Syncronization	Returns	Advantages	Limits
Direct	blocking	necessary	acknowledge every commands	good for debugging	does not preserve inter-frame dynamics
Delay	nonblocking	necessary	error signals	preserve sequence within a time frame	not adaptive to the channel condition
Playback	nonblocking	unnecessary	error signals	preserve the whole sequence	need to know the sequence length in advance.
Broadcast	nonblocking or broadcasting	unnecessary	broadcasted data	for repeated data query	N/A

Cygwin for Windows XP. In the Pioneer 2 case, the server interfaces with the motors using Player's P2OS position driver. [2] The camera is likewise controlled through Player's PTZ driver. In other words, CServer was compiled to use Player as the robot driver. It is important to note that RoboTalk's uses no other feature from Player other than the P2OS and PTZ drivers. This shows how RoboTalk can be integrated with an existing robot architecture.

Both ASIMO and the Pioneer robot are equipped with wireless network connectivity (802.11b). In the former case, CServer resides in a computer connected to the robot on an exclusive wireless channel. In the latter case, CServer and Player are installed in a laptop on board the Pioneer 2, and connected to the motors through its serial ports.

A. Network and Robot Compatibility

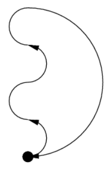
Clients were written for Linux, Mac OSX, and Cygwin for Windows XP. This OS mixture allowed us to verify network compatibility between architectures with different byte orders. For example, a PowerPC CPU has big-endian byte order, while an Intel CPU follows little-endian byte ordering.

To verify robot compatibility, we developed a GUI-driven test application using the Qt API. The test consisted of transmitting an identical motion sequence to both ASIMO and the Pioneer 2 robot using the same client. Only the server network address changes. In other words, the robots' internals are completely abstracted from the client.

The sequence is a series of forward/backward and turning motions for the main body, coupled with panning motions for the head. Both ASIMO and the Pioneer[3] are mechanically capable to reproduce this sequence, although ASIMO follows the motion by walking. The test sequence executes correctly in both robots (Figure 10). Videos are available on our website.

B. Network Robustness

To test RoboTalk's network robustness, we simulated the effect of traffic congestion. We first connected our Pioneer CServer to Stage, the companion simulator to Player. Next, we designed a velocity profile using SETSPEED instructions generated every 0.1s by the client. The resulting path consists of a sequence of four small half circles, followed by a large half circle, such that the robot returns to the starting point after the motion. Under perfect timing, the simulator on the server side should generate the following path:



The shape of the generated path will look very different if the timings are not preserved. Thus, we can judge the robustness of RoboTalk's four different command modes from the overall distortion of their paths.

Random network congestion is simulated by adding a SLEEP subroutine in the ServerDaemon thread on the server side. This does not change the timestamps of the commands in the sequence (as these are generated by the client). But SLEEP halts ServerDaemon for a random duration of $[0, 0.1]$s after the TCP POLL signal is detected. This delay occurs before the TCP package is cached to the corresponding CClientConnection instance. Since the command modes are implemented in separate CClientConnection instances, this add-in delay affects the communication channel to that client only.

The test results show the advantages of the command mode design. The effect of congestion is shown in Figure 11 for different command modes (videos are available on our website). Without congestion, direct mode replays the ground truth exactly, as shown in (a). Delay and playback modes also replay the path exactly under this condition, and are thus omitted from the figure. With simulated congestion, however, the path generated by direct mode is severely distorted (b), while delay mode with a 10s buffer only partially preserves the shape of the path (c) —just the first 10 seconds of motion are replayed correctly. Yet playback mode executes the path correctly, regardless of congestion (d). But with a cost: the motion will not start playing immediately. Delay and playback modes add latency to the overall sequence, direct mode does not (which is why direct mode cannot be replaced in every application). This effect is demonstrated in our video files.

[2]We modified this driver to accept position commands in addition to speed control.

[3]The reader may observe in the video that the Pioneer 2 appears to move abruptly. This is because the motor speeds and accelerations are set very high. The robot position control is acting as a bang-bang controller.

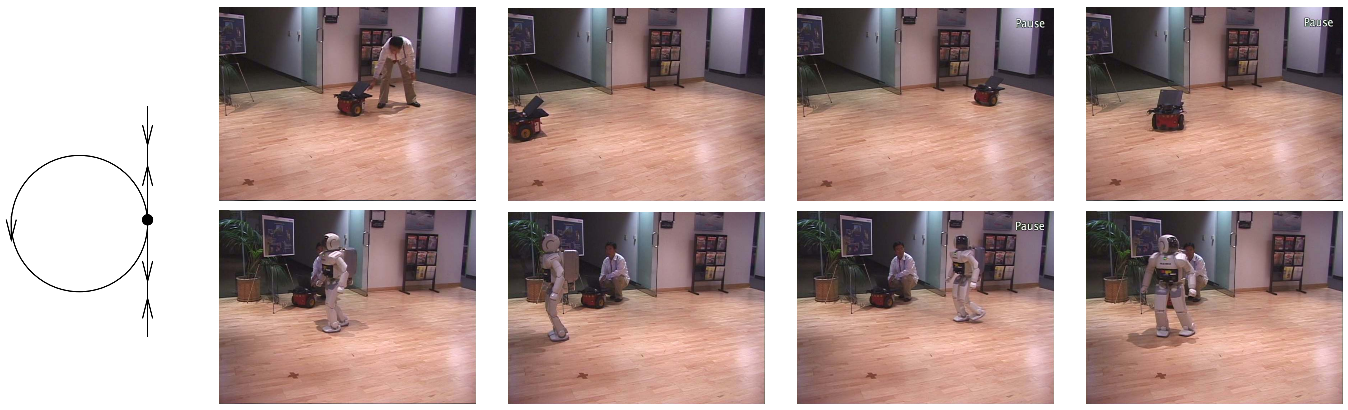


Fig. 10. Snapshots of a motion sequence using GOTO commands, executed on a Pioneer 2 (top row) and Honda's ASIMO (bottom row). The target path is shown on the left. The motion was coupled with pan and tilt actions for joint #1 (the head). We intentionally defined the pin-tilt joint of the camera on Pioneer 2 as joint #1. The client sent the same sequence without knowing the driver implementation.

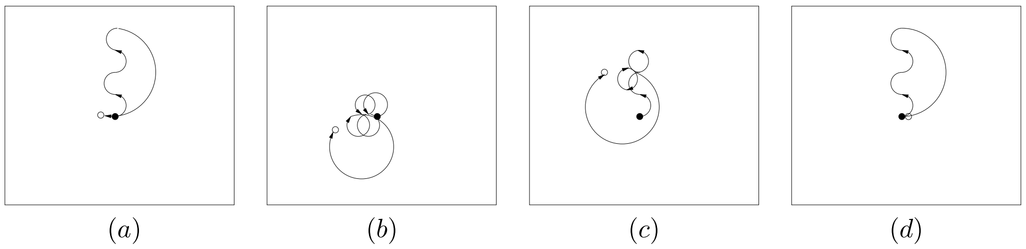


Fig. 11. Four trajectories from the original sequence. (a) Direct mode without network congestion. (b) Direct mode with network congestion. (c) Delay mode with network congestion. (d) Playback mode with network congestion.

VII. Final Discussion

RoboTalk provides an abstraction layer that allows developers to develop systems for motion planning and control without knowledge about the robot's internals. This minimizes the amount of information to be disseminated over a development group. RoboTalk also allows distributed operations, where an assortment of remote processes operate different robot subsystems. This suggests the creation of new types of robots whose physical parts are geographically dispersed.

An important contribution of our work is the provision of four communication modes that acknowledge the presence of network traffic congestion. The use of buffering and monitoring of timestamps allow motions to be enacted in the correct sequence with smooth performance playback. Provisions are made for assigning higher priority to different messages, such as the PANIC signal.

Except for kinodynamic variables, RoboTalk does not address the issue of sensing. Sensing remains beyond the scope of the original RoboTalk's specification. In future work, we will deal with sensors with a parallel interface specifically designed for sensors.

We are currently developing RoboTalk servers to work with a human kinodynamic simulator produced at Honda. In particular, the control frame specification in Section III-B will be used to carryout end-effector position tasks in a virtual environment. These simulators typically have more dofs than current humanoid robots, but otherwise use similar hierarchical models. The motion interface will be used to connect an application to different simulators for testing, and seamlessly switch devices during operation.

References

[1] R. Hirose and T. Takenaka, "Development of the humanoid robot ASIMO," *Honda R&D Technical Review*, vol. 13, no. 1, April 2001.

[2] M. Fujita, Y. Kuroki, T. Ishida, and T. Doi, "A small humanoid robot sdr-4x for entertainment applications," in *International Conference on Advanced Intelligent Mechatronics*, vol. 2, July 2003, pp. 938–943.

[3] F. Yamasaki, T. Matsui, T. Miyashita, and H. Kitano, "Pino the humanoid: A basic architecture," in *Proc. of the Fourth International Workshop on RoboCup*, August 31, Melbourne, Australia 2000.

[4] B. Gerkey, R. Vaughan, K. Støy, A. Howard, G. Sukhatme, and M. Matarić, "Most valuable player: A robot device server for distributed control," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2001)*, 2001, pp. 1226–1231.

[5] R. V. B. Gerkey and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of IEEE Int. Conference on Advanced Robotics*, 2003, pp. 317–323.

[6] H. Aldridge, W. Bluethmann, R. Ambrose, and M. Diftler, "Control architecture for the robonaut space humanoid," in *IEEE-RAS Humanoids 2000 Conference*, September 2000, boston, MA.

[7] J. Biesiadecki, M. Maimone, and J. Morrison, "The athena sdm rover: a testbed for mars rover mobility," in *6th International Symposium on AI, Robotics and Automation in Space (ISAIRAS-01)*, 2001, montreal, Canada.

[8] ActivMedia Robotics. ARIA. [Online]. Available: http://robots.activmedia.com/ARIA/

[9] The OROCOS Project. Open real-time control software. [Online]. Available: http://www.orocos.org/index.php

[10] The Open Automation Project. OAP. [Online]. Available: http://oap.sourceforge.net/

[11] F. Kanehiro, H. Hirukawa, and S. Kajita, "Openhrp: Open architecture humanoid robotics platform," *The International Journal of Robotics Research*, vol. 23, no. 2, pp. 155–165, 2004.

[12] Humanoid Animation Working Group, "H-anim 1.1 specification for a standard humanoid," www.h-anim.org, 1999.