Communications

June 17, 2014

Abstract

This document describes the future communications framework for Clockwork. A Clockwork driver program provides a runtime environment to execute several machines in parallel. Each driver may communicate with other programs via message passing and publisher/subscriber models.

1 Background

The Clockwork interpreters have several communication interfaces to support various control protocols, web server display, system monitoring and to provide a commandline interface. These interfaces have been introduced on the basis of project requirements and are inconsistent in design and implementation. This document addresses the various communications interfaces and describes legacy and future implementations for

- EtherCAT
- Modbus
- commanline
- Internet of Things
- web clients

2 Architecture

The latproc system is to be split into a communications hub with scripting engines and communications modules all interconnected by a shared memory model and messaging system. In the first instance, Redis and ØMQ provide the shared memory and messaging infrastructure. The shared repository provides access to shared static data, it may be implemented in a variety of database/memory systems but the key observation is that the repository is not expected to notify Clockwork instances of data changes.

On startup, a Clockwork driver

- interrogates a repository for
    - machine definitions

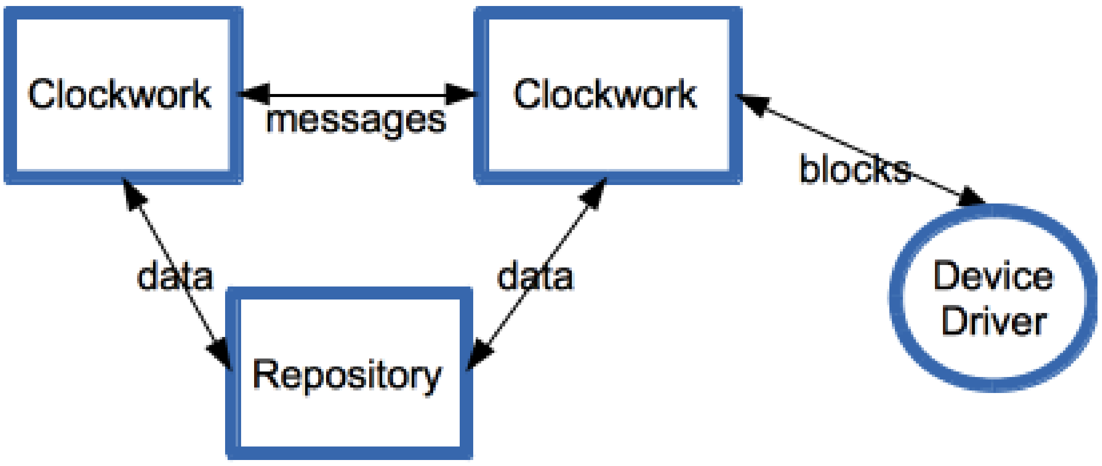

Figure 1: Clockwork messages and data sharing

  - interface definitions

- establishes connections as described in the interface definitions
- begins execution of the machines

Note that failure to establish communications does not cause an instance to fail its startup; it will continue to establish the connection until aborted by an external source. Communication interfaces can be interrogated within Clockwork and the channel state can be used to affect the execution of the models.

3 Interprogram communication

3.1 Interface Definitions

An interface between Clockwork drivers is introduced via the CHANNEL object:

```
channel_name CHANNEL(host:hostname, port:portname) {

    IDENTIFIER id_string ';'
    VERSION version_sequence ';'
    SHARES block_name, ... ';'
    MONITORS machine_name [ interface ], ... ';'
    UPDATES machine_name [ interface ], ... ;
    SENDS message_name, ... ';'
    RECEIVES command_name, ... ';'

}
```

A channel may specify a set of messages sent and commands received on the interface but there is no way to define the parameters of those messages or commands (TBD). The clauses are described as follows:

IDENTIFIER a pre-shared key. If the channel definition includes an IDENTIFIER option, both sides must be able to demonstrate that they know the identifier. The identifier itself should never be transmitted on the channel.

VERSION a numeric value that provides the version of the interface

SHARES lists the memory blocks that are shared between the client and the server. Each memory block must be known to both the server and client

MONITORS lists the machines that the server will send updates for. Each listed machine may also include an interface definition that further describes the states and properties of the machine.

UPDATES lists the machines that the client will send updates for. Each listed machine may also include an interface definition that further describes the states and properties of the machine.

SENDS lists the messages that will be accepted by the server.

RECEIVES lists the messages that will be accepted by the client.

Clauses in a channel definition are all optional.
By way of an example, a BIT has the interface:

```
BIT MACHINE {

    on STATE; off INITIAL;
    COMMAND toggle,turnOn,turnOff;

}
```

A channel definition is directional, how the clauses apply to the machine depends on where the machines are instantiated. A channel may also include templates that define the machines that are monitored or updated via the channel. The definition may describe the states to be exchanged, a state that is the startup or default state and the properties that are to be exchanged on the interface.

```
machine_class INTERFACE '{'

    state1 STATE ';'
    state2 STATE ';'
    initial_state INITIAL ';'
    STATES '{' state3, state4, ... '}'
    property1 PROPERTY [ READONLY | READWRITE ] ';'
    PROPERTIES '{' property2 [ READONLY | READWRITE ], ... '}'
'}'
```

If an interface definition is provided and includes any state definitions, only the states defined will be exchanged. Similarly if properties are included in the definition, only changes to the named properties will be exchanged.

For example, in the following definition, only the states 'on' and 'off' of the switch are communicated to the connecting machine.

```
switch INTERFACE { STATES { on, off } }
```

If the switch enters another state (eg., disabled, no message is sent to the listener and the switch is assumed to be still in the last state communicated. (TBD)

3.2 Shared Memory and sampling

Clockwork if fundamentally a program that looks at changes in a memory map and uses those changes to affect changes in a state model of various simulated machines. It should be possible to share any memory block with clockwork, no matter the source. A block may represent a device or it may be divided into regions that relate to the device in some way, for example in with Beckhoff EtherCAT(tm) the 'Module' refers to a physical device, in MODBUS, *coils* and *registers* refer to distinct memory regions.

Each block may have properties that descibe attributes common to all regions in the block

```
ModbusPanel BLOCK {

    X BIT[256] IN;
    Y BIT[256] OUT;
    V WORD[256] IN OUT;

}
panel ModusPanel
```

Given the above definition, Clockwork may bind fields of the block to a machine, for example:

```
start_button panel X1;
```

defines *start_ button* as a single bit object in clockwork, linked to coil X01 in the modbus memory block.

3.3 Issues

1. How do we manage the startup conditions? When a client connects? When the server restarts?
2. How large is a beckhoff master memory block?
3. How quickly can we send master block updates to a clockwork program. What latency can we expect?

3.4 Message Format

Communication between all components is via a structured message using the JSON format.

```
command string,
need_response bool /* is this used/needed? TBD */
parameters array [

    item {

        type B | BOOL | I | INTEGER | U | UNSIGNED | S | STRING | F | FLOAT | N | NAME
        value <depends on type>

    }

]
```

Note that names conform to clockwork naming rules: ALPHA [ALPHA NUMBER _] ...

3.5 Dynamic configuration

A channel specification needs to be read by both ends of the channel. To facilitate the use of runtime sampling systems, a channel specification can be sent by the client when the connection is made. The channel specification may include interface specifications (TBD).

3.6 Protocol

When a client connects to a server it is required to initiate a negotiation of the channel configuration for the connection. This can be done by a command message with the command: CHANNEL and the name of the channel. In this case, the server is expected to refer to its repository for a matching channel (and version. . .).

The communications protocol is not secure; it has a simple authentication process that will be upgraded in the future.

- If the server does not have a matching configuration, the client may send the configuration.
- If a channel cannot be agreed, the connection will be terminated.
- If an identifier is provided, it will be used as a pre-shared key to authenticate the connection. The server will send a keyword challenge to the client and the client will be expected to combine this with the secret identifier for the channel and send back an encrypted value. The client will also authenticate the server in the same way.
- If a version number is provided, both ends may emit a warning or disconnect, depending on other options that may be in force.

4 Examples

4.1 Weight sampling

Consider a weight sampling machine that is implemented with a remote (client) computer that communicates with the device and a master (server) computer that uses the sampled weight as part of its machine management.

4.1.1 Shared configuration

We define a channel for communication between the computers and define an interface that outlines the interesting states and properties of the device interface. Both the client and server:

```
    ScalesChannel CHANNEL scales {
        MONITORS scales ScalesInterface;
}
    ScalesInterface INTERFACE {
        PROPERTIES weight, status;
        stable STATE;
        unstable INITIAL;
}
```

4.1.2 Server configuration

Using this information, the master computer can define a machine that represents a remote scales machine conforming to the ScalesInterface and use this to implement some logic. The monitor uses the state of the remote system to determine whether the weight is valid but also demonstrates the use of a timer to 'debounce' the remote state change:

```
ScalesMachine ScalesInterface;
WeightMonitor MACHINE scales {

    waiting WHEN scales IS unstable;
    stablising WHEN scales IS stable AND TIMER < 100;
    overweight WHEN scales IS stable AND scales.weight >= 100;
    ready WHEN scales IS stable AND scales.weight >= 10;
    underweight WHEN scales IS stable;

}
remote_scales ScalesMachine;
monitor WeightMonitor remote_scales;
```

The host machine executes the above without needing any additional implementation details for the remote device but it needs to link the remote scales to the channel by instantiating the channel:

```
scales_channel ScalesChannel(port:9999) remote_scales;
```

4.1.3 Client configuration

The Clockwork driver that implements the scales defines a machine to interpret data from the hardware. Device connector will be used in this case to prepare the properties of this machine (not shown here).

```
SCALES MACHINE {
    OPTION weight 0;
    OPTION status ‘‘x’’;
    unstable WHEN status == ‘‘U’’;
    stable WHEN status == ‘‘S’’;
    unknown DEFAULT;
}
scales_machine SCALES;
scales_channel ScalesChannel(host:’’master’’, port:9999) scales_machine;
```

When the client driver starts, it attempts to instantiate the scales channel by making a connection to the host and port provided. The scales channel enters a ’connected’ state once communication is established. In the situation where both sides of the channel implement the scales the channel enters an ’error’ state.

4.2 System Monitoring

To setup a system monitor, a pattern can be used so that a range of machines can be selected without the need to name each machine. Consider an application that monitors a Clockwork instance to record state changes of a set of machines.

```
MonitorChannel CHANNEL {
    MONITORS ‘*‘;
}
```

Note that no parameters are passed to the channel and a wildcard is used to specify the machines to be monitored. The client instantiates the MonitorChannel and when the connection is established, the server transmits the current state and properties of all matching machines.

A wildcard can be associated with an interface, giving greater control of what is monitored:

```
MonitorChannel CHANNEL {
    OPTION Initialisation FALSE;
    MONITORS ‘*‘ MonitorInterface;
}
MonitorInterface INTERFACE {
    PROPERTIES ‘^[A-Z].*‘;
}
```

In the above example, the monitor interface specifies that only properties with names that begin wtih a capital letter will be sent. A similar specification can be used for transmitted states.

5 Implementation

At the client end, clockwork uses the channel object to establish and maintain a server connection. When the connection is setup, the current state of the machines in the ’UPDATES’ list are sent and the server is expected to also send the state of all the machines in the ’MONITORS’ list. When a connection breaks, the client sets the state of all ’MONITORS’ machines to their initial/default state.

At the server end, clockwork instantiates a machine for each entry in the ’UPDATES’ list and updates the state of this machine each time a message is received. This local instantiation shadows the state of the remote machine and can be used by other local machines as normal.

5.1 Implementation notes

- A MachineInterface object now subclasses MachineInstance
- A MachineInterface is a representative or ‘proxy’ for a MachineInstance
- MachineInterface objects are distinguished when setting up a channel and are not normally used as a source.
- MachineInterface objects can be used to propagate state to other instances of clockwork; this is done by registering a distance (hops) to the actual MachineInstance that they represent
- There are issues with setup and teardown of connections and how this effects messaging. TBD

6 Predefined Channels

6.1 EtherCAT

When the EtherCAT driver connects to Clockwork, it announces the modules that it has control over either by providing the names from the shared repository or by providing the definitions directly on the channel.

```
Module BLOCK {
    OPTION position 1;
    OPTION alias 0;
    pdo1 REGION {
        OPTION SM 3;
        OPTION index 0x1a00;
        OPTION name ‘‘Inputs’’;
        BIT[8];
    }
}
```

A Clockwork script can bind objects to fields:

```
EL1814 MODULE 1, 0; # bit 0 of module with position 1
```

The above legacy syntax is not general enough (TBD)

6.2 Modbus

Clockwork provides a facility to export properties, states and commands of machines to modbus. The current syntax provides for automatically generated addresses but is too tightly coupled to modbus. A more general approach is needed (TBD)

6.3 Persistence

7 Legacy Channels

This section is partial documentation of previous channel implementations

7.1 Monitor channel

This is the main channel by which state changes are published, default port 5557

message structure:
command := PROPERTY | STATE
need_response := FALSE
parameters := [{ N, <name> }, { <type>, <value> }]

7.2 Modbus channel

This channel is used to communicate with the modbus daemon, that daemon communicates to devices with the modbus protocol

message structure:
The following commands may be sent to the modbus daemon

STARTUP tells modbus to load the configuration and obtain initial values for all coils etc

DEBUG ON tells modbusd to emit debug information

DEBUG OFF tells modbusd to stop emitting debug information

UPDATE

7.3 Persistence channel

This is used to report changes of property values and state in persistent machines

Message structure

```
command string,
parameters array [
    item {

        type B | BOOL | I | INTEGER | U | UNSIGNED | S | STRING | F | FLOAT | N | NAME
        value <depends on type>
    }
    ...
]
```

7.4 Device connector

There is two way communication with device connector sending property commands and receiving property or state commands or messages.

Messages Device connector may simply read values and pass them to clockwork with no guarantee of delivery, these messages may be lost if something goes wrong at either end but for repetitively sampled data, this is not necessarirly a problem.

Requests are sent from clockwork to device connector sometime to be handled by device connector and sometimes to pass through to the device. The requests are to be formated as the device requires them and a reply is to be sent to clockwork, either from the device or from device connector. Since device connector does not know anything about the device, the request provides additional information.

Replies are sent from device connector to clockwork. If the request included a boolean option "ack_needed" (true), device connector simply sends "OK" when the request has been forwarded to the device. Otherwise, device connector forwards the request to the device and collects its response, using a pattern supplied in the request.

Commands are sent from device connector to clockwork. Commands require an acknowledgment and sometimes require data. Device connector can be configured to supply data matching a pattern via the PROPERTY command or STATE command. Subexpressions in the pattern will become additional parameters.

7.4.1 Messages

For data streaming, matched sub expressions are passed as parameters. ?? incomplete

Message structure:

```
DATA string
list_name name,
params array [ value, ...]
```

7.4.2 Requests

```
REQUEST string
id integer
params array [ value, ... ]
```

7.4.3 Replies

7.4.4 Commands

8 Messaging system

The messaging system attempts to deal with various interprocess issues, using zmq for the heavy lifting but also providing application features:

- Clients negotiating a private communication channel
- Dealing with replies

All applications include the notion that some data is transitory and some is not, transitory data is sent without regard to whether anyone is listening, this is used for repeated messages such as sensor data streaming. For non-transitory data, a request/response method is used.

9 Performance

This may need to be optimised, we could use a temlpate system; a property message looks as follows, where the user data is given in \$1, \$2 etc. Note that the value (\$3) item needs to be converted into a JSON string with the appropriate escape charaters but apart from that the template can be filled fairly quickly.

```
{ "command": "PROPERTY", "params":
[
{ "type": "NAME", "value": $1 },
{ "type": "NAME", "value": $2 },
{ "type": TYPEOF($3), "value": $3 }
] }
```