Behavior Research Methods
2010, 42 (4), 1059-1071
doi:10.3758/BRM.42.4.1059

Whisker: A client–server high-performance multimedia research control system

Rudolf N. Cardinal and Michael R. F. Aitken
University of Cambridge, Cambridge, England

We describe an original client–server approach to behavioral research control and the Whisker system, a specific implementation of this design. The server process controls several types of hardware, including digital input/output devices, multiple graphical monitors and touchscreens, keyboards, mice, and sound cards. It provides a way to access this hardware for client programs, communicating with them via a simple text-based network protocol based on the standard Internet protocol. Clients to implement behavioral tasks may be written in any network-capable programming language. Applications to date have been in experimental psychology and behavioral and cognitive neuroscience, using rodents, humans, nonhuman primates, dogs, pigs, and birds. This system is flexible and reliable, although there are potential disadvantages in terms of complexity. Its design, features, and performance are described.

We describe a new method of implementing software for behavioral research control, using a client–server mechanism. The prototypical problem addressed by this software is as follows. An experimenter has one or more computers, each attached to multiple computer-controlled operant chambers, used for experimental psychology or behavioral/cognitive neuroscience research with experimental animals. Operant conditioning chambers (Skinner, 1938) are chambers that enclose an experimental subject and have, at least, some form of manipulandum, such as a lever, and a mechanism for producing reinforcement, such as a pellet dispenser. Generally, they also have forms of stimulus delivery equipment such as lights, sound generators, and visual displays (which may have touchscreen equipment attached). The experimenter wishes to run a variety of behavioral tasks using this equipment (see Figure 1).

The software components of typical computer-based control systems for behavioral research are organized along two dominant themes. Both may be characterized in computing science terms as monolithic (i.e., not based on a client–server architecture). Some provide extensions to an existing programming language (Fray, 1988, 1990, 1993); these extensions provide access to hardware control routines, timing facilities, and so forth, and typically they control a single type of digital input/output (I/O) interface. Users may program freely in the base language, including all functions for data handling. Each behavioral task thus communicates directly with the I/O hardware. If the host language or computer does not offer multitasking facilities, control of multiple devices or tasks is the responsibility of the user.

Other systems provide a custom language (Campden, 2005; Dixon, 2009; Fray, 1980; Lafayette, 2007; Med Associates, 2004; Palya & Walter, 1993; Panlab, 2004; Tatham & Zurn, 1989), which may be procedural, graphical, or state-based. The custom language may be interpreted directly, or translated internally into a general-purpose computing language, such as Pascal (Wirth, 1971), and executed by the control system's primary process. The latter approach typically allows users to call any other code that can be linked from the general-purpose language (Tatham & Zurn, 1989). Such custom language control systems may offer predefined data-recording mechanisms (Campden, 2005; Lafayette, 2007; Tatham & Zurn, 1989).

We suggest that a good general-purpose approach to the implementation of a control system for behavioral research entails the separation of functions between a single server process and multiple client processes. (We refer throughout to a "process," meaning a program running on a computer; the single server process and the client processes typically, although not necessarily, run on the same physical computer.) The server process has responsibility for all communication with the physical devices being controlled, the management and allocation of these devices as resources for client processes, and temporal control (see Figure 2). Each of the client processes may implement a specific behavioral task and control the required resources independent of other clients.

An additional set of practical problems concerns the range of hardware that can be controlled by the behavioral control system. For example, the range of digital I/O interfaces supported by many existing operant control systems (see above) is limited. Additionally, the use of complex audiovisual stimuli for animal testing has increased recently in the fields of experimental psychology and be-

R. N. Cardinal, rudolf.cardinal@pobox.com

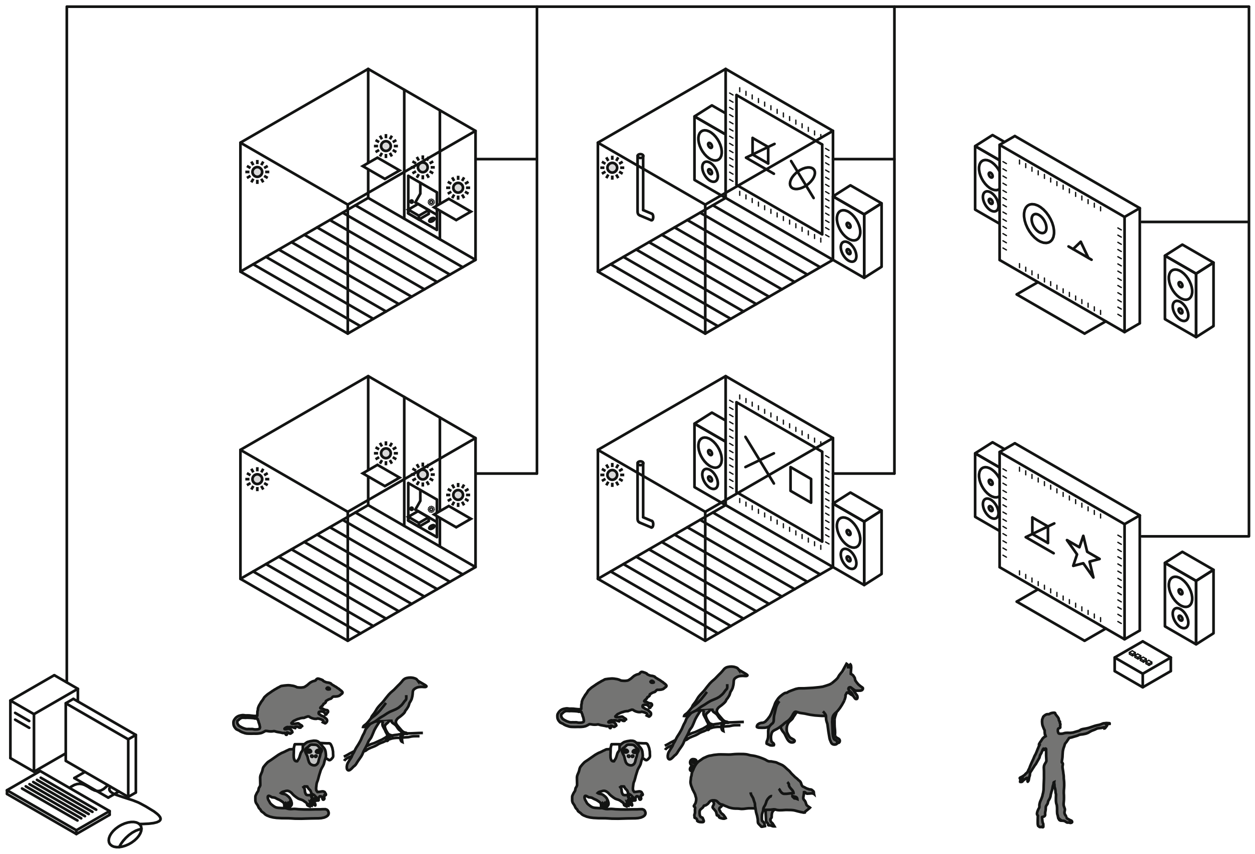


Figure 1. Prototypical hardware that might be controlled by Whisker. A single computer (bottom left) controls multiple behavioral test stations. Several types of test station are shown, and may be mixed freely, running independent behavioral tasks that communicate with the hardware through the Whisker server process. On the left, standard operant chambers are shown, such as those typically used for rodents, birds, and primates. These are depicted here with a grill floor, several stimulus lights, two retractable levers, and between them a central food alcove fitted with a pellet dispenser delivering pellets into a tray, a retractable liquid dipper that rises through a hole in the alcove floor, an alcove light, and an infrared nosepoke detector. In the middle, touchscreen chambers are shown. Those illustrated are fitted with a liquid crystal display screen for presentation of arbitrary visual stimuli, with an infrared touch-detection array attached to it, a ceiling-mounted licker served by a pump for dispensing liquid reward, and loudspeakers. These chambers are suitable as depicted or with minor structural modifications for a wide variety of species, such as rodents, birds, dogs, primates, and pigs. On the right, plain displays are shown, suitable for human testing, with touch-detection equipment and/or response keys (button boxes). The devices being controlled comprise digital output devices that may be on or off (e.g., lights, lever motors, pellet dispensers), digital input devices that may be on or off (e.g., levers, nosepoke detectors, response keys), visual displays, touch detectors, and sound devices.

havioral neuroscience (e.g., Bussey et al., 2008; Fray & Robbins, 1996), yet few control systems provide facilities to control graphical displays (monitors) and touchscreens. In practice, touchscreen control is often accomplished by custom-written touchscreen extensions to generic languages that already have digital I/O control extensions (Parkinson et al., 2001), or by programming all the control systems *de novo* (Bussey, Saksida, & Rothblat, 2001; Gibson, Wasserman, Frei, & Miller, 2004; Markham, Butt, & Dougher, 1996).

We describe the implementation and performance of a client–server design, in which communication is via simple text-based messages passed between client and server processes, which usually run on the same computer (see Figure 2). This method makes use of the transmission control protocol/Internet protocol (TCP/IP) suite's standard facility for a virtual and very fast local network operating between programs running on the same computer (Cerf, Dalal, & Sunshine, 1974; Cerf & Kahn, 1974; Postel, 1981). The hardware controlled by the server process includes monitors, sound devices, and touchscreens, as well as conventional digital I/O interfaces from a range of manufacturers. Our implementation has been used for both animal experimentation and human testing. The use of audiovisual stimuli, touchscreen responding, and digital I/O is common to many species including humans, and human-specific devices (keyboards, mice) are also supported; indeed, many key tasks in modern behavioral neuroscience are explicitly designed to be used across species barriers—for example, in monkeys and humans (Fray & Robbins, 1996). Any experiment using any of the device types supported and requiring a temporal resolution for digital I/O and behavioral timing of no more than 1 kHz may be performed using this system. Not all experimenters would want to use a client–server design, however; we

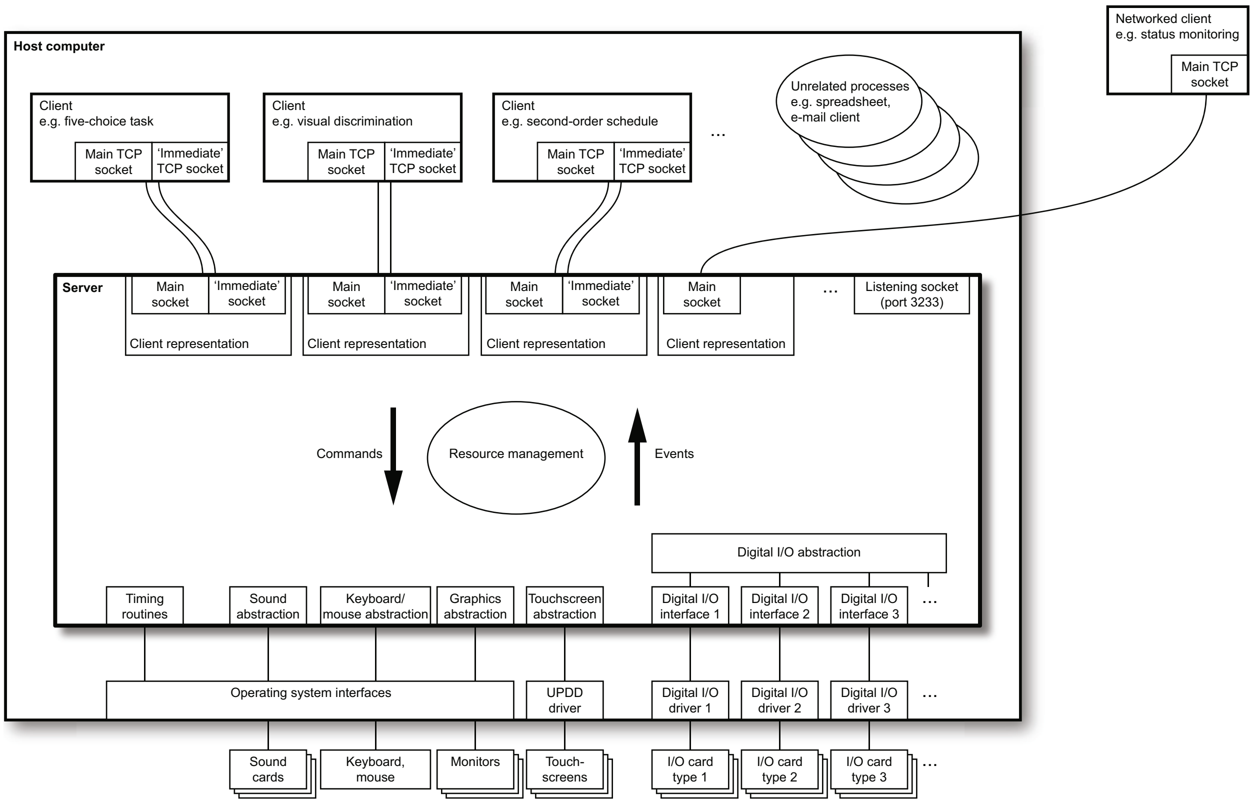


Figure 2. Outline of the software architecture, showing communication routes between a single server program (process) and multiple client processes running on the same computer and on a different computer (typically via two bidirectional transmission control protocol [TCP] sockets per client), between the server program and physical devices that it controls, and between representations internal to the server. The thick black box represents the server process, and the surrounding box represents the host computer. Client programs typically run within the same computer as the server process; three examples are shown (boxes at top left), each linked to the server process via two TCP network sockets. They communicate with the server process to request resources, set the state of digital output devices, display visual stimuli, play audio files, and so on. The server process sends event messages to the clients corresponding to changes in the state of the physical hardware, responses made by subjects, timing events, and so forth. Each client typically requests resources from the server that correspond to one physical operant chamber. Other programs (shown as ellipses at top right) may also run on the computer, independent of Whisker and its clients. The Whisker server listens for new connections from clients on a "listening socket" on port 3233 (depicted at top right of the server process). Connections can also be made from clients on other computers, across a local area network or wide area network (small box at far top right); typically, this facility is used to monitor the status of Whisker and its clients remotely. Beneath the box representing the server process, communications with the operating system and device drivers are shown, and beneath them, in turn, the physical control hardware is depicted. Whisker communicates with sound cards, the keyboard and mouse, and multiple monitors via Windows operating system interfaces; it communicates with multiple touchscreens via the Universal Pointing Device Driver (UPDD) system (see text); and it communicates with as many digital input/output (I/O) cards as are installed via their manufacturer-specific drivers, while providing an internal abstraction of individual I/O lines with which the clients interact. Within the server process, internal representations of connected clients and of the hardware interact, with commands passing from the clients to the hardware, events passing from the hardware or the server process to the clients, and the server process providing resource contention management.

also discuss the practical advantages and disadvantages of this system and its likely research applications.

Motivation, Design Goals, and History

In 1999, our laboratory wished to purchase additional operant chambers for rat behavioral neuroscience work. The choice of digital I/O control hardware and associated software designed for use with operant chambers was small, the hardware was expensive, and the software was limited because the choice of system dictated the computer type, the operating system, and the programming language in which behavioral tasks could be written, with those programming environments being considerably restricted. However, cheap generic digital I/O interfaces and computers were available. Our prototypical experimental station was a single computer connected to 6–8 operant chambers using 1–2 digital I/O cards. The missing component was the software. Having embarked on the task of writing a new control system, we set out to solve the problems we had encountered with previous such systems.

First, we wanted to be able to run several different behavioral tasks simultaneously and asynchronously, one for each operant chamber, without one task interfering with another's operation, even if an individual task encountered a problem or software crash. Examples of actual behavioral tasks might include simple schedules of food reinforcement, a second-order schedule of intravenous cocaine reinforcement, a Pavlovian–instrumental transfer task, and choice involving delayed reinforcement.

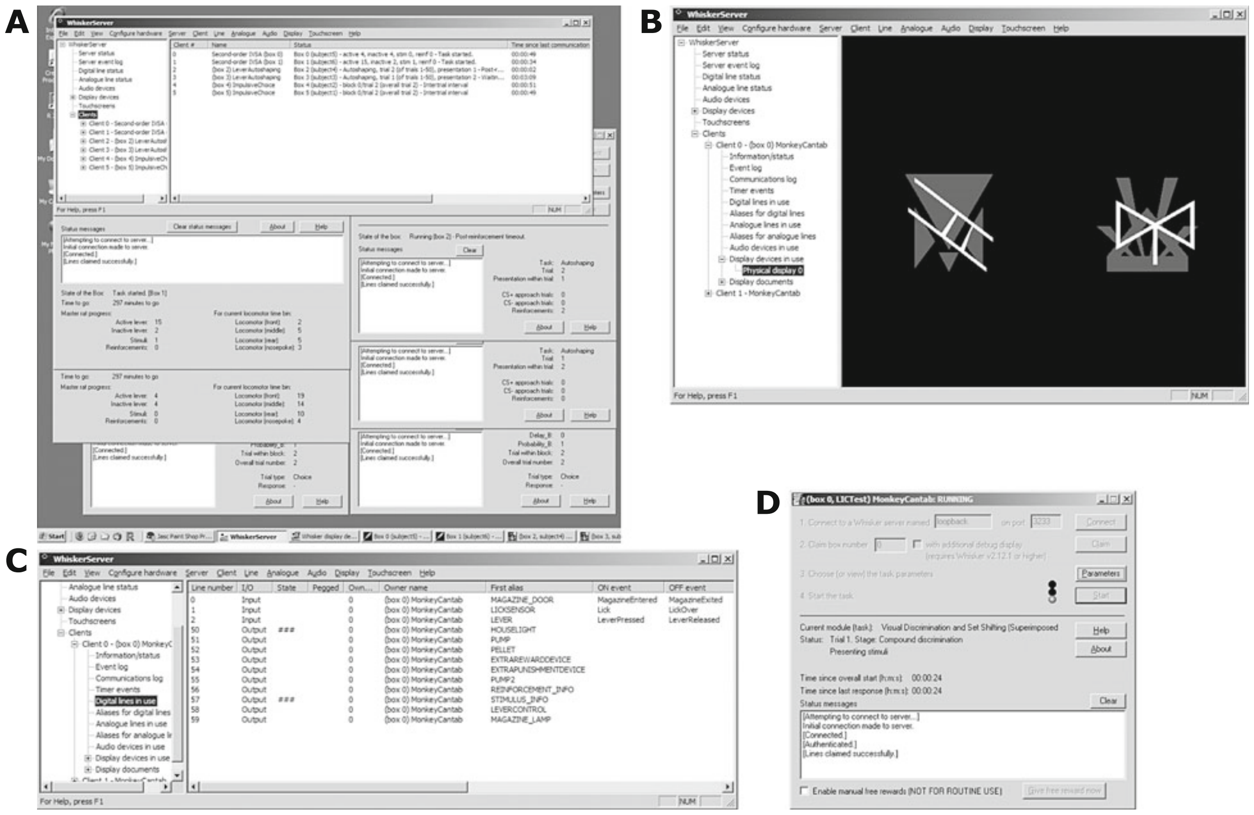
Second, we wanted to be able to use mainstream programming tools to write behavioral tasks, and to be able to use any programming language, so that we could add arbitrary features such as direct communication with databases and complex on-the-fly calculations to our tasks.

These two considerations dictated the choice of a preemptively multitasking operating system and, more importantly, a client–server model, which to our knowledge had not hitherto been used for behavioral research control. The natural method of implementing a client–server model is with the TCP/IP suite, which has been the dominant networking standard worldwide for decades. Therefore, we developed a system in which a single server program communicates directly with all relevant hardware devices (see Figure 2). Clients communicate via TCP/IP with the server process, which provides a simple interface with which to control the hardware.

Third, we wanted to be able to control many kinds of digital I/O devices, including equipment from multiple manufacturers, and to be able to extend the server software in the future to support additional hardware without any modification to the client software.

Fourth, we wanted to create a layer of abstraction by the use of arbitrary device names, so that a task could be written for generically named devices (e.g., "lever") and thus be directly portable across different computers, regardless of whether the lever in question is connected to the first or fiftieth logical line on a given computer.

Fifth, we wanted debugging facilities, such as the ability to inspect the operation of tasks as they were running (see Figure 3), to manipulate the hardware directly for testing purposes, and to create "fake" or virtual devices,


A
B
C
D
WhiskerServer
File Edit View Configure hardware Server Client Line Analogue Audio Display Touchscreen Help
For Help, press F1
NUM

Line number	I/O	State	Pegged	Own...	Owner name	First alias	ON event	OFF event
0	Input			0	(box 0) MonkeyCantab	MAGAZINE_DOOR	MagazineEntered	MagazineExited
1	Input			0	(box 0) MonkeyCantab	LICKSENSOR	Lick	LickOver
2	Input			0	(box 0) MonkeyCantab	LEVER	LeverPressed	LeverReleased
50	Output	###		0	(box 0) MonkeyCantab	HOUSELIGHT		
51	Output			0	(box 0) MonkeyCantab	PUMP		
52	Output			0	(box 0) MonkeyCantab	PELLET		
53	Output			0	(box 0) MonkeyCantab	EXTRAREWARDDEVICE		
54	Output			0	(box 0) MonkeyCantab	EXTRAPUNISHMENTDEVICE		
55	Output			0	(box 0) MonkeyCantab	PUMP2		
56	Output			0	(box 0) MonkeyCantab	REINFORCEMENT_INFO		
57	Output	###		0	(box 0) MonkeyCantab	STIMULUS_INFO		
58	Output			0	(box 0) MonkeyCantab	LEVERCONTROL		
59	Output			0	(box 0) MonkeyCantab	MAGAZINE_LAMP		

(box 0, LICTest) MonkeyCantab: RUNNING

1. Connect to a Whisker server named loopback on port 3233 Connect
2. Claim box number 0 with additional debug display (requires Whisker v2.12.1 or higher) Claim
3. Choose (or view) the task parameters Parameters
4. Start the task Start

Current module (task): Visual Discrimination and Set Shifting (Superimposed
Status: Trial 1. Stage: Compound discrimination
Presenting stimuli

Time since overall start (h:m:s): 00:00:24
Time since last response (h:m:s): 00:00:24
Status messages

```
[Attempting to connect to server...]
Initial connection made to server.
[Connected.]
[Authenticated.]
[Lines claimed successfully.]
```

Enable manual free rewards (NOT FOR ROUTINE USE) Give free reward now

Figure 3. Illustrative screenshots. (A) Whole-screen view of the server process (wide window at the top of the screen) and six tasks connected to it (other six overlapping windows). Here, the clients are two copies of a second-order schedule of reinforcement, two copies of a delay-discounting task, and two copies of a lever-based autoshaping task. Each task controls a separate operant chamber connected to the server. The server process is shown in its summary status view. (B) View of the server process during execution of a visual discrimination task from the MonkeyCantab suite, a Whisker client, showing the server's copy of one graphical display from a multimonitor-equipped computer. This is the view that would be seen by the experimenter on the computer's primary monitor. The subject sees an identical unadorned copy of this display on a secondary monitor located in a touchscreen-equipped operant chamber. (C) The server's view of the digital input/output (I/O) lines being controlled by the same task as in panel B, illustrating a selective view of the subset of lines in use by one client, with information including their on/off status and events attached to them. Other available views are listed on the left-hand side of the server's window, such as a log of communications between the client and the server, a list of timer events, and so on. (D) The client's user interface at the same moment (same task as in panels B and C); this interface is used to connect to the server (usually on the local, "localhost," or "loopback" network address, meaning the same computer), to set up the task and data storage parameters, and to start the task.

so that we could program tasks on computers without physical I/O hardware attached (saving experimental time on the laboratory control computers) and be confident that they would work the first time they went into live operation.

Sixth, we wanted to be able to use visual displays and touchscreens. Hitherto we had worked with such equipment using ad hoc single-tasking monolithic single-display programming methods, since no research control system supported visual displays and touchscreens in addition to digital I/O. We determined to make our system support multiple displays from a single computer, with multiple touchscreens and multiple sound cards (see Figure 2). In part, this was to maximize cost efficiency; for example, when one computer with a single I/O card can support enough I/O lines to drive multiple touchscreen-equipped operant chambers, it may be cheaper to add more displays and touchscreens to the same computer than to add more computers and I/O cards. In addition, we thought that the client–server architecture would naturally extend to make programming visual stimuli, touchscreen events, and sound trivially easy from the point of view of the clients.

The Whisker control suite was developed by R. N. Cardinal in 1999 in the Department of Experimental Psychology, University of Cambridge, and subsequently developed in the same location by R. N. Cardinal and M. R. F. Aitken. Most of the suite is written in C++ (Stroustrup, 1986); the server program and its accompanying tools run to approximately 60,000 lines of code. The compiled code includes third-party libraries for multithreading support, cryptography, random number generation for clients, and for interfacing with specific hardware. The suite was first used for “live” research in 2000 and has been distributed commercially since 2002 under licence from Cambridge University Technical Services Ltd, a subsidiary of the University of Cambridge. The current distributors are Campden Instruments Ltd (U.K.), and the Lafayette Instrument Company (USA). It is currently in use by several academic, commercial, and governmental institutions across the world (Cardinal & Aitken, 2008). The source code to both the server and client software is open to license holders. Full source is distributed on request to licensed users, with rights to modify, compile, and use the source code in all ways other than redistribution to unlicensed users.

Overview of the Hardware and Software Architecture

A prototypical hardware configuration is shown in Figure 1, with a single testing computer controlling multiple operant chambers, some equipped with visual displays, touchscreen sensors, and sound devices. The subjects in the operant chambers or in front of the displays might be humans, nonhuman primates, rodents, dogs, pigs, birds, or other species; this is a matter of experimenter choice and suitable equipment. A single copy of the Whisker server process runs on this computer and communicates with all the hardware. Several client programs run on the same computer, independently of each other, and each communicates with the server process. Each client implements a behavioral task. The software architecture is summarized above and in Figure 2, and is discussed further below.

Example of a Simple Multimodal Task

We begin with a simple example from a client’s perspective. This example presupposes that the server process has been configured for the appropriate hardware, and a suitable device definition file (described in detail later; see the Resource Management for Clients section) has been set up to identify the hardware by name to clients. Thus, in what follows, group names such as “box1” and individual device names such as “leftlevercontrol” and “sound” have been defined already and mapped to individual numbered I/O lines, audio devices, and so on. The following script shows how to use the Whisker command set (described in detail at www.whiskercontrol.com) to set up a touchscreen object that will produce an event when touched, and ultimately produce a noise, and to set up a lever that will report an event when pressed, and ultimately produce a food pellet. The script begins at the point at which the client has made a TCP/IP connection to the server process. The first step is to claim the devices. This script will claim the whole of the “box1” group, representing all devices connected to operant chamber number 1, so that no other client can use it, but then reclaim individual devices, to ensure that those devices are present and no errors are generated. Some aliases are assigned for convenience, and a sound is preloaded:

```
ClaimGroup box1
LineClaim box1 leftlevercontrol -alias
  levercontrol
LineClaim box1 leftleverreport -alias
  leverreport
LineClaim box1 pellet -alias
  pelletdispenser
AudioClaim box1 sound -alias speaker
AudioLoadSound speaker rewardsound
  C:\mysound.wav
DisplayClaim box1 lcddisplay -alias
  display
```

Now we can extend the lever and ask it to generate an event named “LeverPressed” when it is pressed:

```
LineSetState levercontrol on
LineSetEvent leverreport on LeverPressed
```

We can create a display document (described in more detail later), set its background to dark blue (red 0, green 0, blue 100, on a scale of 0–255), add a red rectangle named “rect1” extending from coordinates (100,100) to (600,600), relative to the origin at the top left of the screen, load a bitmap from disk and name it “bmp1,” placing it on top of the rectangle with its top left at coordinates (200,200), add suitable events named “ObjectTouched” to detect when either the rectangle or the bitmap is touched, and show the document on our display:

```
DisplayCreateDocument doc
DisplaySetBackgroundColour doc 0 0 100
DisplayAddObject doc rect1 rectangle 100
```

```
    100 600 600 -penstyle null -brushsolid
    255 0 0
  DisplayAddObject doc bmp1 bitmap 200 200
    c:\mybitmap.bmp
  DisplaySetEvent doc rect1 TouchDown
    ObjectTouched
  DisplaySetEvent doc bmp1 TouchDown
    ObjectTouched
  DisplayShowDocument display doc
```

Now the client can wait for the following message from the server process (via the main TCP/IP port, described below):

```
Event: ObjectTouched
```

and can respond by playing a sound:

```
AudioPlaySound speaker rewardsound
```

Likewise, the client can wait for a lever event from the server:

```
Event: LeverPressed
```

and can respond by dispensing a pellet. For illustration, we shall dispense the pellet with a 50-msec on–off pulse to the pellet dispenser's control line, achieved by a one-off (nonrepeated) timer:

```
LineSetState pelletdispenser on
TimerSetEvent 50 0 EndOfPelletPulse
```

The client should respond to the subsequent message from the server (50 msec later):

```
Event: EndOfPelletPulse
```

with

```
LineSetState pelletdispenser off
```

To add detection of keyboard "down" (key pressed) events via the "KeyEvent:" message, for human testing situations in which a single display is in use and the display window has the input focus, the client can issue the following command:

```
DisplayKeyboardEvents doc down
```

This simple sequence illustrates the basics upon which complex tasks may be built. The full command set is described online (see www.whiskercontrol.com).

The actual method of establishing TCP/IP communications with the server process depends on the client's programming language. We provide examples in Perl, Python, Microsoft Visual Basic, and C++ (see www.whiskercontrol.com).

Communications Architecture

The Whisker server process is the heart of the system, and runs on 32-bit Microsoft Windows operating systems (NT4, 2000, XP, Vista). A client process interfaces with the hardware by communicating with the Whisker server process, using standard TCP/IP operations, with a defined text-based language and command set. A running server process listens on TCP port 3233, which is registered for Whisker use (Internet Assigned Numbers Authority, 2008). The server also offers a second TCP port to all connecting clients, from the dynamic port range; the client typically chooses to connect to this second ("immediate") port as well as the main port, providing a two-port system of communication.

The normal way to run Whisker would be to run the server program and several client programs on the same computer (see Figure 2). All computers running standard Internet protocols have an "internal" network facility, whereby programs running on the same computer can communicate with each other via the network system. This is much faster than communication over a physical network. Programs that communicate via an internal network can communicate over a network between different computers without modification, simply by changing the network address they communicate with; this is useful for Whisker clients in certain situations, but the normal mode of operation would be for all programs to run on the same computer.

Messages and commands are sent as plain text. This is not the most efficient format—numerical encoding and compression would be more efficient—but it makes communication simple for the client, whatever its programming language, and network latency is low enough for this inefficiency to not be a problem (see below). Commands are separated by semicolon, linefeed, or carriage return characters. Parameters are separated by spaces; parameters that include spaces or semicolons may be encapsulated in double quotation marks. Messages may be split across multiple TCP packets silently by the network subsystem, so there is a mechanism for detecting and recombining split messages, by holding any message remnants in a queue until the rest of the message arrives.

Events are reported by the server process to the client via the main port (port 3233). These may reflect digital input state changes, touchscreen responses, or timing events. The server does not spontaneously send information to the client via the immediate port. The messages sent by the server to the client on the main port are strictly defined (see www.whiskercontrol.com).

Clients may issue commands to the server on either the main or the immediate port. For client commands issued on the main port, the server may return multiple result messages on that port. Client commands sent on the immediate port are always followed by a single, rigidly defined response message; thus, each "immediate" client command always receives a corresponding "reply" message. Client commands that request information are therefore usually issued on the immediate port.

The rationale for the two-port system is as follows. In an event-driven system, events may occur unpredictably and at any time (for example, in response to a subject pressing a lever). The server process must send the event to the client process immediately. Consequently, there must be a port on which the client is constantly listening, ready to process event messages. This is an example of asynchronous or nonblocking I/O (Stevens, 1998). At the same time, it must be possible for the client to send a command or request to the server, and to obtain a reply. This is an example of synchronous or blocking I/O: The client wishes to obtain a reply immediately—that is, to pause or "block" until the reply is obtained. The server should not send spontaneous events down a socket on which the client might be expecting a reply to its command, since this could confuse the client, and it is highly inconvenient for clients to have to use a general event-processing system to deal also with all possible replies to its commands. This is a general problem when events may be sent at unpredictable times through a single channel of communication. The solution to this problem adopted by Whisker is to have two channels, using the main socket for sending unpredictable events, while the immediate socket returns responses to commands in a rigidly defined manner, so that the client can wait for its reply. The optimal way for clients to behave is therefore to send commands and await their replies on the immediate port, and to expect spontaneous events on the main port. Within the server process itself, all communications are implemented via nonblocking sockets so that any communication failure cannot hold up processing.

To ensure network responsiveness, the server and client processes disable the Nagle algorithm (Nagle, 1984), an algorithm for delaying and concatenating small TCP network packets to improve efficiency by minimizing the proportion of traffic taken up by packet "header" information. Since the Whisker architecture relies on network timing, efficiency is sacrificed for speed. The algorithm is disabled by setting the TCP_NODELAY parameter in the network stack.

Network latency within a single computer is theoretically extremely small, and real-world performance is explicitly testable through the server. We tested a client–server–client round trip on a computer with a dual-core Intel E6600 2.4-GHz processor running Windows XP under the unusually demanding conditions of 10^4 back-to-back tests without pause. Each individual test consisted of the client sending a Ping command and awaiting a PingAcknowledged reply (for syntax, see www.whiskercontrol.com). The round trip took a mean of 132.3 μsec (standard deviation = 29.2, range = 93–377, median = 123). Mean one-way network latency, relevant to commands and events, was thus 0.066 msec, with standard deviation 0.015 msec. All events generated by the server, whether in response to timers, digital inputs changing state, or touchscreen activity, may additionally be time-stamped with the system clock time at the moment of first receipt by the server, maximizing accuracy.

Implications for Client Behavioral Task Programming

The use of a TCP- and text-based system means that behavioral tasks can be implemented within any programming language or third-party task design software capable of TCP/IP communication. The majority of tasks to date have been written in C++ and Microsoft Visual Basic, although Python and Perl are other obvious candidates and work well. This freedom of choice regarding programming language means that task design is unrestricted, and hardware control facilities provided by the Whisker server process may be combined with those of other programs (e.g., dedicated electrophysiology software, video presentation) if so desired. It is also possible, for example, to create translation software to migrate behavioral tasks originally implemented as scripts for another research control system. Others have developed graphical front-end clients to aid task writing (Campden/Lafayette, 2009). Practical advantages and disadvantages of this system are discussed further below.

Since behavioral tasks run as independent client processes, the system is protected from failures should an exception or error arise within an individual task. A Whisker server process typically controls several operant chambers, and allows arbitrary combinations of tasks to run simultaneously and asynchronously within these chambers: A reversal learning task in one operant chamber can be started and stopped independently of the autoshaping task in another.

Timing and Threading Architecture

The server process uses the Windows high-performance timing system to monitor hardware status with a poll frequency of 1 kHz. Use of the high-performance multimedia timing system rather than the basic Windows timer system is necessary to guarantee performance: The basic Windows timer system is unsuitable for reliable timing, since the generated messages have the lowest priority within Windows (e.g., dragging windows around on the desktop can suspend basic Windows timers), whereas the multimedia timers provide guaranteed performance. The server process can be configured to run with real-time priority to ensure that other processor-intensive applications have minimal impact on the server's performance as a result of competition for processor time. The polling performance is monitored live to allow performance to be assured. We tested performance on a 2.5-GHz Intel Pentium 4 motherboard running Windows 2000 and obtained a mean polling period of 999.6 μsec (standard deviation = 53.6, range = 53–1,129) across 10^4 consecutive polls.

Using the highest, real-time priority in Windows will allow the server process to deny processor time to other processes (including the client processes) that perform essential tasks. In order to ensure that the system remains usable, and that any client processes are run correctly, the server process performs internal monitoring to ensure that it can use no more than 50% of processor time when performing hardware polls. This is achieved by "yielding" at the start of a poll if the time since the end of the preceding poll has been less than 500 μsec. Yields may thus occur if the processor is heavily loaded, or if the Windows polling system polls the server faster than 1 kHz, as it sometimes does.

The server process provides millisecond-resolution timing to connected clients. A client may request a timer to be fired at defined times, lending temporal structure to the task being implemented as in any other behavioral control system. The precise resolution depends on the stability of the server's 1-kHz poll routine (measured standard deviation = 53.6 μsec, as above) and the latency of a within-computer network message (measured latency = 66 μsec, with standard deviation = 15 μsec, as above). Timer commands result in Event messages from the server process to the client process (for syntax, see www.whiskercontrol.com).

Internally, the server process is multithreaded, with dedicated threads to handle incoming communication from each client. This is necessary so that any potentially slow operation (such as loading a bitmap from disk) does not affect communications with other clients, which can proceed uninterrupted. A separate high-priority thread, called from the Windows high-performance multimedia timing system, polls the digital I/O hardware (see below) and client timers and sends event messages to the clients. Another thread receives touchscreen messages. A final thread handles all graphical systems and the graphical user interface (GUI).

The use of a multithreaded architecture gains substantial performance but brings a standard set of programming problems (Cohen & Woodring, 1998). Most concern the problem of two or more threads trying to access the same object or variable simultaneously. Without proper safeguarding, Thread A might read part of a data structure and begin to work using this information, at the same time that Thread B is writing to the data structure, leading to internal inconsistency and sometimes program failure. Data access must be made "atomic": Each item of nonconstant data (or set of interdependent data) that may be accessed by more than one thread must be protected by a system of "locks" or "mutex" (mutual exclusion) algorithms, to prevent uncontrolled access (Cohen & Woodring, 1998). The ownership of objects is also important to determine, since serious memory management problems can arise if objects are created by and used within one thread and destroyed by another.

Digital I/O Architecture

Digital I/O cards for binary (on–off) control and detection are supported from several manufacturers, including Amplicon, Advantech, ICS Advent (Kontron), Lafayette, and National Instruments. Currently, these span ISA, PCI, and USB interfaces. The popular Intel 82C55 generic I/O controller chip is also supported directly, and the server process allows conventional serial (RS-232, COM) ports to be used to provide four lines of input and two lines of output each. Additional custom support is provided for some manufacturers' control hardware, including devices that use internal multiplexing systems to increase the number of physical devices controlled by a given I/O card. Internally, the representation of a digital I/O device is abstracted, allowing support for additional manufacturers' cards to be added simply.

Clients may set the state of output lines, read the state of any line, and attach events to line on/off transitions. In general, digital I/O cards are actively polled at the server's primary event frequency (1 kHz), although some types of cards that implement different (e.g., interrupt-driven) event notification have also been integrated into the system. Arbitrary different types of I/O card may be mixed in one system, limited by physical space to connect them to the computer. The digital I/O command set is described online (see www.whiskercontrol.com).

In addition to attached physical I/O lines, the server may be configured with virtual or "fake" digital input and output lines. These allow the development and testing of behavioral task software on computers that do not have physical I/O hardware attached, or the use of client software preconfigured to use devices that are not needed. All digital lines, whether physical or virtual, may be monitored and controlled via the server's console, which we have found to provide a simple way of testing client task software.

Internally, the server uses two key I/O abstractions. The first is a base class representing an abstract digital I/O board (see Figure 2). This encapsulates central features provided by any digital I/O board, including functions to update its current input and output representations from and to the physical hardware, and to report the number of inputs and outputs it provides. All classes representing digital I/O boards are derived from this abstract base class. Such derived classes represent an instance of a specific I/O board from a particular manufacturer, and the derived classes have hardware-specific code to talk to the relevant manufacturer's driver software. This software architecture allows support for new manufacturers' equipment to be added in an encapsulated manner by adding a new class.

The second key abstraction within the server is of an individual digital I/O line. Lines represent, among other things, their state (on or off, this being updated to or from the physical hardware state periodically), their owner (if a client has claimed them), and any on/off events currently attached to them, so they can report those events to their client when the line's state changes. They also provide a layer at which the logical state can be disconnected from the physical state or from the state their client intended, enabling the user to manipulate digital I/O lines from the console for debugging purposes.

Safety Features

Some digital I/O cards, including those controlled by the common Intel 82C55 chip, default to settings that turn on all connected output devices when the controlling computer is powered up. If the devices controlled by these output lines are potentially data- or life-threatening (e.g., intravenous infusion pumps), inadvertent temporary power loss or rebooting of a computer may be hazardous. Uninterruptible power supply use is one important defense against this, but the server also allows a subset of digital output lines to be dedicated as power control lines ("failsafe" outputs). If these are attached to a power control relay and configured so that some control lines need to be on and others simultaneously off before power is provided to the devices, the risk of inadvertent device activation is substantially reduced. However, our system is not certified to control medical devices in humans and should not be used for this purpose. Failsafe outputs are controlled exclusively by the server and are not available to clients.

Clients may also specify that it is hazardous to leave particular lines on. The server will then ensure that none are left on for more than a specified time, in case the client fails to turn them off, and will return them to a specified safe state if communication with the client is lost. Typically, this facility is used to provide an extra degree of safety for intravenous infusion pumps, as used in animal drug self-administration experiments. The safety facility is implemented internally within the representation of a digital I/O line discussed above.

Graphics Architecture

Our objective was to make all graphics primitives of the Windows graphics device interface (GDI) available for multiple monitors, without the client's having to be aware of the programming problems associated with multimonitor use, and to make display objects capable of responding to events such as touches. We summarize key design decisions here; further implementation details are available online (see www.whiskercontrol.com). Display devices (monitors) are enumerated via the Windows operating system, allowing multimonitor configurations to be used directly. Typically, the primary display is used to view the conventional Windows desktop, including the Whisker server console and any tasks that are running (see Figure 3). Additional monitors are typically placed inside operant chambers (optionally with touchscreen equipment attached to them; see below) and are used to provide stimuli in behavioral tasks. For human testing, clients may also create "new" display devices, represented as windows on the conventional operating system desktop. A copy of all additional displays is provided on the server's console, with representations of subjects' screen touches, so stimuli and subjects' responses may be monitored from the console.

The display command set is described in full online (see www.whiskercontrol.com). Once a client has claimed ownership of a display device (see below), it may create arbitrary "documents," and draw to them with or without the document's being visible. One document at a time may be assigned to a display to make it visible. To maximize flexibility from the client's perspective, all Windows basic GDI calls are made available via the server's command set, allowing arbitrary graphics and text presentation. Bitmaps (BMP files) may be displayed, allowing stimuli to be created with third-party photographic or illustration software. Most classes of graphical object may have "touch" events attached to them (e.g., touched, touch removed, touch moved), so that if touch-detection equipment is present, the objects will generate events when touched.

Touchscreen Architecture

Our objective was to support the widest range of touchscreens possible, and to support multiple touchscreens simultaneously. Since there is a large range of commercial touchscreen equipment (touch-detection equipment sold attached to a monitor, or sold to be positioned in front of an existing monitor), and no unifying touchscreen architecture built into Windows NT/2000/XP/Vista, the server process relies on the commercial UPDD (Universal Pointing Device Driver; Touch-base Ltd, U.K.) common touchscreen interface, which in turn supports a large majority of available touchscreens, often via serial interfaces. Touchscreen sensors are configured and calibrated via the UPDD interface, and the server takes control of any touchscreen(s) selected for use with Whisker while it is running, via the UPDD driver (see Figure 2). Touchscreen sensors are assigned to display devices (monitors) by the server process, and the combination of a display device and a touchscreen sensor is treated as a single functional unit. Touchscreen events are detected via display objects (for details, see www.whiskercontrol.com) and thus require no specific commands.

Sound Architecture

In order to place the fewest restrictions on sound operation, the server process enumerates sound devices via the operating system (through the Windows DirectSound interface), and thus all Windows-enabled sound devices are available for use. Optionally, a stereophonic sound device may be split into two monophonic devices. If the device itself provides adequate stereo separation, this is a cheap way of doubling the number of sound outputs. "Fake" audio devices may be created for testing purposes on computers without sufficient sound cards.

The sound system was designed to allow clients to play arbitrary combinations of sounds. Having claimed a sound device, clients may attach any number of sound buffers to it, limited only by system memory. Buffers may be played independently, including simultaneously. Buffers may contain either simple sound waveforms (e.g., sine wave, square wave, etc.) specified by the client, or wave format (WAV) files stored on disk. The latter method allows arbitrary sounds to be played. Volume control features are provided. The audio command set is described online (see www.whiskercontrol.com).

An undesirable feature of DirectSound primary sound buffers (an undocumented bug) is that stopping a sound and starting another can produce a small click or burst of one of the sounds. Since this would be problematic in behavioral experiments, the server plays a "blank" sound continuously to prevent this problem.

Keyboard and Mouse Input

For human testing, keyboard and mouse events are also supported. Keyboard events are available from the display device that has the operating system focus. Mouse clicks are likewise processed through display devices. The keyboard and mouse are also used in the conventional way on the server's console, including for testing; for example, mouse input to the server's copy of an active display device may be used to mimic touchscreen input for testing purposes.

Resource Management for Clients

A single text-based device definition file is used to provide a mapping between device numbers (e.g., output line 20) and convenient device names (e.g., "PelletDispenser"). Thus, client behavioral tasks may be written without knowledge of the specific I/O device numbering on the host computer. Multiple names are supported, so that a device may be known by different names to different tasks. Devices may also be grouped; typically, a group of devices represents an operant chamber. This allows clients to refer to the "PelletDispenser" in "chamber1." All device types (including display and sound devices) may be addressed in this way.

Client processes request ("claim") devices or groups of devices (for syntax, see www.whiskercontrol.com). Once claimed, the server process will prevent any other client from claiming or accessing those devices until the client frees the resources or disconnects from the server. This prevents unwanted crosstalk between operant chambers. Intentional crosstalk is still possible if one client program takes control of several operant chambers, or if two separate client programs cooperate deliberately, and the server supports client–client communication.

Having claimed devices, the client may also assign them aliases and refer to them by alias. Clients have the option to create aliases on the fly, and to assign the same alias to multiple devices, thereby allowing simple programming of counterbalancing and yoking. Counterbalancing might be accomplished, for example, by creating an alias LEVER to refer to either LEFTLEVER or RIGHTLEVER in the counterbalancing code, and then referring only to LEVER in the main task code. Yoking might be accomplished, for example, by assigning the alias LIGHT to the stimulus lights in all of several operant chambers, so when the task proper switches on the LIGHT, the lights in all the chambers come on.

Server-Based and Remote Monitoring and Debugging

Clients are encouraged to provide status information to the server, allowing the server's console to provide a snapshot status view of all current clients, in addition to any status information the clients choose to display themselves. The server's console allows all aspects of current tasks to be monitored, including digital I/O line status, active timers, copies of any graphical displays in use, events that have been generated, client–server communications, and so on (see Figure 3). The console allows manipulation of I/O devices and events for testing purposes (e.g., manually turning devices on and off, or simulating subjects' responses), and provides test facilities for display, touchscreen, and sound devices.

Remote access to the server process, if permitted by the user, allows status monitoring to be conducted from computers anywhere on the same IP network (see Figure 2; for details, see www.whiskercontrol.com). In certain situations, behavioral tasks themselves could run on different computer systems connected to the computer hosting the Whisker server process via a network; of course, this approach requires guaranteed network performance.

Data Capture

Although event and communication log files may be captured and saved from the server process, primarily for debugging purposes, data capture is the responsibility of the client process (the behavioral task), and this separation of responsibility contributes to server performance and reliability. In our practice, we write clients that record data both to an appropriately structured relational database, for power in data queries, and to a text file as a backup. However, for additional security, the client can request the server to log events, communications, and arbitrary client-generated messages to disk (see www.whiskercontrol.com). These logs can be digitally signed by the server to assist compliance with regulatory frameworks such as the Good Laboratory Practice (GLP) system (U.S. Food and Drug Administration, 2002).

Auxiliary Software

The status of current behavioral tasks can be monitored remotely using a remote status client, or via a Web browser (provided the computer hosting the server runs a third-party Web server and Java applet). A demonstration client allows manual testing and learning of the system commands. Since many client tasks we have written use the open database connectivity (ODBC) interface for structured data storage, a tool to manage ODBC connections and databases is also supplied.

Examples of behavioral task frameworks are available in Perl, Python, Visual Basic, and C++ (see www.whiskercontrol.com). As discussed above, the free choice of client programming language enables procedural programming, the use of state-based models, or any other approach to behavioral task creation.

Performance, Practical Experience, and Behavioral Tasks

The concept of client–server separation of responsibilities aims for reliability, efficiency, and performance through specialization and simplicity, akin to the programming philosophy behind the development of UNIX (Ritchie & Thompson, 1974). In practice, we have found that client–server separation generates an extremely reliable platform. We estimate that the server process has run for well over a quarter of a million hours in our own institution from 2000 to 2008; at the time of writing in late 2009, we are unaware of any server system "crash" in the worldwide deployment during the last 5 years.

The system is also fast enough for behavioral testing, reliably providing digital I/O facilities at 1 kHz, even on a decade-old computer: The slowest computer tested to date is a 1999 system with an AMD K6-2/450 processor and 128-MB RAM under Microsoft Windows NT 4.0. Subsequent improvements in typical computing speed have meant that computing performance is rarely a consideration. Timing performance is described above.

The client–server separation, device abstraction, and the test facilities built into the server mean that it is easy to write and test behavioral task software fully without needing access to physical operant chamber hardware; this frees up the laboratory hardware for active research.

The system has now been used for research involving human and nonhuman primates, rodents, dogs, pigs, and birds. Several research groups use the system with their own client task software, and others use precompiled tasks designed and written in the University of Cambridge, including the five-choice serial reaction time task, second-order schedules of reinforcement paradigms, delay-discounting choice tasks, and the MonkeyCantab touchscreen-based task suite (Roberts, Robbins, Everitt, & Muir, 1992; see Figure 3) that implements a primate-oriented version of the human Cambridge neuropsychological test automated battery (CANTAB) (Robbins et al., 1994; Sahakian & Owen, 1992). Published research conducted encompasses human work (e.g., Fairchild et al., 2009; Kehagia, Cools, Barker, & Robbins, 2009) and animal work (e.g., Belin, Mar, Dalley, Robbins, & Everitt, 2008; Dalley et al., 2007; Hutcheson, Everitt, Robbins, & Dickinson, 2001; Ito, Robbins, & Everitt, 2004; Mui, Haselgrove, Pearce, & Heyes, 2008; Vanderschuren & Everitt, 2004; Von Huben et al., 2006; also see Cardinal & Aitken, 2008).

Advantages and Disadvantages of Client–Server Versus Monolithic Architectures for Behavioral Control

We think that the trade-offs in complexity, reliability, and flexibility between client–server and monolithic methods for behavioral experiment control are, in general, as follows.

Complexity is the major disadvantage of the client–server system, although the actual level of complexity for the end user can vary considerably. Client–server models involve considerable complexity in programming for the server process, although this is a one-off problem. Programming of the client processes may be simple or complex (depending on the choice of language, the programming model followed, and so on), although there is a minimum extra level of complexity required of the clients in that they must incorporate a system for network communications with the server process. Monolithic architectures (e.g., using a custom script language) do not require the same degree of network communications programming by the architecture's creator, but require more language support for behavioral tasks; for example, they must support conditional execution and other programming constructs that may be left to the client in a client–server model. The complexity of a behavioral task in a monolithic system is determined by the custom script language, although support for network communications is not required in this situation.

Complexity for users is thus closely related to the choice of programming or scripting language, and the choice of a client–server architecture may at first appear to be independent of the choice of a scripting language. That is, within a client–server architecture such as ours, programmers may write clients that communicate directly with the server process using the Whisker command set, but it is also possible to create intermediate steps, such as graphical or text-based script interpreters to interpret a different script language and translate it into commands passed to the Whisker server process. Under such a system, the end user would work with the alternative graphical system or script language, and retain the other benefits of the client–server architecture (such as independence of multiple client processes). However, the converse is not necessarily true: In a system based on a single custom script language whose interpreter also controls the physical hardware (e.g., Campden, 2005; Fray, 1980; Lafayette, 2007; Med Associates, 2004; Palya & Walter, 1993; Panlab, 2004; Tatham & Zurn, 1989), the choice of programming language is not free, and clients are independent only if the script interpreter enables this. In either case, the range of physical hardware that may be controlled depends on the process that actually communicates with that hardware.

The reliability of client–server versus monolithic systems is an empirical question. Client–server architectures allow their clients to fail without disrupting other clients, and the server process does not take responsibility for client reliability. Client–server models also have a potential failure point in the network connecting the client and server processes; this is a real problem if a network between computers is used, but not a practical problem if the client and server processes run on the same physical computer. Monolithic architectures take responsibility for execution of their behavioral tasks, and therefore for their reliability. In practice, we have found a client–server system to be highly reliable, as discussed above.

In terms of flexibility, there are clear advantages to the client–server system, given the independence of execution and of programming language for the client processes.

Research Applications and Users

To use directly a client–server system such as the one we outline here, users must be able to program in a general-purpose programming language with TCP/IP network capabilities. This is an undoubted hurdle, exposing users to languages that may be more powerful and complex than typical dedicated scripting languages, and it probably restricts use to a subset of undergraduates, graduate students, and researchers with some programming experience. The choice of language determines the complexity; for example, it is likely that writing a text-based client in Python using prewritten examples (see www.whiskercontrol.com) is much simpler and within the grasp of more researchers than writing a GUI-based client in C++; both methods could achieve the same experimental aim, and the choice is a matter of personal preference. In either case, there is also a need to deal, directly or indirectly, with the server's command set. However, an alternative option also exists to use a client–server system indirectly through a special client designed to make task programming easy or graphically based (e.g., Campden/Lafayette, 2009).

The strongest arguments for using a client–server system come when a single computer is used to control multiple independent test stations, and thus a single server process would communicate with multiple client processes. This reflects the situation in which digital I/O control hardware and computers are relatively expensive compared with operant chambers (including when an expensive I/O control card and interface can support many more devices than would be found in a single operant chamber), so the researcher would not wish to assign a computer, I/O controller, and other I/O interface equipment for each operant chamber. It also reflects situations in which operant chambers may be used independently or in groups (e.g., for yoked experiments), again requiring one computer to communicate with several chambers. In these situations, client–server architectures provide efficient use of experimental resources. In our experience, these situations are most common in rodent, bird, and primate research within behavioral neuroscience and experimental psychology.

Conclusion

Real-time research control systems based on a client–server architecture are feasible and reliable. A system based on TCP/IP communications and a tightly controlled timing architecture can implement 1-kHz digital I/O performance on cheap and readily available hardware and consumer operating systems. Internally, our system's abstractions allow the control of interfaces from multiple manufacturers, while providing safety features, contention management, and simplified control of a range of hardware devices (including visual displays, sound cards, and touchscreens) in any programming language capable of TCP/IP communications. The use of a client–server architecture can carry some costs in terms of programming complexity for the clients, but provides substantial practical benefits in terms of task software development flexibility, efficient use of experimental resources, and data management.

AUTHOR NOTE

The authors thank Barry Everitt, Rob Milner, Maria Pilla, Simon Howes, and Dan Hutcheson for discussion, support, and testing. R.N.C., M.R.F.A., the Department of Experimental Psychology at the University of Cambridge, and the University of Cambridge itself all receive royalty income from sales of licenses for use of Whisker and of those client tasks written by R.N.C. and M.R.F.A. that are sold commercially. Correspondence concerning this article should be addressed to R. N. Cardinal, Behavioural and Clinical Neuroscience Institute, Department of Psychiatry, University of Cambridge, Addenbrooke's Hospital, Hills Road, Cambridge CB2 0QQ, England (e-mail: rudolf.cardinal@pobox.com).

REFERENCES

Belin, D., Mar, A. C., Dalley, J. W., Robbins, T. W., & Everitt, B. J. (2008). High impulsivity predicts the switch to compulsive cocaine-taking. *Science*, **320**, 1352-1355.

Bussey, T. J., Padain, T. L., Skillings, E. A., Winters, B. D., Morton, A. J., & Saksida, L. M. (2008). The touchscreen cognitive testing method for rodents: How to get the best out of your rat. *Learning & Memory*, **15**, 516-523.

Bussey, T. J., Saksida, L. M., & Rothblat, L. A. (2001). Discrimination of computer-graphic stimuli by mice: A method for the behavioral characterization of transgenic and gene-knockout models. *Behavioral Neuroscience*, **115**, 957-960.

Campden (2005). Behavioural Net Controller Icon. Loughborough, U.K.: Campden Instruments. Available at www.campdeninstruments.com.

Campden/Lafayette (2009). ABET-II Touch. Loughborough, U.K., and Lafayette, IN: Campden Instruments and Lafayette Instrument Company. Available at www.campdeninstruments.com and www.lafayetteinstrument.com.

Cardinal, R. N., & Aitken, M. R. F. (2008). *Whisker*. Retrieved August 20, 2008, from www.whiskercontrol.com.

Cerf, V. [G.], Dalal, Y., & Sunshine, C. (1974). RFC675—Specification of Internet Transmission Control Program. Internet Engineering Task Force. Retrieved November 19, 2009, from www.ietf.org/rfc/rfc0675.txt.

Cerf, V. G., & Kahn, R. E. (1974). A protocol for packet network intercommunication. *IEEE Transactions on Communications*, **22**, 637-648.

Cohen, A., & Woodring, M. (1998). *Win32 Multithreaded Programming*. Sebastopol, CA: O'Reilly.

Dalley, J. W., Fryer, T. D., Brichard, L., Robinson, E. S., Theobald, D. E., Laane, K., et al. (2007). Nucleus accumbens D2/3 receptors predict trait impulsivity and cocaine reinforcement. *Science*, **315**, 1267-1270.

Dixon, P. (2009). A hybrid approach to experimental control. *Behavior Research Methods*, **41**, 615-622.

Fairchild, G., van Goozen, S. H., Stollery, S. J., Aitken, M. R., Savage, J., Moore, S. C., & Goodyer, I. M. (2009). Decision making and executive function in male adolescents with early-onset or adolescence-onset conduct disorder and control subjects. *Biological Psychiatry*, **66**, 162-168.

Fray, P. J. (1980). ONLIBASIC, a system for experimental control. *Trends in Neurosciences*, **3**, 13-14.

Fray, P. J. (1988). Spider (extension to BBC BASIC for the BBC Microcomputer). Cambridge: Paul Fray Ltd.

Fray, P. J. (1990). Arachnid (extension to BBC BASIC V for the Acorn Archimedes). Cambridge: Paul Fray Ltd.

Fray, P. J. (1993). Personal computers and the control of behavioural experiments. In A. Saghal (Ed.), *Behavioural neuroscience: A practical approach* (Vol. 1, pp. 185-210). New York: Oxford University Press.

Fray, P. J., & Robbins, T. W. (1996). CANTAB battery: Proposed utility in neurotoxicology. *Neurotoxicology & Teratology*, **18**, 499-504.

Gibson, B. M., Wasserman, E. A., Frei, L., & Miller, K. (2004). Recent advances in operant conditioning technology: A versatile and affordable computerized touchscreen system. *Behavior Research Methods, Instruments, & Computers*, **36**, 355-362.

Hutcheson, D. M., Everitt, B. J., Robbins, T. W., & Dickinson, A. (2001). The role of withdrawal in heroin addiction: Enhances reward or promotes avoidance? *Nature Neuroscience*, **4**, 943-947.

Internet Assigned Numbers Authority (2008). *Port numbers*. Retrieved August 20, 2008, from www.iana.org/assignments/port-numbers.

Ito, R., Robbins, T. W., & Everitt, B. J. (2004). Differential control over cocaine-seeking behavior by nucleus accumbens core and shell. *Nature Neuroscience*, **7**, 389-397.

Kehagia, A. A., Cools, R., Barker, R. A., & Robbins, T. W. (2009). Switching between abstract rules reflects disease severity but not dopaminergic status in Parkinson's disease. *Neuropsychologia*, **47**, 1117-1127.

Lafayette (2007). ABET II. Lafayette, IN: Lafayette Instrument Company. Available at www.lafayetteinstrument.com.

Markham, M. R., Butt, A. E., & Dougher, M. J. (1996). A computer touch-screen apparatus for training visual discriminations in rats. *Journal of the Experimental Analysis of Behavior*, **65**, 173-182.

Med Associates (2004). Med-PC Version IV. St. Albans, VT: Med Associates (www.med-associates.com/software/medpc.htm).

Mui, R., Haselgrove, M., Pearce, J., & Heyes, C. (2008). Automatic imitation in budgerigars. *Proceedings of the Royal Society B*, **275**, 2547-2553.

Nagle, J. (1984). RFC896—Congestion control in IP/TCP Internetworks. Internet Engineering Task Force. Retrieved November 18, 2009, from www.ietf.org/rfc/rfc0896.txt.

Palya, W. L., & Walter, D. E. (1993). A powerful, inexpensive experiment controller for IBM PC interface and experiment control language. *Behavior Research Methods, Instruments, & Computers*, **25**, 127-136.

Panlab (2004). PackWin. Barcelona, Spain, and Holliston, MA: Panlab S.L. Harvard Bioscience, Inc. Available at www.panlab.com.

Parkinson, J. A., Crofts, H. S., McGuigan, M., Tomic, D. L., Everitt, B. J., & Roberts, A. C. (2001). The role of the primate amygdala in conditioned reinforcement. *Journal of Neuroscience*, **21**, 7770-7780.

Postel, J. (1981). RFC793—Transmission control protocol, DARPA Internet program, protocol specification. Information Sciences Institute/ Defense Advanced Research Projects Agency. Retrieved November 19, 2009, from www.ietf.org/rfc/rfc0793.txt.

Ritchie, D. M., & Thompson, K. (1974). The UNIX time-sharing system. *Communications of the ACM*, **17**, 365-375.

Robbins, T. W., James, M., Owen, A. M., Sahakian, B. J., McInnes, L., & Rabbitt, P. (1994). Cambridge Neuropsychological Test Automated Battery (CANTAB): A factor analytic study of a large sample of normal elderly volunteers. *Dementia*, **5**, 266-281.

Roberts, A. C., Robbins, T. W., Everitt, B. J., & Muir, J. L. (1992). A specific form of cognitive rigidity following excitotoxic lesions of the basal forebrain in marmosets. *Neuroscience*, **47**, 251-264.

Sahakian, B. J., & Owen, A. M. (1992). Computerized assessment in neuropsychiatry using CANTAB: Discussion paper. *Journal of the Royal Society of Medicine*, **85**, 399-402.

Skinner, B. F. (1938). *The behavior of organisms: An experimental analysis*. New York: Appleton-Century-Crofts.

Stevens, W. R. (1998). *UNIX network programming, Vol. 1. Networking APIs: Sockets and XTI*. Upper Saddle River, NJ: Prentice Hall.

Stroustrup, B. (1986). *The C++ programming language*. Reading, MA: Addison-Wesley.

Tatham, T. A., & Zurn, K. R. (1989). The MED-PC experimental apparatus programming system. *Behavior Research Methods, Instruments, & Computers*, **21**, 294-302 [currently available from Med Associates Inc., St. Albans, VT: www.med-associates.com/software/ medpc.htm].

U.S. Food and Drug Administration (2002). 21 CFR 58.185 (Code of Federal Regulations Title 21: Food and Drugs; Chapter 1: Food and Drug Administration, Department of Health and Human Services; Part 58: Good Laboratory Practice for Nonclinical Laboratory Studies). Available at www.access.gpo.gov/nara/cfr/ waisidx_02/21cfr58_02.html.

Vanderschuren, L. J., & Everitt, B. J. (2004). Drug seeking becomes compulsive after prolonged cocaine self-administration. *Science*, **305**, 1017-1019.

Von Huben, S. N., Davis, S. A., Lay, C. C., Katner, S. N., Crean, R. D., & Taffe, M. A. (2006). Differential contributions of dopaminergic D1- and D2-like receptors to cognitive function in rhesus monkeys. *Psychopharmacology*, **188**, 586-596.

Wirth, N. (1971). The programming language Pascal. *Acta Informatica*, **1**, 35-63.

(Manuscript received September 4, 2009;
revision accepted for publication May 11, 2010.)