A Modular Backend Computing System for Continuous Civil Structural Health Monitoring

Ting-Chou Chien[1], Chengjia Huo[1] and Pai H. Chou[1,2]

[1]University of California, Irvine, CA USA
[2]National Tsing Hua University, Hsinchu, Taiwan

ABSTRACT

This paper describes a computing backend for a waterpipe monitoring system. Today, most such systems are divided into event-triggered and continuous monitoring, but they all lack systematic handling of data. Many systems simply store data in files with specific naming conventions and ad hoc formats, making them difficult to retrieve, maintain, disseminate, and analyze.

To address these problems, our backend supports data management and dissemination. Unlike previous systems that store data in files or conventional databases before analysis, our modular architecture not only saves data in efficiently searchable ways by indexing as a baseline dataset but also detected events in discrete time manner and other processed data. To facilitate analysis, we design a plug-in structure to allow processing modules to perform inline processing and shorten detection time. For data dissemination, our architecture can compose multiple visualizations including geographical maps to create powerful tools to yield new insight into massive datasets. The backend system enables Internet web service for visualization, data mangement, and remote sensor control for better integration. Our system is applicable to not only water pipelines but also bridges and civil structures in general.

Our proposed backend system has been implemented and validated through field deployment. One such system has been running for over 1.5 years and has collected millions of records to date. A Google Map intergrated visualization service has been developed to demostrate lively collected records in real-time. This is expected to be more helpful for better understanding of civil structures' behavior in the long term.

Keywords: Water pipe monitoring, MEMS sensors, civil structural health monitoring, wireless sensor network, backend system, modular design

1. INTRODUCTION

Modern societies depend on the reliable operation of a wide variety of civil structures and infrastructure systems. Their failure to perform properly can adversely affect the lives of millions. To ensure the performance of these systems, smart sensor systems are installed to monitor their state of health. They are deployed in buildings and on bridges for structural health monitoring, so that unsafe structures can be repaired or shut down before they collapse. Aging infrastructure systems such as freshwater and waste water pipeline networks are also being monitored using smart sensors. In case of rupture or leakage, the smart sensors can help localize and contain the damage.

Deploying the smart sensors are only half of the story; more challenges are posed by the design development, deployment of the monitoring system, and the actual monitoring process. Access to smart sensors after deployment is often necessary for maintenance such as firmware upgrade, bug fixes, and even battery replacement, but it can be expensive and cumbersome to visit deployment sites, which may require administrative approval. For example, sensors for civil structural health monitoring are installed in hidden locations to prevent possible unintended access by non-technical people. These water pipelines are often installed underground, and their accessibility may be limited to manholes covered by metal lids. Due to potentially explosive gasses, technicians need special training to go down the manhole. In short, post-deployment access to smart sensors is expensive, and their remote administration should be enabled as much as possible.

To make the entire system operate as a coherent system, a *backend computing system* is crucial. A backend system serves several purposes, including data storage, processing, dissemination, supervisory control, and system administration.

Further author information: (Send correspondence to Pai H. Chou)
E-mail: phchou@uci.edu, Telephone: +1 949 824 3229

First, the smart sensors upload either raw data or processed data upstream to the backend system for data archive. In some cases, data may need to be disseminated to related agencies and even the public at large. Second, the archived data may be post-processed for analysis of long-term trends as well as short-term event detection. Third, the backend system should be able to provide an interface for the operator to have a global, supervisory view of the state of the entire system and to control it, especially in response to emergency situations. The response may be triggered automatically or specified manually. Fourth, the smart sensors themselves may require support for administrative tasks, including firmware update, system diagnostics, and periodic tests. The backend system can be crucial to the integration of all of these tasks.

The backend system may be implemented using a wide variety of technologies, ranging from simple web scripts to databases and cloud computing technologies. What makes it challenging is the overall structure that is scalable with the size of the network. To achieve best allocation of computing resources and communication bandwidth, the system designer may need to decide how to partition the functionality among the subsystems. A systematic representation of collected data is the key component for backend system, and the right visual representation can be very effective at conveying complex information instantly. However, different classes of monitoring systems require different solutions. Many existing monitoring systems fall under the category of *event-triggered* monitoring, where the sensor signal value is a direct indication of events without requiring processing, and little or no action is required in the absence of events. In this case, a file-based logging mechanism is sufficient, and the monitoring backend system can follow the structure of a conventional server. In contrast, a *continuous monitoring* system processes much more data samples, and both the raw and processed data sets may need to be archived and made available for retrieval, which usually cannot be handled by a file-based system. Instead, it requires a modular structure where different processing algorithms may be plugged in, and it must also be able to integrate external data sources (e.g., overlaid data on Google Earth) in the form of *mashup* to enrich the data presentation to the user without having to re-invent these features. The resulting system must also be robust and highly available with minimal or no down time.

To meet these requirements, we describe our work on a backend computing system. It provides a modular hardware/software organization, robust hardware and software, device management, data collection, and dissemination.

2. BACKGROUND

2.1 Event Triggered vs. Continous Monitoring

Sensor systems for structural health monitoring (SHM) can be categorized into *event-triggered* and *continuous monitoring* systems. An event-triggered one wakes up sensors to collect data in the presence of some pre-defined events, such as earthquakes and heavy winds. In contrast, a continuous monitoring system collects all data samples. In general, SHM systems require sampling rates on the order of 1000 Hz and can be considered continuous monitoring systems. However, Depending on the response time and battery-life requirement, the system may perform *undersampling*, i.e., by sampling at the full 1000 Hz rate only for a few minutes per day and sampling at 50 Hz or lower at other times. One such system is Pakzad,[1] which has deployed a total of 64 nodes on the Golden Gate Bridge in San Francisco, California to test the scalability and performance of the wireless sensor network. Each node is based on the commercially available MicaZ platform with two accelerometers and one temperature sensor. Ho[2] deployed a solar-powered vibration and impedance sensor based on the iMote2 on the cable-stayed Hwamyeong Bridge in Busan, South Korea and evaluated hybrid SHM capability, solar-powered operation, and wireless communication.

2.2 PipeTECT System for SHM and Water Pipe Monitoring

In our previous papers,[3–5] we have developed a continuous monitoring system called PipeTECT to monitor disaster events on water pipelines. The smart sensors, called DuraMotes, perform noninvasive monitoring on the exterior of the water pipes by measuring its vibration at 1000 Hz. The same sensing mechanism can be applied to civil structure health monitoring as well, such as buildings and bridges. The PipeTECT system has a backend computing system for controlling sensing system, storing acquired data, data post-processing, and data dissemination. We have design the backend system to provide a web service for users. This makes the system accessible on regular personal computers and also modern mobile devices with Internet connectivity. As with many experimental systems, the massive amount of data generated by our sensing systems on pipelines and other civil structures were archived on our backend server as files. To discover a sudden disaster event in time domain, continuous monitoring was necessary, since events could not be readily detected by the sensors themselves without a global analysis of data from different sensors in the network. Due to the limited number of accessible locations (e.g., manholes or fire hydrants) for noninvasive water pipe monitoring, the sensors cannot be densely deployed but are often kilometers apart, one of the fundamental tasks is *rupture localization*, which is to pinpoint where rupture occurs. Unlike triangulation, which can be done as collaborative event detection, the backend server collects vibration data from sensors deployed over the entire city and computes the pressure change gradient on a hydraulic model. Subsequently, the system must trigger a notification to administrator for how to respond to the event if rules for automatic response have not been defined.

One issue with the backend is how data are stored and handled. The web server is the universal way for the dissemination of all types of data, including sensor data, and the natural way is to organize the data as files whose names encode the date, time, location, and other metadata. The data may even be in human-readable text for manual inspection. Also, if plain text files are used instead of self-descriptive formats such as XML, then the processing algorithm may need to load in different parsers specific to each file's data format or risk incorrect data interpretation. However, it can be inefficient to search the very large volume of data, and the text representation can also be space requirement. Although text files are easy to compress, it will only exacerbate the search problem.

3. SYSTEM

Our PipeTECT monitoring system can be divided into *sensing* and *backend* subsystems. The nodes in our sensing subsystem form a tiered network, where the sensing tier is wired for underground operation, and the aggregation tier is wireless for above-ground operation. Our backend system is further divided into three subsystems base on functionality: *mudular kernel*, *data collection and storage*, and *data dissemination*. We briefly review our sensing system first and then describe our backend system.

3.1 DuraMote Smart Sensor

DuraMote is the name of our tiered networked sensing system. The two tiers are the sensing tier and the aggregation tier. Because the system was originally designed for water pipeline monitoring, and most water pipes are underground, the sensing tier must be able to transmit data and power in underground settings. Therefore, a wired network solution is used for the sensing tier. Multiple sensors in a sensing tier can be connected to a data aggregator node at the sensing tier, whose nodes can collaborate to route data to or from an Internet uplink. Because the data aggregators are far apart from each other, a wireless network is more practical in terms of ease and cost of deployment. To support this tiered network operation, we design two types of nodes: a sensing node named *Gopher* and an aggregator node named *Roocas*.

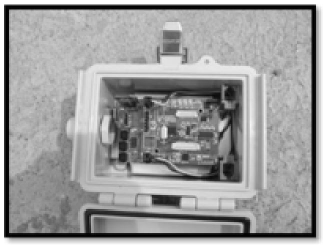


Figure 1: Gopher

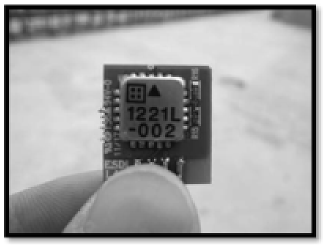


Figure 2: Accelerometer

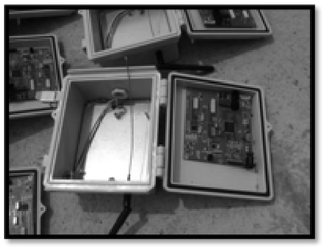


Figure 3: Roocas

A Gopher node as shown Figure 1 is the wired smart sensor to be deployed directly on the exterior of pipelines or on civil structures for non-invasive monitoring. It contains up to three uniaxial MEMS accelerometers (SD1221L-002) as shown in Figure 2, where the Z-axis accelerometer is soldered on-board while the X and Y axes ones are of removable type. The main board also includes a tilt sensor and expansion sockets for connecting other sensors, such as humidity, pressure , and gas. The sensor signals are digitized by the QuickFilter chip (QF4A512), which contains a 4-channel, 16-bit analog-to-digital converter with digital signal conditioning functions. It can be configured as a band-pass filter for different applications and generate different sampling rate accordingly. The Gopher nodes support Controller Area Network (CAN) as its primary data communication interface in daisy-chain topology in the sensing tier. One end of the daisy chain is a data aggregator for Internet uplink. The sensor nodes transmit sensing data via CAN bus upstream through the data aggregator, which has the option of logging data in its own memory card and processing before forwarding the data to the backend. Data can also travel downstream for commands and system administration.

A Roocas node as shown in Figure 3 is a data aggregator node without sensors. It contains communication interfaces for downstream, upstream, and in-tier communication. The primary downstream interface to Gophers is CAN, which is used for both data communication and power. The primary in-tier and upstream interface is Wi-Fi (802.11n, up to 250 m) by default, although Ethernet is also an option. In addition, the Roocas board contains expansion interfaces for other wireless modules, including XBee (up to 1 km range), XStream (by Digi, up to 64 km range), and Bluetooth Low Energy (BLE). The XBee and XStream can serve as the protocol for the aggregation tier, whereas BLE is more suitable for downstream, but there are no inherent limitations. Another common use of the Roocas is data logging onto a Secure Digital (SD) flash memory card when an uplink to the backend is unavailable.

We use the RJ-9 modular connector commonly used for telephones as the wires for our CAN bus. As RJ-9 cables contain four conductors in a bundle, we use two of the wires for data and two for power distribution. Each Roocas node can connect up to 4 Gopher nodes in daisy-chain topology due to bandwidth limits of 1 KHz sampling rate, even though CAN itself can support a much larger network. The length of cable is up to 25 m between Roocas and Gopher, mainly due to cable resistance. If a longer length is desired, then cables of better quality or an extra cable must be used to reduce the voltage drop. This design enables a Gopher node to reach an underground pipeline covered by a metal lid or sealed confined spaces while the Roocas is deployed aboveground with a steady power supply and Internet uplink.

3.2 Backend System

We started out our backend system design with following goals in mind: cross-platform, universal accessibility, dynamic software module loading, fast data storage service, visual representation, and service mashup capability. Our backend system can be viewed as a next-generation supervisory control and data acquisition (SCADA) system, but instead of being implemented in native code only for one operating system, our backend system is executable on Windows, Linux, and Mac operating systems. We achieve the cross-platform interoperability by implementing most backend features in the Python programming language, which runs on most operating systems. On the front end, we build our user interface as a web application to be access from multiple devices with minimum effort. We now describe the subsystems in the backend, including the kernel, data collection and storage, and data dissemination.

3.2.1 Kernel

The kernel subsystem is responsible for coordinating the flow of data in our backend system by dispatching the associated tasks in each stage, as illustrated in Figure 4. The basic tasks are networking, data collection and storage, and data dissemination, while additional tasks can also be loaded by the dispatcher as needed. The kernel dispatches the networking task to receive data packets from the sensing subsystem. Then, the kernel dispatches the data collection and storage task to process the received data and store the results. Once the data or results are available, the data dissemination task renders them in different forms as requested.

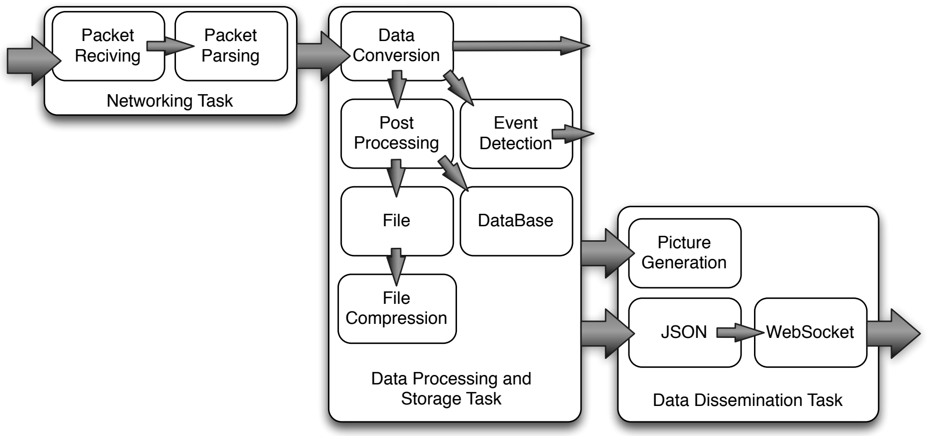


Figure 4: Overview of Our Backend System

Each task is a standalone process and communicates with another task through message queues. Unlike a process (in operating systems) that needs to be spawned and is destroyed upon completion, tasks (in our kernel) always exist in the system and are awaiting messages. In a task, there are one or more software components. These components are loaded when a new message received by the task. The execution sequence of these software components is either sequential or parallel base on the sequence of execution. Sequential software components can be grouped as a thread.

When a data is placed on the message queue, the dispatcher will check if the recipient task exists in the system and loads those that will be needed. Upon receiving a message, the task process loads all registered software components on the fly. This dynamic loading into the plugin system of the kernel is accomplished by Python's metaclass.[6] With the plugin system, new tasks and software components will be loaded dynamically without requiring system reboot. The plugin system also checks if updated tasks exist by checking the object hash of the task and components. While software components are loaded when needed, It takes more time to fully replace a task on the fly and may not be fast enough for a continuous monitoring system. Thus, the updated tasks would be invoked first and start handling incoming data, and the dispatcher would first disconnect the message queue to the old task. The old task would continue to exist after current data is handled.

3.2.2 Data Collection and Storage

The data collection and storage subsystem is divided into two tasks: network communication and data processing. The former interfaces with the sensor network, while the latter handles data storage as well as applying algorithms on the data.

The kernel dispatches the networking task when a data packet is received on the backend system's network interface from the PipeTECT sensing subsystem. The data packet is first cached in the message queue before a computation unit becomes available in the networking task. Then, the networking task parses the packet into multiple data records according to the pre-defined format and puts them in the outgoing message queue to the next task, so that they are ready for further processing. There may be multiple network interfaces with different message formats and communication protocols, but usually only one networking task exists in the backend system.

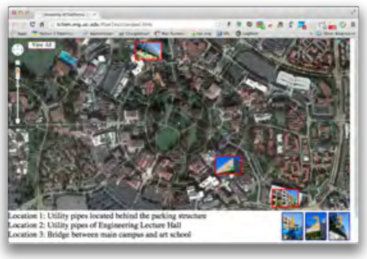
The data-processing task inputs a set of received data records and applies processing algorithms to extract higher-level knowledge about the structure being monitored such as its structural health, remaining service lifetime, residual payload capacity, or the rupture location. The task can also store the raw or processed data on the backend computer's disk space or send them to the dissemination task, which can render the data in different forms to users on other computing systems. The data processing task first converts the data records into the appropriate unit before sending data records to other tasks or other software components in the task. The converted data may be sent directly to the data dissemination task for visual rendering, to the file logging component, or to the event-detection component to see if a warning message need to be issued to administrators. Data processing in a task can be executed in parallel depending on the knowledge distilled from the collected data.

3.2.3 Data Dissemination

The data dissemination subsystem is responsible for not only making raw and computed data available for download but also for handling queries into the data sets and rendering them for visual presentation. For a continuous monitoring system, the raw data should be archived and made available to researchers and future investigators, and in the most extreme cases, the data should be made available in entirety. However, most of the time, the user may be interested in only selected segments that satisfy the filtering criteria; in fact, even with filtering, the number of data records may still be too large to show in the raw form. However, limiting the data to a small subset may be insufficient for detecting long-term trends. Today, most such visualization systems are implemented today as a standalone window-system application program. In contrast, we start our dissemination system as a web service, which is readily accessible by all systems with a browser.

The first generation of our graphical user interface is based on HTML and static image files generation. Generated files can be put on a existing web server and be ready for viewing by users. Figure 5 and 6 show example screenshots of static PNG images generated by the data dissemination process using the matplotlib Python library[7] to visualize the sensing data collected by Gopher nodes. The generation time of an image depends on the size of the collected data and resolution of the result image. However, a web-based image presentation may not be suitable for mobile devices due to the limited screen size. For a smoother user experience, the visual presentation needs to be manually adjusted based on the web page's refresh rate, the amount of sensing data, and the resolution of the generated image, but this manual tuning process is time-consuming. An acceptable configuration is to have a 5-second refresh time interval, 30 seconds of sensing data, and an 800×600 image resolution for a 1024×768 web page. User could also view the content on a mobile device such as smart phones or tablet, but the configuration is not optimized for this kind of usage in Figure 7. In any case, the first generation suffers from a rough user experience, because images fetched from a browser cannot be perfectly synchronized with frequent image generation.

The second generation of the system improves the user experience by means of dynamically generated web contents instead of static files. That is, the backend system serves as a content provider rather than a static file generator, and much of the data rendering is done locally by the user's browser. We use the Python Tornado Web Server and enable picture generation by the user's browser using the Javascript library named flot,[8] instead of static image generation. Because each browser knows its own size, it can render the image size and layout accordingly for the best viewing experience. The webpage would refresh every 5 seconds, request new sensing data from the backend system, then generate the new image locally on user's personal computer, laptop, or mobile phone. The browser needs to fetch enough sensing data in each request of sensing data for image generation. When Internet is slow or unstable, the display will be delayed, but local manipulation will still be responsive. Although our second-generation system has solved the problem with static file generation of the first generation, it still suffers from high data traffic in every webpage refresh.


Location 1: Utility pipes located behind the parking structure
Location 2: Utility pipes of Engineering Lecture Hall
Location 3: Bridge between main campus and art school
Figure 5: Campus

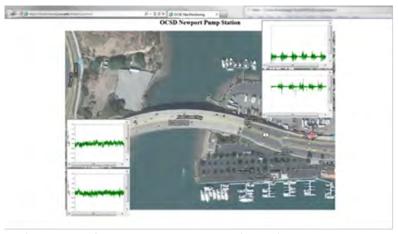


Figure 6: OCSD monitoring system

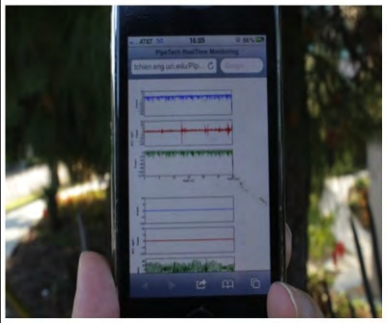


Figure 7: Using Smartphone to show the monitoring system

Our third-generation data dissemination subsystem tries to reduce the redundant data traffic pulled by the browser in each refresh. The local browser can keep a data window that caches the received data and update only the most recently sensed data from the server. Upon each data request, the browser pulls only the newly added data since the last data request. The amount of transferred data is reduced by about two orders of magnitude, from thousands of bytes to several tens of bytes in average. In addition to pulling, we can also push newly added data from backend system to connected browsers using a new technology, websocket, introduced along with HTML5. We further improve the user experience by converting fixed-rate data pulling by the browser to data pushing by the backend server. Pushing enables the backend to control the web page content based on the incoming data rate and is more bandwidth-efficient. This also eliminates the problem of sudden massive simultaneous requests to the backend server, which can overwhelm the server.

In addition to the data traffic improvement, we also create mashup services by integrating existing web servics such as Google Map.[9] We can use the latest and scalable map information instead of fixed image. A sample monitoring system is shown as Figure 8.

After the above improvements in the data dissemination subsystem of our backend system, the system was upgraded from a file provider to a data provider and then to a content source provider through the Tornado web server. Now the content provided by our backend system, i.e., the acceleration data from Gophers, can be subscribe to by not only our own dynamic web pages but also other HTML5 clients in general. All these web message exchange are in JavaScript Object Notation (JSON) as a widely used format.

4. EVALUATION

We have performed multiple short-term and long-term experiments with our PipeTECT system. Short-term experiments are mostly to validate the capability and stability of the DuraMote smart sensor system. Our test sites include (1) Vincent Thomas Bridge in Long Beach, California, (2) Hwamyeong Bridge in South Korea, (3) a miniature pipeline model, (4) buildings at University of California, Irvine, and (5) fire hydrant at a fresh water facility. Experimental results on the collected data have been presented in our previous papers. The experience from the short-term experiments provided

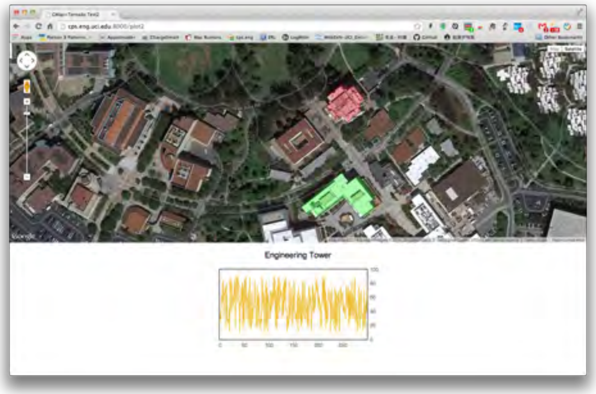


Figure 8: Google Map Service integration

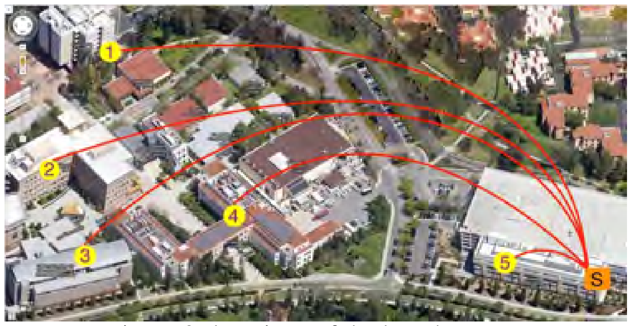


Figure 9: locations of deployed sensors

feedback for us to further developing our backend system. We also installed our PipeTECT system in several buildings at the University of California, Irvine for our long-term experiment.

4.1 Lessons from Field Experiments

From the field experiments, we have learned lessons about the scalability, system stability, and network performance. The weakest links and bottlenecks have been identified in all different subsystems.

Initially, our system was not built with scalability in mind. In our short-term experiments, we deployed only several DuraMote sensors and backend system worked well, but our experiments on the Hwamyeong Bridge in Busan, South Korea, revealed our performance bottlenecks, where communication speed, bandwidth, and computation delay can all limit the number of sensing units that a backend system can serve. Each Gopher node sampled at 450Hz and generated 3.6 KB of data per second. The 802.11n Wi-Fi infrastructure provided sufficient network speed and bandwidth for the 14 Gopher nodes that generated a total of 50.4 KB of data needed to be handled per second. However, the computation delay turned out to be a performance bottleneck. The system was failing due to insufficient processing time for the networking task in our original design. It combined networking and data processing, which took too much time to process the data and could not keep up with the incoming packets. As a result, the backend system exhausted all available memory and crashed.

In Hwamyeong Bridge deployment, we made the in-field backend system to provide collected data through web service for our centralized backend located in University of California, Irvine to access. Although the data could be delivered to the centralized backend, the amount of data traffic was more than the cross-Pacific network bandwidth, this led to poor visual presentation performance and unstable network connections. We have since overcome this bottleneck by converting to websockets.

In these short-term experiment, we stored all data as files. The file-based storage lacked an efficient query interface for later data dissemination, and we have since converted it to MySQL and SQLite database storage as our data logging engine. The SQLite database is a relative simple database compared to MySQL. Since SQLite database can be found on mobile devices, our mobile version of backend system uses SQLite for logging and local browsing at hand. MySQL is a more powerful database that provides a fast query interface. When searching for a certain time epoch of data in all stored data, it outperforms the file-based storage.

4.2 Data Traffic Optimization

This subsection shows quantitative results of data traffic improvement by upgrading our data dissemination task from static files to dynamic web content to websockets, as described in Section 3.2.3. As shown in Table 1, the data traffic improvement was three-fold while the image generation time reduced slightly. At the same time, the modular plug-in system implementation enabled seamlessly software upgrade. The transmitted data size has also reduced.

For the baseline, our graphical user interface generated static files using the matplotlib Python library. The backend system creates new image files as new sensing data arrive, but to avoid a surge of the image files that all need to be created within a short period, image generation process of each Gopher is separate by time. As a result, the average image generation time is 0.69 second, but the delay can easily add up to several seconds if the backend system needs to take care of more PipeTECT sensing systems. The generated image size is nontrivial and takes about 64 KB in 800×600 resolution. User can feel the delay when the web page is refreshing. A Javascript-based solution has been developed to solve these problems, as the image generation process is now run in the browsers for more responsiveness.

Table 1: Improvemnts in Data Dissemination Subsystem

System types	Image gen. method	Image gen. time(sec)	Transmitted data(bytes)
Static files	Matplotlib(Python)	0.69	64K
Dynamic web content	Flot(Javascript)	0.56	1.5K
Incremental content update	Flot(Javascript)	0.56	>20

The size of transmitted data from backend to browse is almost 43 times less then before. To view the result of collected data is much smoother not but still not ready for a larger monitoring system. When more then 17 PipeTECT systems exists, the communication delay happens due to too much data need to be sent from backend system. To further improve our system, an incremental update concept has brought in. The transmitted data is cached locally and the backend system only sent newly added data to browser. Only then the browser requests for a full set of data for display then big chunk of data is transmitted. The transmitted data is now no more than 20 bytes for a single PipeTECT system shown on the web page. We also make the incremental update only occur with viewed PipeTECT system. Third is the load on backend has been alleviated due to the image generation now moves to browsers. Backend system only is responsible of data transmission and the browser do the heavy image generation. This enables the backend to focus on data processing and leave all other display tasks to the actual viewing devices.

4.3 Long-Term Installation at UC Irvine

We have installed DuraMote smart sensors in several buildings and on utility pipes at the University of California, Irvine to evaluate our proposed backend system. Wi-Fi infrastructure is available for direct Internet connectivity. The proposed backend system is implemented on an Intel Pentium-4 3.0 GHz desktop PC with 4 GB of memory and installed with Ubuntu Linux operating system version 11.10. This nonserver-class PC is utilized to show our backend system does not required tremendous computing power.

Figure 10: Engeneering Tower

Figure 11: Engineering Hall

Figure 12: CALIT2

Figure 13: Engineering gateway

Figure 14: AIRB

The monitored area covered by this deployment is approximately 200 m^2. Five nodes were installed on five different buildings inside the engineering quad of UCI campus. Figure 9 shows all the monitored buildings in covered area. The locations of the nodes are basement of Engineering Tower, fourth floor of Engineering Hall, fourth floor of Engineering

Gateway, roof of CalIT2 building, and fourth floor of AIRB building, as shown in Figures 10 to 14. The backend server is placed on the ground floor of the AIRB building, where the laboratory is located.

All DuraMote sensors are connected to the campus Wi-Fi and send collected data directly to the server in the AIRB building. Most Gopher nodes are configured for a sampling rate of 150 Hz, while the one in the basement of Engineering Tower is configured for 1000 Hz to monitor a utility pipe. The deployed nodes have been monitoring since 2012 for more then a year and a half. It successfully demonstrates that our backend system is ready for permanent installation.

During the installation, several software upgrades were necessary to improve the performance and functionality of our backend system. Due to the modular design, we were able to avoid rebooting, which would have affected the network connection of these deployed sensor and disrupt the monitoring process.

5. CONCLUSION AND FUTURE WORK

In this paper, we describe our modular design of a backend computing system for continuous monitoring of civil structures and water pipelines. Its use of the Python programming language enables different software components to be developed in a modular way, and the kernel system loads the required modules accordingly using the internal plugin system. The downtime of the backend system is minimized since the software module loading is seamless without affecting the rest of the system. Based on our experience with deploying our smart sensors, we improved our backend system by enhancing the data logging facilities with fast data query. and optimizing the data display for reduced network traffic by two orders of magnitude, making the backend system much more scalable. We also create mashup services by integrating Google Map overlaid on our data visualizer. One direction for future work is to integrate even more services such as earthquake and weather information to enable cross references of collected data for discovering more knowledge about these structures being monitored.

ACKNOWLEDGMENTS

This study was sponsored in part by a National Institute of Standards and Technology (NIST) Technology Innovation Program (TIP) Grant 080058, an NIH STTR contract 11R41HL112435-01A1, and SBIR contract 1R4HD074379-01.

REFERENCES

[1] Pakzad, S. N., Fenves, G. L., Kim, S., and Culler, D. E., "Design and implementation of scalable wireless sensor network for structural monitoring," *Journal of Infrastructure Systems* **14**(1), 89–101 (2008).

[2] Ho, D.-D., Lee, P.-Y., Nguyen, K.-D., Hong, D.-S., Lee, S.-Y., Kim, J.-T., Shin, S.-W., Yun, C.-B., and Shinozuka, M., "Solar-powered multi-scale sensor node on Imote2 platform for hybrid SHM in cable-stayed bridge," *Smart Structures and Systems* **9**(2), 145–164 (2012).

[3] Shinozuka, M., Lee, S., Kim, S., and Chou, P. H., "Lessons from two field tests on pipeline damage detection using acceleration measurement," in [*SPIE Smart Structures and Materials+ Nondestructive Evaluation and Health Monitoring*], 798328–798328, International Society for Optics and Photonics (2011).

[4] Kim, S., Yoon, E., Chien, T.-C., Mustafa, H., Chou, P. H., and Shinozuka, M., "Smart wireless sensor system for lifeline health monitoring under a disaster event," in [*SPIE Smart Structures and Materials+ Nondestructive Evaluation and Health Monitoring*], 79832A–79832A, International Society for Optics and Photonics (2011).

[5] Torbol, M., Kim, S., and Chou, P. H., "Remote structural health monitoring systems for next generation SCADA," *Smart Structures and Systems* **11** (March 2013).

[6] Talin, "Metaclasses in Python 3000."

[7] Hunter, J., Dale, D., Firing, E., Droettboom, M., and the matplotlib development team, "Matplotlib: python plotting."

[8] IOLA and Laursen, O., "Flot: Attractive JavaScript plotting for jQuery."

[9] Google, "Google Maps API – Google Developers."