Xylem Pump Research Project Remote Access Report

Peter Spenler

I. Introduction

As part of a masters research project an experimental pump loop was constructed in a lab at the University of Guelph. As a component of this project a remote monitoring system was developed to allow the system to be monitored and controlled remotely.

II. Overview

The goal of this project is to create a system which can allow the remote monitoring and eventual control of an experimental pump testing setup (pump loop). The overarching project hopes to look at how pumps degrade over time and how that degradation affects performance. The system will allow researchers and interested parties to monitor the performance of the pump loop in real time, and in the future send control signals and return detailed recordings.

III. Design Concerns

This section outlines a number of concerns which controlled decisions made for the project.

A. Timing / Speed

Timing and speed of the system is incredibly important. Since the system is being used to perform detailed research of the pump loop's performance at high sample rates it's important that the system's measurements are accurate and not impeded by the rest of the system. This means making sure that neither writing the data to the disk nor transmitting data to the server affects the timing of the recorded data.

This also means that speed should be considered in the development of the client and the server application as well. Since the system's desired final use is for monitoring and control of the pump loop, it's important that the client and the server application introduce a minimal amount of latency into the system. Part of the decision to use Go for the server was its high performance as a compiled language, and it's focus on concurrency thus reducing the performance impact of many connections or high rates of data transfer.

B. Security

Since this system is being used for ongoing research, and especially because it will be able to send control signals to the research loop in the future, security is of utmost importance. When the system is able to control the loop, it is critical that only authorized users can access the system to prevent damage to the system, or tampering with the results. As a result considerations like user authentication, data encryption, and the prevention of man-in-the-middle attacks are important to consider throughout development.

C. Ease of Use

Ease of use is also important to consider for this system. Since the system is intended to be used by researchers who may not be very familiar with programming, the system should be easy to use in a production environment. This involves not only making each component's interface easy to understand and making error messages clear, but also ensuring that each component is simple to run and can generally handle errors on their own.

IV. System Breakdown

This section will outlay each component of the system, it's purpose, and it's function.

A. LabView Program

The LabView program facilitates the connection between the physical sensors and the server. The program is built of two loops: the Data Acquisition loop, and the Data Write and Transmit loop.

The LabView program uses a DAQ Assistant block running in the Data Acquisition loop to interact with the sensors. The DAQ Assistant allows the developer to easily manage all the sensors connected to the Data Acquisition Computer including ordering into a compound signal, and unit scaling (For converting sensors that read a voltage into meaningful units such as PSI or GPM). The DAQ Assistant outputs a single signal (a LabView "Dynamic Data Type" wire) which contains the data from each sensor compounded in the order created in the DAQ Assistant. This signal is then en-queued in a signal queue so that the data can be brought outside the Data Acquisition loop. Each packet which is en-queued in the signal queue contains a series of frames, which each contain one reading from each sensor. The number of frames in each packet is determined by the sample rate and number of samples set on the LabView interface.

The data from signal queue is then de-queued in the Data Write and Transmit loop. As the name implies this loop performs two functions. First this loop serves to save the recorded data to the disk. This is done using the Write to Measurement File block. This block is configured to write the data into a file with a programmed name followed by the next free sequential number (ex. it writes to "pump_data0.tdms", then "pump_data1.tdms", followed by "pump_data2.tdms", etc.). This block is also configured to write to this file using National Instrument's (NI's) Technical Data Measurement Streaming (TDMS) file format. This file format is explicitly designed to be used by LabView for writing data with a high poll rate to the disk. These TDMS files can then be opened and converted to excel spreadsheets using an excel plugin which comes bundles with LabView. Finally since the system is only periodically used for running tests, file saving does not need to be a continuous function. Thus a series of boolean blocks are used to trigger an if-else container so that data is only saved when recording has been enabled. The design has been implemented so that the user can press the "Start Recording" button on the LabView interface to begin having data written to the disk. The system will then continue writing data to the

disk only until the time specified in the ”Time to Record” box has been elapsed. If the user wants to stop recording data earlier they can press the ”Stop Recording” button.

The second function of the Data Write and Transmit loop is transmitting the read data to the server. Since this data is only being used for real time monitoring and not experimentation, it does not need to be as comprehensive or precise as the data being written to the disk. As such, only the most recent frame in the packet is taken for transmission. This frame is then converted to a double array and sent to the custom ”Array to JSON” block. This block is configured to only take a double array of a certain size. If the number of sensors being used changes then this block needs to be modified to reflect that change. The ”Array to JSON” then outputs a formatted string. In the current implementation the program this block’s title is a misnomer as the output string is not actually in the JSON format. This string is then appended to the address for the server’s data reception API. This final concatenated string is then used as the address for a GET request to the server, which runs outside of the university’s network (it can be run within the university’s network, but this will prevent the server from being accessible from outside the universities network).

B. Main Server

The main server component acts as the backbone of the system. It takes data from the LabView program and distributes it to all connected clients. It also implements authentication so that only authorized users can access the data and, in future implementations, send commands to the system. The server can be defined as three components: Data Receiver, Data Transmitter, and Authenticator.

The first component is the Data Receiver. This component handles receiving data from the LabView program. The LabView program makes a POST request to the server at ”SERVERIP/data/submit” where ”SERVERIP” is the ip address or domain name of the server including the port. By default the server runs on port 4000, but this can be changed in the code. The POST request from the LabView program contains two form fields: ”key” and ”data”. The ”key” field contains a secret key which identifies the LabView program so that only the authentic LabView program can send data to the server. The ”data” field contains the sensor data in a string formatted like this: ”SRC$${float data}$${float data}$${float data}”. Each ”{float data}” in the sample string is where each float sensor reading goes. The string can have an unlimited number of sensor readings, but each must be delimited by ”$$”. The component then calls handleLVData() with the string to start the transmission process.

The next component of the server is the Data Transmitter. This component takes the string transmitted by the LabView program and transmits it to all the authenticated clients using WebSockets. First the formatData() function takes the string and turns it into a slice of floats containing each sensor reading. Then the buildArray() function is used to convert the float slice into a string containing the sensor data in a JSON array. A simple improvement to the system would be to have the LabView program simply send the data in the JSON array format, thus eliminating the need for formatData() without making LabView perform any more string operations. Finally this array is placed into a JSON object. This object has two name object pairs: ”measurement” and ”error”. The ”measurement” pair holds the array of sensor data. The ”error” pair holds a string containing either a relevant error message or ”none” if there is no error. This JSON object is then transmitted using WebSockets to all clients whose ”auth” flag is set to true (i.e. has an authorized connection).

Finally the server also handles authentication. This is performed with a few simple steps. First the user attempts to log into the system. To do this the client makes a POST request to ”SERVERIP/api/login” where ”SERVERIP” is the ip address or domain name of the server including the port. A GET request to this address will return a Status 405 Method Not Allowed. The POST request contains two form fields: ”uname” and ”password” which hold the submitted username and password respectively. The server then queries the user database for the data for the user with the submitted username. The server then uses verifyPassword() to check the submitted password against the password hash returned from the database. The passwords are hashed using the modern bcrypt algorithm. If the password does not match the hash then the server returns a Status 401 Unauthorized. If the password does match then the server uses createLoginToken() to construct a JSON Web Token (JWT) containing the user’s username, full name, user id, and the expiry time of the current session. The JWT is a structure that encodes a JSON object along with a cryptographic signature so that the client can send this identity data back to the server, and the server can confirm that the data has not been modified by the client. The client then stores the JWT and can use it to access the application.

To finish authentication the client and server then perform a few more steps. The client loads the application page and opens a WebSocket connection with the server. The client then sends the server it’s JWT and the server verifies its authenticity. If the JWT is authentic then the server sets that WebSocket connection’s ”auth” flag to true. This means that the server will now broadcast the data to this client, and in future implementations this client can send commands to the server.

C. User Client

The user client is the user facing part of the system which allows the user to see the loop data and, in future implementations, send control messages to the system. The user client is an electron application built using HTML, JavaScript, and CSS. The two main components of the user client are the Login and Data components. The Login component is built with ”index.html” and powered by ”js/login.js”. The Data component is built with ”stats.html” and powered by ”js/data.js”. ”index.js” starts the application window, and handles file downloading and cookie storage.

The Login component is made up of a simple form with a username and a password input. The user enters a username and password and the server then makes an AJAX POST request to the server. If the server authorizes the application,

then the client stores the JWT and redirects to the stats page (the Data component). If the client is not authorized, then the client displays an error message and the user can try to log in again.

The Data component is the core of the client. It has two graphs and a series of control buttons along with a table that displays all of the current sensor data. The graphs display the data that the client is receiving, and the control buttons change what data the graphs show. When the Data component first loads it uses ”js/ws_connection” to establish a WebSocket connection with the server. It then sends it’s stored JWT to authorize itself. Then whenever a new message is received ”js/ws_connection” reads the ”measurement” pair and updates the application with the new sensor data. The component uses an object in ”js/chartConfig.js” called ”histDataNew” to store all of the sensor data. This object has two parameters: ”vals” and ”time”. The ”vals” parameter holds an array of arrays, where each subarray stores all of the N most recent readings for a specific sensor where N is the variable ”keepTime” in ”js/chartConfig.js”. The ”time” parameter holds a series of time values for each entry in all the ”vals” subarrays. This parameter makes graphing the sensor data easier. The current time is stored in histTime in ”js/chartConfig.js”. The tables on the Data component are updated with the most recent sensor reading by ”js/ws_connection” every time a new message is received.

D. Multi-Component Features

1) Heartbeat: The heartbeat is a combined effort between the main server and the user client. The heartbeat allows the user client to inform the user when either the user client has lost connection to the main server or when the main server has lost connection to the LabView program. Both the user client and the main server run a heartbeat loop which work in similar ways. On the main server the loop is continuously running and a heartbeat variable is used to determine whether or not data has been received. When the loop starts the heartbeat variable is set to 0. If the server receives a message from the LabView program it sets the heartbeat variable to 1. The next time the heartbeat loop runs if it sees that the heartbeat variable is 1, then it sets it back to 0. However, if the heartbeat variable is still 0 then the server knows it hasn’t received a message since the loop last ran. In this case the server sends a ”{”error”:”src_disconnect”}” JSON object to the client and sets the heartbeat variable to 2. The clients heartbeat function works the same way except with messages from the server. The client then displays a message for the user if either it’s heartbeat doesn’t update, or if it receives the error message from the server.

2) Public Accessibility: One of the end goals of this project is to have the system be publicly accessible so that others interested in the project, besides the researchers, can access the system and monitor the loop. This requires at the minimum running the server component on a machine which is publicly accessible (i.e. port forwarded on a network so that it is accessible from outside that network). Unfortunately since the loop is running at the University of Guelph the only network option for the computer running the LabView program is the secured university network. As such, the server software must be running on another machine outside of the universities network. This can be accomplished by running the software on a VPS or with a cloud provider like Google Cloud or AWS, or by running the software on a machine attached to a network that the developer controls.

V. Running the System

A. LabView Program

First open LabView 2017 on the data acquisition computer (the one next to the pump loop). Then open ”Pumps_Data.lvproj”. This is the project file for LabView. Then open ”HTTP_Record_and_stream.vi”. This is the virtual instrument (VI) used for the system. To start the LabView program press the arrow button on the Front Panel, just below the title-bar. To edit this VI select ”View” and then ”Show block diagram”. This will open the block diagram for the program which can be edited when the program is not running.

B. Main Server

To run the server to Go run-time must first be installed. This can be done on a Linux machine by installing the ”go” package, or on a Windows machine by installing a binary from www.golang.org/dl/. Then to use the code from the Github repository the ”key-dummy.go” file must be renamed ”key.go”, and the variables should be filled in with your secret server key; JWT secret; and database username, password, and table name.

If the server is running on Linux the Makefile can be used from here. From the server’s root directory run the command ”make setup” to download all the necessary dependency code for the server application. Then run the command ”make” to run the server. On windows run all the commands in the Makefile under ”setup” to install the dependency code. Then run the command in the Makefile under ”test” to run the server.

C. User Client

To run the user client for development, the Node Package Manager (npm) is required. On a Linux machine it can be installed by installing the npm package. On Windows it can be installed using the binary downloaded from www.nodejs.org (This binary installs both npm and Node.JS). Then npm must be used to install some dependencies. On Windows and Linux run ”npm install electron”, ”npm install electron-dl”, and ”npm install jquery” in the Desktop Application v2 folder to install the dependencies.

D. Running in Production

1) Main Server: Before building the server uncomment the ”gin.SetMode(gin.ReleaseMode)” line in router.go. This will make Gin (the web framework powering the server) run in release mode rather than debug mode. To build the main server on a Linux machine run ”make build” in the Server folder to create a binary. To build the server on a Windows machine run the command under the ”build” section of the Makefile in the Server directory to create a binary.

2) User Client: The user client can be built for any OS from any OS (Although building a Mac application works better on a Mac). To build a Windows application run ”npm build” in the Desktop Application v2 folder. To build a Mac application run ”npm build-mac” in the Desktop Application v2 folder. To build a Linux version of the application ”package.json” needs to be modified. A ”Linux” entry needs to be made after the ”win” and ”mac” entries. Then either the ”build” script needs to be modified to build a Linux app, or a new script needs to be added to the ”scripts” object for building a Linux app. Then run ”npm SCRIPTNAME” with the appropriate script name. All of these build methods will create a binary in the ”dist” folder.

VI. Future Work

There are a number of things that can be improved upon for this system. These improvements can generally be split into upgrades and new features.

A. Validation

While the LabView program that was initially written was tested to ensure that it’s timing and performance was accurate, the newer version of the program (the VI titled ”HTTP_Record_and_stream.vi”) has not been thoroughly tested. This VI still needs to be tested to confirm that when the ”Record” button is pressed the right number of samples is taken, and that the time between each sample is identical. The LabView program takes sensor readings in batches whose size is defined by the ”Samples to read” property of the DAQ Assistant block. In older iterations of the LabView program there would be an additional delay between the last reading of the previous block and the first reading of the next block. For example the readings within a block would be 3.33 ms apart, but there would be 5 ms between the last reading of one block and the first reading of the next. The current LabView program should be tested to ensure that timing issues like this are not present in the recorded data.

B. Upgrades

There are a number of things that can be upgraded in the system. A number of them are related to security. First all the connections should be changed from http:// and ws:// to https:// and wss:// to encrypt the traffic. At the moment all components are using unsecured connections meaning that passwords, keys, and JWTs can all be overheard on a network. In addition making LabView authenticate more like the user client may be beneficial. Having a single transaction where the LabView sends an identification key to the server and the server returns a JWT which the client can send with messages instead of a key would make authentication more uniform across the system. Also changing the LabView program so that it connects to the server with WebSockets instead of POST requests would mean that the server only needs to authenticate itself at the start of the connection. However, this may be difficult as I struggled to get LabView to send a message over WebSockets after making a connection. This is made even more difficult since the WebSocket implementation is user developed and not very thoroughly supported.

The LabView program could also be made more scalable by upgrading the ”Array to JSON” block. Currently the block is hard coded to accept a certain number of sensor values. Upgrading this block to handle sensor data dynamically would make adding or removing sensors simpler. Also, this block should be upgraded so that it outputs the data in a JSON array rather than the arbitrarily defined format it uses now. This would eliminate the need for formatData()and buildArray() functions in the server. Then the server could simply pass the message from the LabView program straight on to the user clients with maybe only validity checks in between.

C. New Features

There are two main areas of expansion for the system in terms of new features. The biggest is adding control of the system to the app. To implement the control the system needs: a way for LabView to send signals to the physical system, and a way for the user client to send commands to LabView. To tackle the first challenge National Instruments (NI, the developer of LabView) make Digital to Analog Converter (DAC) modules which can be controlled from LabView. These can be configured to interact with the pump loop hardware. Next there are two clear avenues for sending data to LabView from the user client depending on how LabView is communicating its data with the Main Server. The biggest challenge here is that if the server is being run on a publicly accessible machine so that the user client can connect to the system from any network in the world, then that means that the server is not running on the university network and thus the server cannot reach out to the LabView machine. If the system has been modified so that the LabView program transmits it’s data though WebSockets then the solution is clear. The WebSocket connection is full duplex and thus, once the connection has been opened, the server can easily send messages back to the LabView program despite not being able to send it requests. However, if LabView can only send data to the server with POST requests there is still a method to send commands to the LabView program. A message queue can be created in the server which stores every command sent from a user client. Then the next time the LabView program sends a POST request with data, the server can dispatch the queued messages in the response. This will allow the server to regularly send commands to the LabView program even though it cannot send requests to the LabView program.

The other main area of expansion for the system is an overhaul of the interface. While the current interface conveys all the transmitted information, it can be modified to more clearly deliver useful information and indicators to the user. Adding elements like gauges to display values, potentially with colors to indicate extreme readings could expand the usability of the interface. In addition useful measurements which are not directly measured, like efficiency, should be added based on the needs of the researchers. The general goal of improving the interface is to make it closer to industry standard interfaces and generally make it more user friendly and easier to understand at a glance.