# Turk Framework

## version 0.1

**Rob O'Dwyer**

**May 07, 2010**

# Contents

# Welcome to Turk's documentation!

## Overview

### *What is Turk?*

Turk is a software framework for interfacing devices and communication protocols to web applications. It allows hardware creators and web application developers to work together to make reconfigurable, decoupled user interfaces for electronics and software. Turk uses an extension of the XMPP protocol to allow multiple web interfaces to send messages to custom drivers, which then talk to the hardware or application service. These drivers are run automatically through the framework, and can be started, stopped, reconfigured, or subscribed to by many web applications at once. Turk also includes a growing library of optional services that allow drivers to interface with hardware, web APIs, desktop applications, and home media.

### *TODO: Include overview diagram of Turk system here.*

### *What Can I Use It For?*

Turk is an ideal solution for creating a creative web interface to a device that can be accessed all over the world. Turk is probably not a good solution for applications that require high-speed communication, military-grade security, or are highly safety-dependent. However, that still leaves a large number of potential applications, such as:

- Controlling lighting and sound equipment
- Adding a web interface to an electronic device
- Monitoring sensors and power usage remotely
- **Connecting two devices or applications:**
    - Control your lights with your clock time
    - Lock the fridge door with your bathroom scale
    - Change lamp color to indicate temperature outside

### *What Are The Design Goals?*

The main goals for for Turk and its future development and features are:

- Simple extendability with drivers and an open API
- Platform independence
- Complete de-coupling of the interface between devices/hardware/protocols and the web
- Making the most common use-cases of the framework the simplest to implement
- Encouraging and simplifying collaboration between developers using different programming languages, frameworks and technologies

Since the Turk framework exists to simplify the task of reliably connecting a device or application to web interfaces, most of these goals can be summarized as follows: Turk should be simple to learn, use, and extend. Making the framework over-complicated or difficult to use would mostly defeat the purpose of using it.

## Who Should Use It?

# Getting started with the Turk Framework

## Installing

Turk is a package on the Python Package Index (http://pypi.python.org/pypi), and can be installed using the easy_install or pip command-line tools:

```
$ easy_install Turk

$ pip install Turk
```

It can also be installed from source, by downloading the latest source archive from a variety of places and using the included setup script:

```
$ tar xvzf Turk-0.1.1.tar.gz
$ cd Turk-0.1.1/
$ [sudo] python setup.py install
```

Needless to say, Python needs to be installed to do this. Turk works with any version in the 2.5-2.6 series, which are installed by default on many platforms. The other main dependencies are the D-Bus library and its python bindings (usually called python-dbus). These can be obtained from freedesktop.org.

## Setting up the configuration

Setting up a customized configuration file is a very important step. This file is not only used to configure the framework to work properly on your system, but also to organize drivers to be run and servers to connect to. It's written in YAML [1], and can be placed anywhere on the system.

### Why use YAML?

According to the YAML website:

> YAML™ (rhymes with “camel”) is a human-friendly, cross language, Unicode based data serialization language designed around the common native data types of agile programming languages. It is broadly useful for programming needs ranging from configuration files to Internet messaging to object persistence to data auditing.

Many other data formats are used for software configuration, including XML, INI files, and programming languages themselves. However, YAML has two significant advantages, namely that it is one of the most human-readable formats, and that it maps well to many logical data types. This allows the configuration file to be easily edited by both users and software.

### Example Configuration

The configuration file has a set of known defaults, which means an empty file will result in a set of reasonably sane values being used. However, these values may change between versions, and shouldn't be relied upon for important settings. A safe method is to start with a configuration that specifies all values as the default, and change them as necessary. That way, when the software is upgraded, only new values will be unspecified, and should be set to non-disruptive values (i.e. new features will be turned off by default).

The following is a basic template to start off with:

```
# Global configuration
global:
    bus: SessionBus

# control interface and launcher (corectl.py)
turkctl:
```

```
    pidfile: '/var/run/turk.pid'
    debug: True

# bridge (handles XMPP relaying)
bridge:
    server: macpro.local
    port: 5222
    username: platform@macpro.local
    password: password
    debug: True

# spawner (starts/stops and manages drivers)
spawner:
    # Location of drivers (prefixed to all driver filenames)
    drivers: /usr/share/turk/drivers

    # Add drivers that should be automatically started here along with their
    # environment variables and command line arguments
    autostart: [
        {'device_id': 1, 'filename': 'tick.py', 'env': {}, 'args': []},
        #{'device_id': 2, 'filename':'rgb_lamp.py', 'env':{'DEVICE_ADDRESS':'0xFF'}, 'args':
    ]
    debug: True

# xbeed (handles XBee communication)
xbeed:
    name: xbee0
    port: '/dev/ttys8'
    baudrate: 9600
    escaping: True
    debug: True
```

## Writing a simple driver

Although the framework comes with drivers for some simple tasks such as fetching the current date and time, and controlling simple wireless devices, most projects will need their own custom drivers.

Drivers are meant to be a way of translating the XML protocol used by Turk applications into another protocol, using a network or serial interface, or a web API. A web application can send XMPP messages to a predefined JID [2], and the framework will forward those messages to the correct driver. The drivers can send out their own messages, and any number of applications can subscribe to these updates.

Drivers are usually started by adding a listing to the configuration file that specifies the location of the file to run, any environment variables or command-line arguments it needs, and a unique identification number, or "device ID". This ID represents the abstracted "device" that the driver controls, and allows multiple drivers of the same type to be run at once. An example of such a listing can be seen in the sample configuration file above, in the autostart section.

Once started, communication between the driver and the rest of the framework is done through the D-Bus protocol. This allows drivers to use other services in the framework through remote method calls, and to receive messages through signals. For more information on how D-Bus method calls and signals work, read this introduction to D-Bus.

### Example Driver

The following is an example of a simple self-contained driver, written in Python. It uses both the Bridge API to receive updates from applications, and the XBee service to send binary packets to a wireless device.

```
#! /usr/bin/env python
import gobject
```

```
import dbus
import dbus.mainloop.glib
from turk.xbeed import xbeed
from xml.dom.minidom import parseString
import turk

class RGBLamp(dbus.service.Object):
    def __init__(self, device_id, device_addr, bus):
        """ Initializes the driver and connects to any relevant signals """
        dbus.service.Object.__init__(self, bus, '/Drivers/RGBLamp/%X' % device_addr)
        self.device_id = device_id
        self.device_addr = device_addr
        self.bus = bus

        # Get proxy for XBee interface
        self.xbee = self.bus.get_object(xbeed.XBEED_SERVICE, xbeed.XBEED_DAEMON_OBJECT % 'xb

        # Register update signal handler
        listen = '/Bridge/Drivers/%d' % (self.device_id)
        self.bus.add_signal_receiver(handler_function=self.update,
                                     bus_name=turk.TURK_BRIDGE_SERVICE,
                                     signal_name='Update',
                                     path=listen)

    def update(self, driver, app, xml):
        """ Called every time an update for this driver is received. """
        try:
            tree = parseString(xml)

            command = tree.getElementsByTagName('command')[0]
            ctype = command.getAttribute('type')

            if ctype == 'color':
                # Parse hex color into RGB values
                color = command.childNodes[0].nodeValue.lstrip('# \n\r')
                red, green, blue = [int(color[i:i+2], 16) for i in range(0, 6, 2)]

                # Build a message of the form "[RGB]"
                msg = ''.join(['[', chr(red), chr(green), chr(blue), '#]'])

                # Send it to the device
                self.xbee.SendData(dbus.ByteArray(msg), dbus.UInt64(self.device_addr), 1)

            elif ctype in ['on', 'off', 'shift', 'noshift']:
                command_byte = {
                        'on' : '@',
                        'off' : '*',
                        'shift' : '$',
                        'noshift' : '|' }[ctype]
                msg = ''.join(['[\x00\x00\x00', command_byte, ']'])
                self.xbee.SendData(dbus.ByteArray(msg), dbus.UInt64(self.device_addr), 2)

        except Exception, e:
            # emit an error signal for Bridge
            self.Error(e.message)

    def run(self):
        """ Loops forever and waits for signals from the framework """
        loop = gobject.MainLoop()
        loop.run()
```

```
    @dbus.service.signal(dbus_interface=turk.TURK_DRIVER_ERROR, signature='s')
    def Error(self, message):
        """ Called when an error/exception occurs. Emits a signal for any relevant
            system management daemons and loggers """
        pass

# Run as a standalone driver
if __name__ == '__main__':
    import os
    device_id = int(os.getenv('DEVICE_ID'))
    device_addr = int(os.getenv('DEVICE_ADDRESS'), 16)
    bus = os.getenv('BUS', turk.get_config('global.bus'))
    print "RGB Lamp driver started... driver id: %u, target xbee: 0x%X" % (device_id, device
    dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
    driver = RGBLamp(device_id, device_addr, getattr(dbus, bus)())
    driver.run()
```

There are more examples of drivers included with the framework written in several other languages, including Ruby, Java, and C++. As D-Bus has bindings for most commonly used programming languages, this allows developers to leverage already-written code or libraries to write their drivers. The main limitation of this approach is the relative lack of support for the Windows platform, as there is currently no stable port available. However, this situation should change relatively soon, as the project is still under active development.

For more detail about writing drivers and the APIs available to them, see the *Advanced Driver Design* section.

## Writing and deploying a web application

Creating a web application that uses Turk is even simpler, as they just send simple XMPP messages to the framework, and only need to be able to process HTTP POST requests. An application's work-flow looks something like this:

- Application sends a "register" XMPP message to the Turk platform when activated, to subscribe to any updates from a specified set of drivers.
- Application sends "update" XMPP messages to the platform on input from the user, and they are automatically forwarded to the relevant drivers.
- Driver sends a new "update", and the framework translates it into a HTTP POST to the application.
- Application is re-activated by the POST, and can choose to send a message back to the platform through XMPP.

### How does it work?

The important concept to understand here is that the communication from application to driver is done through XMPP, whereas drivers send messages back through HTTP POST requests. Although the content uses the same XML-based protocol, the transport is different. This is necessary due to the nature of the client-server model used by most web applications. The web application can't actively listen for XMPP messages, thus requiring HTTP requests to "wake it up", and the Turk platform likely isn't listening on a known internet address, requiring XMPP messages to "push" data to it.

## Implementation

The two main difficulties involved in designing an application lie in sending the XMPP messages and determining when the application registers itself to the platform. XMPP messages can be easily sent server-side using a variety of available libraries for languages such as PHP, Python, Ruby and Java. Depending on the XMPP server used, there are also ways of sending client-side messages using Javascript and AJAX requests. Some XMPP servers, such as ejabberd and OpenFire, support an extension called BOSH (Bidirectional-streams Over Synchronous HTTP), which enables applications to use XMPP through HTTP requests.

The following shows a simple example of sending an XMPP message with PHP and the XMPPHP library:

```
<?php
include 'XMPPHP/XMPP.php';

$platform = "turk-platform-account@xmpp-server.tld";
$driver_id = 8;
$conn = new XMPPHP_XMPP('xmpp-server.tld', 5222, 'turk-app-account', 'password', 'xmpphp');

try {
    # Connect to server and indicate presence
    $conn->connect();
    $conn->processUntil('session_start');
    $conn->presence();

    # Build update message containing simple XML command
    $msg = '<message xmlns="jabber:client" to="'.$platform.'">';
    $msg .= '<update xmlns="http://turkinnovations.com/protocol" to="'.$driver_id.'" from="0
    $msg .= '<command type="on" />';
    $msg .= '</update>';
    $msg .= '</message>';

    # Send message and close the connection
    $conn->send($msg);
    $conn->disconnect();
} catch(XMPPHP_Exception $e) {
    die($e->getMessage());
}
?>
```

Running this script on your server will send the following XMPP message to turk-platform-account@xmpp-server.tld (the XMPP JID the platform is using).

```
<message xmlns="jabber:client" to="turk-platform-account@xmpp-server.tld">
    <update xmlns="http://turkinnovations.com/protocol" to="8" from="0">
        <command type="on" />
    </update>
</message>
```

In this case, the "update" stanza is interpreted by the Turk framework as a request to forward data to a driver. The "to" attribute holds the ID of the driver to send it to. The "command" stanza (and anything else inside the update) is custom data for the driver to receive, and can be anything, including text or binary data, as long as it is properly escaped or encoded as XML.

For the application to receive data back from the driver, it needs to provide the platform with a URL to connect back to. Subscribing to a driver's updates is done by sending a "register" to the platform.

```
<message xmlns="jabber:client" to="turk-platform-account@xmpp-server.tld">
    <register xmlns="http://turkinnovations.com/protocol" app="2" url="http://example.com/up
        <driver id="8" />
    </register>
</message>
```

This notifies the platform that any updates from driver #8 should be sent to "http://example.com/updates/" as an HTTP POST request. The data from the driver will be wrapped up in a "update" stanza, with the "to" and "from" fields automatically filled in.

```
POST /update/ HTTP/1.1
Host: example.com
User-Agent: TurkFramework/0.12
Content-Type: application/xml; charset=utf-8
Content-Length: 268

<?xml version="1.0" encoding="utf-8"?>
<update xmlns="http://turkinnovations.com/protocol" to="0" from="8">
    <status>OK</status>
</update>
```

### *Deploying an Application*

Although applications can send messages to a Turk platform from anywhere in the world through the internet, they need to have a well-known, publically visible URL for Turk to send updates back. To get around this limitation, there will most likely be an update to the protocol allowing applications to register their XMPP JID (e.g. "turk-app-account@xmpp-server.tld") instead of a URL, so that they will receive updates as XMPP messages. This will also be useful for client-side Javascript applications, as they will be able to send and receive data from the platform without involving the web server at all!

However, the most common usage is to have the application hosted somewhere on the web, with server-side scripts doing the XMPP and HTTP processing. This is the simplest method, and allows the application to store semi-permanent state information about the drivers it controls.

---

1 YAML: YAML Ain't Markup Language (see yaml.org)

2 JID: Jabber ID, a unique identifier for a user on an XMPP server. Structured like an email address (username@host.tld)

## Configuring Turk

### *Setting up Drivers*

The Turk configuration file is designed to allow for maximum flexibility when running drivers. Driver processes can actually be launched separately from the other services and still function perfectly fine; the Spawner service just makes this much simpler by both managing the drivers and passing them information about the rest of the framework. Executables can be passed information in two ways, command-line arguments and environment variables. These are specified in the driver's configuration entry in the "args" and "env" settings.

### *Which is better?*

It is recommended to use environment variables to pass anything other than a variable-length list of data (e.g. a list of RSS feeds that a driver will fetch) to a Turk driver. They are much simpler to fetch and parse in most programming languages, and tend not to suffer from as many formatting and semantic errors as command-line arguments. Command-line arguments are difficult to use because they are essentially a list of words passed to the program that need to have some kind of meaning assigned to them. They require strict checking to make sure invalid input is not accepted by the program. Environment variables, on the other hand, are keyed by name, making them ideal for passing settings and simple variables to a program. An example of this in the real world can be found in the CGI standard, which used environment variables to pass information from web requests to server scripts. Although somewhat outdated, this standard was the most common way of producing dynamic content on the web for a long time.

## Unique Identifiers

The other important consideration for launching a set of drivers is the Device ID. This number uniquely identifies a driver instance on a platform, and can be used to ensure that a driver is not started twice by accident. The ID is represented as a 64 bit (8 bytes) unsigned integer, which means there are virtually unlimited numbers of Turk devices on one platform. One recommended practice is for the driver to claim a D-Bus service name that includes the ID somehow on startup. The D-Bus daemon will automatically enforce the requirement that these names be unique, and the driver can gracefully shut down instead of causing potential problems to an existing driver process.

An example of this:

```
import dbus, os

# The device ID is passed automatically as an environment variable
device_id = int(os.getenv('DEVICE_ID'))

# Get the Session or System bus
bus = getattr(dbus, os.getenv('BUS'))()

# Register an object
dbus.service.Object(bus, '/Drivers/MyDriver/%d' % device_id)
```

## The Driver List

Drivers are typically started in the order they appear in the configuration file, after a brief delay to ensure that the rest of the Turk services are running. Since many programs have a short delay before becoming fully operational, it is probably a good idea to carefully consider the order in which the drivers are started. If a service, either from a lower-level plugin or just another driver, is not available, the program must be able to handle this gracefully. A reasonably effective way of handling this is to keep trying to access the required services, with a short delay every time to avoid slowing down the entire system.

```
# Started first, since wall_display and robot_arm communicate with it
{'device_id': 1, 'filename': 'remote_control', 'env': {}, 'args': []},

# Order doesn't matter for these, since they don't connect at all
{'device_id': 2, 'filename': 'wall_display', 'env':{'DEVICE_ADDRESS':'0x13A2004052DADD'}, 'a
{'device_id': 3, 'filename': 'robot_arm', 'env':{'DEVICE_ADDRESS':'0x13A2004052DA9A'}, 'args
```

## All Settings

The following is a list of all the possible settings currently available in the Turk configuration file. Most of these settings are reasonably permanent, and can be relied on by developers to stay the same. Experimental features will be usable in various releases of the framework, but may disappear or change significantly, so use them at your own risk.

- **global: These settings apply to all services**
    - bus: The D-Bus bus that all services and drivers connect to (SessionBus or SystemBus)
- **turkctl: Options for the Turk launcher utility**
    - pidfile: A file used to store the process ID of the main Turk process. turkctl uses this to stop/start the framework.
    - debug: Controls the amount of information when running this utility

- **bridge: Options for XMPP communication**
    - server: macpro.local
    - port: 5222
    - username: platform@macpro.local
    - password: password
    - debug: Set to True to log additional information about XMPP messages
- **spawner: Options for driver spawning and management**
    - autostart: [
    - drivers: /usr/share/turk/drivers
    - debug: Set to True to see additional information about drivers being started and stopped.
- **xbeed: Options for XBee radio interface. Leave this section out to disable, if you aren't using one.**
    - name: The name of this daemon. Used by drivers to contact this specific instance. May be used to allow multiple instances of this service in the future.
    - port: The serial port that the radio module is connected to (e.g. /dev/ttyS0)
    - baudrate: The baudrate the radio is configured at (e.g. 9600)
    - escaping: Set to True if this module has the escaping option on (see manual for more details)
    - debug: Set to True to see additional information about XBee communication

# Driver Connectivity

## *Advanced Driver Design*

The examples of simple drivers given in the *getting_started* section are effective for establishing simple communication between a device or service and a web application, but have some major limitations. Due to the latency of the Internet and various other factors, there is an upper limit on the speed at which messages between drivers and applications can be sent. However, there are many workarounds for this limitation. Messages from applications are primarily meant to be used to update the current state of a driver, not to stream large amounts of real-time data. With this in mind, the ideal way is to instruct the driver to connect independently to a server, by giving it the necessary details in a message.

Sending data between two drivers with the application as the intermediary is another problem. Although the simplest way to do this is to relay the data through the application and back down to the driver, this is far from ideal. If there is minimal translation of the data involved, the drivers should be able to communicate directly with each other.

## *Interconnecting Drivers and Apps*

A basic outline of the process of connecting a driver to an external service is as follows, using RSS feeds as a simple example of an web service.

- Application registers to updates from driver:

```
<message xmlns="jabber:client" to="turk-platform-account@xmpp-server.tld">
    <register xmlns="http://turkinnovations.com/protocol" app="2" url="http://example.com/up
        <driver id="8" />
    </register>
```

```
</message>
```

- User adds a new feed to be monitored to application
- Application notifies driver that it should start checking a new feed:

```
<message xmlns="jabber:client" to="turk-platform-account@xmpp-server.tld">
    <update xmlns="http://turkinnovations.com/protocol" to="100" from="101">
        <feed type="atom" update="1min">
            <title>Digg News</title>
            <url>http://feeds.digg.com/digg/news/popular.rss</url>
        </feed>
    </update>
</message>
```

- Driver starts fetching the RSS content from the URL every 1 minute
- When a new story is found, the driver sends out an update addressed to the app:

```
<message xmlns="jabber:client" to="turk-platform-account@xmpp-server.tld">
    <update xmlns="http://turkinnovations.com/protocol" to="101" from="100">
        <story type="atom" date="1273212547">
        <title>Baboon attacks Prime Minister! Nation in panic.<title>
        <author>Lois Lane</author>
        </story>
    </update>
</message>
```

Connecting two drivers together is even simpler, as all that is needed is a D-Bus address and path. This allows one driver to make RPC calls to the other, without going through the overhead of the XMPP handlers and the internet communication.

```
<message xmlns="jabber:client" to="turk-platform-account@xmpp-server.tld">
    <update xmlns="http://turkinnovations.com/protocol" to="101" from="100">
        <driver type="dbus" method="Update">
            <address>org.turkinnovations.exampledriver</address>
            <path>/Drivers/ExampleDriver/42</path>
        </driver>
    </update>
</message>
```

## Driver Dependencies

A common issue with setting up a Turk system arises when drivers are dependent on each other to run properly. Drivers can be reused as services to maximize their capabilities, and can also be connected directly to each other for more efficient communication. However, there are several important things to keep in mind when implementing a system like this.

Another issue that can arise with dependencies is fault tolerance. Unfortunately, even the best drivers can have software bugs, and occasionally, they crash. The framework is designed to deal with this, however, and will automatically restart a driver when it detects a problem like this, unless configured otherwise. However, other drivers must be able to handle any connections being dropped or data lost in the process. Depending on the method of communication used, other drivers may need to re-open the connection when the program at the other end is restarted. DBus proxy connections, for example, are based on the unique name of the connection. This means that if a driver was using a reference to another driver, received from a controlling application, that reference would be useless once the other driver was restarted. The same principle as before applies to this situation - the driver has to monitor the connection, and take steps to fix any communication problems when they occur.

## Running Drivers on Demand

The simplest way to make sure a required driver is running is to include in the configuration file and restart the framework. However, there are some use cases for starting and stopping drivers on the fly. Examples of these are:

- Multi-driver systems that can turn off certain features when not in use
- Utility drivers that other drivers can start up when needed
- Controlling the Turk Framework from a graphical user interface

As a result, there is a way to do this through the Turk API. The API calls StartDriverByName, RestartDriverByID, StopDriverByID and GetDriverList can be used to control the Spawner component. This allows drivers and plugins to be written that can dynamically control the list of drivers currently being managed by the framework.

# Designing Web Interfaces

## Speed and Latency Issues

The simplest way to send and receive messages through a web application (using a server-side script) has some limitations that become obvious fairly quickly. Even with a fast computer and internet connection, the time it takes to make a new connection to an XMPP server is usually several seconds. Once the connection is ready, however, messages can be sent very quickly. This means that there is a huge performance advantage for an application to stay connected to the server. There are two ways to achieve this:

- Start a daemon process on the server that holds a connection, and send messages through it using some kind of RPC call.
- Make a client-side connection with Javascript, and occasionally synchronize with the server-side code if necessary.

Each have their own pros and cons. The daemon process method is more complicated, and adds another layer of complexity to the application, but can be re-used for other applications, and has a much wider range of uses. The Javascript solution, on the other hand, is trivial to set up, but requires a web browser to operate. In other words, it can't do anything in the background, because it requires the user interface to be "running".

Another downside to the Javascript solution is the problem of cross-site scripting. Most modern web browsers prevent client-side scripts from connecting to other domains or web services. This causes a problem for XMPP client scripts, since the server they connect to is frequently located on a different domain or port. Common workarounds for this include running a web proxy that forwards any requests to the XMPP server, and using Flash to provide an alternate means of communication.

## Connecting Web Apps to Platforms

Once you have a Turk platform running, and a couple of applications to use with it, the next step is to reliably connect them together. Depending on the usage, it may be a good idea to use a separate XMPP account (JID) for each application. XMPP servers have different policies regarding multiple clients using the same account, and some may refuse access in such a situation. It is also possible to connect as a specific *resource*, which allows for multiple connections to the same JID, as long as the resource names are different.

The next step is to work out the mechanics of receiving updates from drivers. Applications need to have a URL for the platform to send updates to, and have to register to receive updates with the drivers they use. This can be done whenever the UI is opened, or just on a regular interval. The framework will ignore any identical register requests, so it is better to err on the side of caution.

# The Turk XMPP Protocol

## Current Protocol Specification

The Turk XMPP protocol currently consists of three tags that can be sent to and from a platform:

- **<update />**
  The update tag is used to carry messages from applications to drivers. It must specify the driver ID to deliver the message to.
- **<register />**
  The register tag is sent by an application to subscribe to updates from a particular driver.
- **<require />**
  The require tag asks the framework to ensure that a required driver is running. This currently does nothing, but will eventually be used in the driver launching part of the framework.

These tags are all under the XML namespace "http://turkinnovations.com/protocol", and are sent inside jabber:client message tags like most XMPP chat messages.

## Protocol Examples

### Registering to a driver

```
<message xmlns="jabber:client" to="turk-platform-account@xmpp-server.tld">
    <register xmlns="http://turkinnovations.com/protocol" app="2" url="http://example.com/up
        <driver id="8" />
    </register>
</message>
```

### Sending an update

```
<message xmlns="jabber:client" to="turk-platform-account@xmpp-server.tld">
    <update xmlns="http://turkinnovations.com/protocol" to="8" from="0">
        <command type="on" />
    </update>
</message>
```

### Requiring a driver

```
<message xmlns="jabber:client" to="turk-platform-account@xmpp-server.tld">
    <require xmlns="http://turkinnovations.com/protocol" app="2">
        <driver id="8" />
    </require>
</message>
```

## Message Structure

### UPDATE

The update tag must include the "to" and "from" attributes, indicating the destination driver and source application IDs, respectively. It may also include an optional "type" attribute, which can be used to indicate the context of the update. This may be used in a future version of the framework.

### *REGISTER*

The register tag must include the "app" and "url" attributes. The first is used by the framework to identify the app, and the second to forward messages using HTTP. Drivers are specified with a list of "driver" tags, each of which must include an "id" attribute representing the driver ID.

### *REQUIRE*

The require tag must include the "app" attribute, which has the same meaning as it does for the register tag. Drivers are specified with a list of "driver" tags, each of which must include an "id" attribute representing the driver ID.

## *Future Developments*

There are several planned updates to the protocol, which will probably be indicated by version numbers in the XML namespace. This will ensure compatibility by allowing newer applications to bundle newer protocol features into their messages that will be ignored by older versions of the framework. Some of these features include:

- Allowing other services besides drivers to be specified in register/require tags
- Adding a "protocol" attribute to register, which determines the method of notifying the application (e.g. HTTP, HTTPS, XMPP)
- Allowing register/require tags to specify drivers by name instead of ID
- Commands to control drivers (restart, stop)
- A "status" tag that will request an update containing a bundle of information about the framework

# System Services

## *Services for Connecting Drivers*

- MIDI Services
- XBee Daemon(s)
- Extensions

## *MIDI Services*

Nullam elementum erat. Quisque dapibus, augue nec dapibus bibendum, velit enim scelerisque sem, accumsan suscipit lectus odio ac justo. Fusce in felis a enim rhoncus placerat. Cras nec eros et mi egestas facilisis. In hendrerit tincidunt neque. Maecenas tellus. Fusce sollicitudin molestie dui. Sed magna orci, accumsan nec, viverra non, pharetra id, dui.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nullam placerat mi vitae felis. In porta, quam sit amet sodales elementum, elit dolor aliquam elit, a commodo nisi felis nec nibh. Nulla facilisi. Etiam at tortor. Vivamus quis sapien nec magna scelerisque lobortis.

Curabitur tincidunt viverra justo. Cum sociis natoque penatibus.

## *XBee Daemon(s)*

Proin neque elit, mollis vel, tristique nec, varius consectetuer, lorem. Nam malesuada ornare nunc. Duis turpis turpis, fermentum a, aliquet quis, sodales at, dolor. Duis eget velit eget risus fringilla hendrerit. Nulla facilisi. Mauris turpis pede, aliquet ac, mattis sed, consequat in, massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Etiam egestas posuere metus. Aliquam erat volutpat. Donec non tortor. Vivamus posuere nisi mollis dolor. Quisque porttitor nisi ac elit. Nullam tincidunt ligula vitae nulla.

Vivamus sit amet risus et ipsum viverra malesuada. Duis luctus. Curabitur adipiscing metus et felis. Vestibulum tortor. Pellentesque purus. Donec pharetra, massa.

### *Writing Services*

Malesuada elementum, nisi. Integer vitae enim quis risus aliquet gravida. Curabitur vel lorem vel erat dapibus lobortis. Donec dignissim tellus at arcu. Quisque molestie pulvinar sem.

Nulla magna neque, ullamcorper tempus, luctus eget, malesuada ut, velit. Morbi felis. Praesent in purus at ipsum cursus posuere. Morbi bibendum facilisis eros. Phasellus aliquam sapien in erat. Praesent venenatis diam dignissim dui. Praesent risus erat, iaculis ac, dapibus sed, imperdiet ac, erat. Nullam sed ipsum. Phasellus non dolor. Donec ut elit.

Sed risus.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Vestibulum sem lacus, commodo vitae, aliquam ut, posuere eget, dui. Praesent massa dui, mattis et, vehicula.

### *Alternate Protocols and Transports*

Justo ac sem.

Pellentesque at dolor non lectus sagittis semper. Donec quis mi. Duis eget pede. Phasellus arcu tellus, ultricies id, consequat id, lobortis nec, diam. Suspendisse sed nunc. Pellentesque id magna. Morbi interdum quam at est. Maecenas eleifend mi in urna. Praesent et lectus ac nibh luctus viverra. In vel dolor sed nibh sollicitudin tincidunt. Ut consequat nisi sit amet nibh. Nunc mi tortor, tristique sit amet, rhoncus porta, malesuada elementum, nisi. Integer vitae enim quis risus aliquet gravida. Curabitur vel lorem vel erat dapibus lobortis. Donec dignissim tellus at arcu. Quisque molestie pulvinar sem.

Nulla magna neque, ullamcorper tempus, luctus eget.

## Roadmap

### *Planned Features*

Eleifend nisi et nibh. Maecenas a lacus. Mauris porta quam non massa molestie scelerisque. Nulla sed ante at lorem suscipit rutrum. Nam quis tellus. Cras elit nisi, ornare a, condimentum vitae, rutrum sit amet, tellus. Maecenas a dolor. Praesent tempor, felis eget gravida blandit, urna lacus faucibus velit, in consectetuer sapien erat nec quam. Integer bibendum odio sit amet neque. Integer imperdiet rhoncus mi. Pellentesque malesuada purus id purus. Quisque viverra porta lectus. Sed lacus leo, feugiat at, consectetuer eu, luctus quis, risus. Suspendisse faucibus orci et nunc. Nullam vehicula fermentum risus. Fusce felis nibh, dignissim vulputate, ultrices quis, lobortis et.

### *Other Projects in Development*

Velit eget risus fringilla hendrerit. Nulla facilisi. Mauris turpis pede, aliquet ac, mattis sed, consequat in, massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Etiam egestas posuere metus. Aliquam erat volutpat. Donec non tortor. Vivamus posuere nisi mollis dolor. Quisque porttitor nisi ac elit. Nullam tincidunt ligula vitae nulla.

Vivamus sit amet risus et ipsum viverra malesuada. Duis luctus. Curabitur adipiscing metus et felis. Vestibulum tortor. Pellentesque purus. Donec pharetra, massa quis malesuada auctor, tortor ipsum lobortis ipsum, eget facilisis ante nisi eget lectus. Sed a est. Aliquam nec felis eu sem euismod viverra. Suspendisse felis mi.

### *Support for Hardware Protocols/Standards*

Vestibulum enim felis, interdum non, sollicitudin in, posuere a, sem. Cras nibh.

Sed facilisis ultrices dolor. Vestibulum pretium mauris sed turpis. Phasellus a pede id odio interdum elementum. Nam urna felis, sodales ut, luctus vel, condimentum vitae, est. Vestibulum ut augue. Nunc laoreet sapien quis neque semper dictum. Phasellus rhoncus est id turpis. Vestibulum in elit at

odio pellentesque volutpat. Nam nec tortor. Suspendisse porttitor consequat nulla. Morbi suscipit tincidunt nisi. Sed laoreet, mauris et tincidunt facilisis, est nisi pellentesque ligula, sit amet convallis neque dolor at sapien. Aenean viverra justo ac sem.

Pellentesque at dolor non lectus sagittis semper. Donec quis mi.

### *Support for Web Protocols*

Et ultrices posuere cubilia Curae; Morbi urna dui, fermentum quis, feugiat imperdiet, imperdiet id, sapien. Phasellus auctor nunc. Vivamus eget augue quis neque vestibulum placerat. Duis placerat. Maecenas accumsan rutrum lacus. Vestibulum lacinia semper nibh. Aenean diam odio, scelerisque at, ullamcorper nec, tincidunt dapibus, quam. Duis vel ante nec tortor porta mollis. Praesent orci. Cras dignissim vulputate metus.

Phasellus eu quam. Quisque interdum cursus purus. In orci. Maecenas vehicula. Sed et mauris. Praesent feugiat viverra lacus. Suspendisse pulvinar lacus ut nunc. Quisque nisi. Suspendisse id risus nec nisi ultrices ornare. Donec eget tellus. Nullam molestie placerat felis. Aenean facilisis. Nunc erat. Integer.

## Indices and tables

- *Index*
- *Module Index*
- *Search Page*