A MOdular **R**emote **A**ibo **I**nterface (**AMORAI**)

http://www.cs.umn.edu/~chilton/amorai/
John Chilton - chilton@cs.umn.edu

Overview:

AMORAI allows for the control of the Sony AIBO robot from a remote PC. On the AIBO side, AMORAI is a collection of Tekkotsu (http://www.tekkotsu.org/) behaviors that send data out and accept commands via the AIBO's wireless. AMORAI isn't a specific piece of client software on the remote PC, but is a socket based API that allows for the easy creation of clients in any sufficiently powerful programming language.

TekkotsuMon is a remote framework that also allows for the remote control of the Sony AIBO. Like AMORAI, TekkotsuMon is in part a collection of Tekkotsu behaviors that send data and respond to commands, and unlike AMORAI, TekkotsuMon is also a nice piece of client software for the remote machine that is written in Java. Other clients could be developed that make use of the TekkotsuMon AIBO behaviors. Pyro (http://emergent.brynmawr.edu/pyro/) is one such client. So the question is raised why AMORAI?

There are several issues with using TekkotsuMon as general purpose socket based API for controlling the AIBO, which, in its defense, it was probably never meant to be. The biggest such is there are simply things that it cannot do, such as controlling the AIBO's LEDs, and expanding TekkotsuMon to include such a behavior would require you to write the behavior essentially from scratch, and would be, even if you know Tekkotsu, a nontrivial task, requiring you to do tasks such as establishing the wireless connection and explicitly reading to and writing from sockets. AMORAI attempts to address this by taking these tasks that must be done by each behavior and doing them behinds the scenes, greatly simplifying the task of writing new wireless behaviors.

Another advantage of AMORAI over TekkotsuMon is that the complexity of the interface is slightly reduced. Specifically activating a TekkotsuMon (without modifying the TekkotsuMon code) requires connecting a TekkotsuMon main controller and navigating the Tekkotsu menus to activate the behaviors you want to interact with. AMORAI lets you specify which behaviors you want to start at startup with a plain text file on the AIBO's memory stick (/ms/config/amorai.cfg), that way in your client program, you can just connect to the right port and start writing commands and reading data.

In addition to being able to construct new listeners and writers easily, there are already many existing AMORAI behaviors for things that TekkotsuMon can do like controlling the the joints and sending out the AIBO's world state, and for things that TekkotsuMon does not currently do like controlling the AIBO's LEDs, controlling the AIBO's ears, and running posture

(.pos) files. In fact you can download an AMORAI memory stick image, add a correct wlanconf file for your network, and edit the /ms/config/amorai.cfg file to make sure the right behaviors are starting, and just use AMORAI as is, without needing to know a thing about Tekkotsu or compiling code for the AIBO.

AMORAI is written entirely on top of Tekkotsu as a Tekkotsu “project”, so no changes need to be made to the existing Tekkotsu libraries and classes. So once Tekkotsu has been set up on a system, it should be no problem to modify or extend AMORAI with behaviors. Installing the AMORAI source code should be as easy as unpacking the AMORAI source code underneath the Tekkotsu directory and copy a properly configured Environment.conf file into the AMORAI directory. Then just building the project with 'make' just like any other Tekkotsu project.

AMORAI – The AIBO/Tekkotsu Side

At the present time there are two board classes of AMORAI behaviors, ones that wait for data from a remote computer and ones that send data to remotely connected machines. The former are referred to as listeners and the latter as writers. All of these are just Tekkotsu behaviors, which are implemented as C++ classes. Listener behaviors are implemented as a derived class of AMORAI's SocketListener class, which handles all of the work associated setting up the appropriate socket listener and callback function, and allocating the read buffer. Similarly, AMORAI writer behaviors are implemented as derived classes of the SocketWriter class. It should be noted, that none of the code AMORAI uses to utilize the AIBO's wireless capabilities is original, it uses Tekkotsu's existing infrastructure and ideas taken from the TekkotsuMon classes, all that AMORAI's SocketListener and SocketWriter classes do is hide the details from the AMORAI/Tekkotsu developer. As an illustration of what is hidden, Appendix A has an AMORAI behavior and a comparable TekkotsuMon behavior.

For both listeners and writers, there is an additional subdivision in the way data transfer is handled. If a listener expects to always receive the same number of bytes or if a writer always sends the number of bytes, then they are referred to as fixed buffer listeners or writers. If this is not case, then when sending or receiving data the number of bytes being transferred must proceed the actual data being transferred. LEDListener is a listener which expects 27 4 byte IEEE floating point numbers each time it is sent data, one for each LED the AIBO can control. Hence every transfer is of 108 bytes of data, and hence this is a fixed buffer listener. PostureListener on the other hand is not a fixed buffer listener, since it expects a string composed of 1 byte characters that specifies a posture filename to run, and this string can vary in length. More details on how to implement each are discussed below.

As stated above AMORAI behaviors are derived from either SocketListener or SocketWriter. Additionally, each AMORAI behvaior should have is a default C++ class constructor, that is a constructor with no arguments. Inside the constructors initializer list the SocketListener's or the SocketWriter's constructor should be called. Both classes constructor's take in three arguments, the name of the class being written as a string, a boolen value specifying whether this is a fixed buffer listener (or writer), and the third value should be a positive integer. If this class is going to be a fixed buffer class then this integer specifies the number of bytes in that buffer, if it is not a fixed buffer class then this integer is an upper limit on the number of bytes a message may contain (this amount of memory will be allocated if the behavior is started). An example of this is given below.

```
class HelloWorldListener : public SocketListener {
     HelloWorldListener() :
          SocketListener(“HelloWorldListener”, false, 1024)
     {}
```

Tekkotsu BehaviorBase is a super class of both the SocketListener and SocketWriter class, so AMORAI listeners and writers should contain overloaded DoStart() and DoStop() methods which are executed when a the listener or writer starts up and when they are shut down. Also it is in the super class's corresponding method that memory is allocated and the wireless configuration is handled, so in both methods the super classes corresponding method is the first thing executed. For our HelloWorldListener example this looks like:

```
     virtual void DoStart() {
          SocketListner::DoStart();
     }
     virtual void DoStop() {
          SocketListener::DoStop();
     }
```

At this point implementing listeners and writers diverge. For listeners, the method processData() should be overloaded for fixed buffer classes, or processData(int size) for classes that are not fixed buffer. These functions are called when data has been sent to classes. The data sent is stored in the member variable *buffer*, which is of type char*. In the case when the class is not a fixed buffer class, the data does not include the size, but the size of the message is the argument to the processData method. Finishing off the HelloWorldListener example looks like this:

```
     virtual void processData(int size) {
          /* Will print "Hello World" is this message is sent to
it*/
          char* str = calloc(sizeof(char), size+1);
          memcpy(buffer, str, size);
          str[size] = '\0';
          sout->printf("%s\n", str);
     }
};
```

At the present time, each message sent to a listener should be proceeded by a 4 byte integer indicating the port the data is being sent on. Removing this requirement is one of things that needs to be worked on. Anyway, just like the size of buffer for not fixed buffer objects, this is removed from the buffer before the processData method is called.

Using the SocketWriter is marginally more complicated. Sending data is a two part process, first the member method ready() should be called which will setup the member variable *buffer*, again of type char*, for sending. Then the data to be sent should be inserted into the buffer, and then method send() should be called. In the case of classes that are not fixed buffer, send should be called with the number of bytes to send.

Most likely, writer objects will need to send messages in response to events, for example new sensor data or button presses, to do so overloading the Tekkotsu BehaviorBase method processEvent(Event) will be necessary. For an example of this and using ready()/send() checkout the source code for WorldStateWriter.

The AIBO more likely than not does not process the processing power of the machine it will communicate with, so existing AMORAI classes do not convert data to network byte order before sending or convert from network bytes before processing. The AIBO's architecture is little endian, so this will not be a problem if your host machine is an X86 compatible PC, but architectures using big endian byte order such as Sun Machines will need to convert the byte order of multi-byte data types such as ints and floats.

Outside of the file(s) containing a listener or writers definition, there is only one other file that needs to be edited. AMORAIStartupBehavior.cc needs to be edited so that AMORAI knows about the new class. There are two things that need to be done in this file, the first is there needs to be a #include statement for the header file that contains the new class's declaration. Also a line needs to added in the initailize() method for the new class. For our HelloWorldListener class this would look like

```
     add(AMORAIController<HelloWorldListener>(),
        “HelloWorldListener”,
        13099, true);
```

“HelloWorldBehavior” needs to be the same string that was passed to the SocketListener or SocketWriter constructor back in the definition. The most clear name is just the name of the class. In the above example, 13099 is the port the class will listen on or write to. The true just indicates that the an object of this class should be created and setup at startup. The same thing must be done for new writers.

Alternatively it is not necessary to pass these later two arguments if there values are specified in the AMORAI config file, i.e. /ms/config/amorai.cfg. For this example, the following two lines would need to added to the file:
HelloWorldListenerPort 13099
HelloWorldListenerStart true

Specifying these two things in the configuration file is the preferred way of doing it, since it then allows changing the port or preventing the object from running without needing to recompile the Tekkotsu objects.

AMORAI – The Client Side

In theory, software can be written to control the Sony AIBO ERS7 with AMORAI in any programming language that has socket capabilities and allows for the reading and writing of binary data. Working sample code for Guile scheme and C/C++ can be found at the AMORAI website. Also on the website Python code can be found to add novel features to Pyro.

AMORAI Library

The following AMORAI listeners and writers are compiled into the AMORAI memory stick image, and can be used without needing to know anything about Tekkotsu and without compiling. The guile scheme and C/C++ code mentioned above make use of the following listeners and writers.

LEDListener
Port: 13001
Fixed Buffer: Yes
Interface: Expects 27 floating point numbers from 0-1.0
Description: Uses Tekkotsu's LedMC controller to set each led to the value sent. Soon a list of which led corresponds to which light will be available at the AMORAI website.

JointListener:
Port: 13005

Fixed Buffer: Yes
Interface: Expects 17 floating point values
Descriptions: Each value corresponds to a joint position, adjusts each joint to to value sent. Unfortunately, the values seem random, the people working on Pyro have done the work of normalizing these values to a -1.0-1.0 scale. This normalization should probably be done on the client side for because of the AIBO's resource limitations. This class is analogous to TekkotsuMon's Aibo3DControllerBehavior.

WalkListener:
Port: 13002
Fixed Buffer: Yes
Interface: Expects 3 floating values on -1.0-1.0
Description: The values in the order given correspond to percent of max walk velocities in the forward-back, side-to-side, and rotational directions.

PostureListener:
Port: 13003
Fixed Buffer: No
Interface: String of ASCII characters (do not null terminate and remember to send size of string as integer first)
Description: Send the name of a posture file in the AIBO's /ms/data/motion/ directory and it will be ran, right now the only ones that work well are 'stand.pos', 'situp.pos', and 'pounce.pos'.

EarListener:
Port: 13004
Fixed Buffer: Yes
Interface: 2 Floating point values (each either 0.0 or 1.0)
Description: Set the value of AIBO's ears. [0,.5) correspond to the ears being in the lowered stated, and [.5,1.0] correspond to the ears being in their raised state.

OutputListener:
Port: 13010
Fixed Buffer: No
Interface: n integer values followed by n floating point values
Description: Uses Tekkotsu's OutputMC, to set the values of the outputs indexed by the n integers to the values corresponding to the same position in the n floating point values. Valid outputs include the AIBO's joints, LEDs, and ears. Which indexes correspond to which of these outputs will can be found at the AMORAI website soon.

WorldStateWriter:

- Port: 13101
- Fixed Buffer: Yes
- Interface: (In order)
  - 1 integer (Time stamp of sensor reading)
  - 1 integer (Number of PID joints (i.e. 18))
  - 54 floats (PID values)
  - 18 floats (joint values)
  - 1 integer (number of sensors (i.e. 11))
  - 11 floats (sensor values)
  - 1 integer (number of buttons (i.e. 10))
  - 10 floats (button values)
  - 18 floats (pid utility values)
  - (460 bytes)

Description: This classed was based off of TekkotsuMon's WorldStateSerializer behavior. The time interval between writing world states can be adjusted by adjusting the configuration parameter WorldStateInterval in the ms/config/amorai.cfg file. This number is in units of milliseconds.

Future Work:

There are a lot of exciting things that are planned for this project. As mentioned above, it would be nice to eliminate the necessity to send the port number to listeners as part of the data sent.

On the client side, it would be nice to turn the existing C client code into a shared a library. This would make the process of porting the client code to new languages faster. In the mean time, I plan to duplicate the C and Scheme efforts in other popular languages, including Python, PHP, and Perl.

Also I plan to increase the number of existing AMORAI listeners and writers. In terms of listeners, I plan to add a listener for playing sounds, a timed walked listener. In terms of writers, I like to implement a mechanism for registering events, like button taps or getting close to a wall, and then have a message sent when they occur. I also plan to develop slimmed down versions of WorldStateWriter. One just for the joint positions, and then one just for the button and sensor values.

Other plans for the project include increasing the amount of documentation, developing the website, and creating a help wiki.

Appendix A: A comparison of AMORAI and TekkotsuMon

JointListener from the AMORAI library does about the same thing as TekkotsuMon's Aibo3DControllerBehavior. Their socket interfaces are the same and both control joint movement, although Aibo3DControllerBehavior has a few lines dealing with the Java GUI. As a demonstration of the details hidden by AMORAI, below is JointListener and then Aibo3DControllerBehavior.

JointListener.h Contents –

```
#ifndef INCLUDED_JointListener_h_
#define INCLUDED_JointListener_h_

#include "Motion/MotionManager.h"
#include "Motion/RemoteControllerMC.h"
#include "Shared/RobotInfo.h"
#include "Shared/ERS7Info.h"
#include "SocketListener.h"
#include "Shared/SharedObject.h"

class JointListener : public SocketListener {
 protected:
  MotionManager::MC_ID rcontrolID;
  float* jointVals;

 public:
  JointListener() :
    SocketListener("JointListener", true, NumPIDJoints*sizeof(float)),
    jointVals((float*)buffer), //Joint vals aliases buffer as float*
    rcontrolID(MotionManager::invalid_MC_ID)
  {}

  virtual void processData() { //Called when buffer fills
    //Code Directly From AIBO3DController
    RemoteControllerMC *rcontrol = (RemoteControllerMC*)motman-
>checkoutMotion(rcontrolID);
    for (unsigned int i=0; i<NumPIDJoints; i++)
      rcontrol->cmds[i]=jointVals[i];
    rcontrol->setDirty();
    motman->checkinMotion(rcontrolID);
  }

  virtual void DoStart() {
    SocketListener::DoStart();
    rcontrolID = motman->addPersistentMotion(SharedObject<RemoteControllerMC>());
  }

  virtual void DoStop() {
    motman->removeMotion(rcontrolID);
    SocketListener::DoStop();
  }

 private: //For Error Suppression
  JointListener(const JointListener&);
  JointListener operator=(const JointListener&);

};

#endif
```

Aiibo3DControllerBehavior.h –

```
#ifndef INCLUDED_Aibo3DControllerBehavior_h_
#define INCLUDED_Aibo3DControllerBehavior_h_

#include <iostream>
#include "Wireless/Wireless.h"
#include "Behaviors/BehaviorBase.h"
#include "Motion/MotionManager.h"
#include "Motion/RemoteControllerMC.h"
#include "Events/EventRouter.h"
#include "Events/EventBase.h"
#include "Shared/RobotInfo.h"
#include "Behaviors/Controller.h"
#include "Shared/WorldState.h"

//! gets input from the GUI
int aibo3dcontrollercmd_callback(char *buf, int bytes);
class Aibo3DControllerBehavior;

//! so aibo3dcontrollercmd_callback knows where to send the input from the GUI
Aibo3DControllerBehavior *aibo3dControllerBehavior = NULL;


//! Listens to aibo3d control commands coming in from the command port.
class Aibo3DControllerBehavior : public BehaviorBase {
 protected:
        MotionManager::MC_ID rcontrol_id; //!< remote controller motion command's id

        //! The input command stream socket
        Socket *cmdsock;
 protected:
        MotionManager::MC_ID rcontrol_id; //!< remote controller motion command's id

        //! The input command stream socket
        Socket *cmdsock;

  float val[NumPIDJoints]; //!< the value to use for each of the PID joints
  char *fbuf;  //!< alias to val
  unsigned int pos; //!< a counter to know when we've gotten 4 frames

 private:
        Aibo3DControllerBehavior(const Aibo3DControllerBehavior&); //!< don't call
        Aibo3DControllerBehavior operator=(const Aibo3DControllerBehavior&); //!<
don't call

 public:
        //! constructor
        Aibo3DControllerBehavior() :
          BehaviorBase("Aibo3DControllerBehavior"),
          rcontrol_id(MotionManager::invalid_MC_ID),
          cmdsock(NULL),
          fbuf((char*)val), pos(0)
        { aibo3dControllerBehavior = this; }
        //! destructor
        virtual ~Aibo3DControllerBehavior() { aibo3dControllerBehavior = NULL; }

        //! processes input from the GUI
  int registerData(char *buf, int bytes) {
    int read=0;
    while (read<bytes) {
      fbuf[pos]=buf[read];
                        pos++;
                        read++;
      if (pos==NumPIDJoints*sizeof(float)) {
```

```
                                updateRC();
                                pos=0;
                        }
    }


        //! sends the new joint commands to the motion command
        void updateRC() {
                RemoteControllerMC *rcontrol = (RemoteControllerMC*)motman-
>checkoutMotion(rcontrol_id);
                for (unsigned int i=0; i<NumPIDJoints; i++)
                        rcontrol->cmds[i]=val[i];
                rcontrol->setDirty();
                motman->checkinMotion(rcontrol_id);
        }

        virtual void DoStart() {
                // Behavior startup
                BehaviorBase::DoStart();
                for(unsigned int i=0; i<NumPIDJoints; i++)
                        val[i]=state->outputs[i];
                // Enable remote control stream
                rcontrol_id = motman-
>addPersistentMotion(SharedObject<RemoteControllerMC>());
                updateRC();
                // Turn on wireless
                cmdsock=wireless->socket(SocketNS::SOCK_STREAM, 2048, 2048);
                wireless->setReceiver(cmdsock->sock, aibo3dcontrollercmd_callback);
                wireless->setDaemon(cmdsock,true);
                wireless->listen(cmdsock->sock, config->main.aibo3d_port);
                // open gui
                /*              std::vector<std::string> tmp;
                                        tmp.push_back("Aibo3D Load Instructions");
                                        tmp.push_back("To load Aibo3D, you will need
to install java3d\nand then run Tekkotsu/tools/aibo3d/")\
;
                                        tmp.back()+=getGUIType();
                                        Controller::loadGUI("ControllerMsg","LoadAibo3
d",getPort(),tmp);*/
                Controller::loadGUI(getGUIType(),getGUIType(),getPort());
        }

        virtual void DoStop() {
                Controller::closeGUI(getGUIType());
                // Close socket; turn wireless off
                wireless->setDaemon(cmdsock,false);
                wireless->close(cmdsock);
                // Disable remote control
                motman->removeMotion(rcontrol_id);
                // Total behavior stop
                BehaviorBase::DoStop();
        }

        //! returns string corresponding to the Java GUI which should be launched
        virtual std::string getGUIType() const { return
"org.tekkotsu.aibo3d.Aibo3DPick"; }
        //! returns port number the Java GUI should connect to
        virtual unsigned int getPort() const { return config->main.aibo3d_port; }

        static std::string getClassDescription() {
                char tmp[20];
                sprintf(tmp,"%d",config->main.aibo3d_port);
                return std::string("Listens to aibo3d control commands coming in from
port ")+tmp;
        }
        virtual std::string getDescription() const { return getClassDescription(); }

};


int aibo3dcontrollercmd_callback(char *buf, int bytes) {
  if (aibo3dControllerBehavior!=NULL)
    return aibo3dControllerBehavior->registerData(buf, bytes);
  return 0;
}
```