Monitoring Heritage Buildings with Wireless Sensor Networks: The Torre Aquila Deployment

Matteo Ceriotti[1,2], Luca Mottola[1,2], Gian Pietro Picco[1], Amy L. Murphy[2], Ştefan Gună[1], Michele Corrà[3], Matteo Pozzi[4], Daniele Zonta[4], Paolo Zanon[4]

[1]Dip. di Ingegneria e Scienza dell'Informazione, University of Trento, Italy
[2]Bruno Kessler Foundation—IRST, Trento, Italy [3]TRETEC S.r.l., Trento, Italy
[4]Dip. di Ingegneria Meccanica e Strutturale, University of Trento, Italy

ABSTRACT

Wireless sensor networks are untethered infrastructures that are easy to deploy and have limited visual impact—a key asset in monitoring heritage buildings of artistic interest. This paper describes one such system deployed in Torre Aquila, a medieval tower in Trento (Italy). Our contributions range from the hardware to the graphical front-end. Customized hardware deals efficiently with high-volume vibration data, and specially-designed sensors acquire the building's deformation. Dedicated software services provide: *i)* data collection, to efficiently reconcile the diverse data rates and reliability needs of heterogeneous sensors; *ii)* data dissemination, to spread configuration changes and enable remote tasking; *iii)* time synchronization, with low memory demands. Unlike most deployments, built directly on the operating system, our entire software layer sits atop our TeenyLIME middleware. Based on 4 months of operation, we show that our system is an effective tool for assessing the tower's stability, as it delivers data reliably (with loss ratios <0.01%) and has an estimated lifetime beyond one year.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Routing protocols*; D.1.0 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*

General Terms

Design, Experimentation, Measurement

Keywords

Wireless sensor networks, Heritage buildings, Middleware

(a) External view.

(b) Inside views.

Figure 1: Torre Aquila.

1. INTRODUCTION

Heritage buildings are a fundamental constituent of a country's historical memory. Their preservation is thus a major concern. Planning the maintenance of such structures requires a careful assessment of their structural integrity, along with a precise and quantitative understanding of the factors that may affect them. The latter is traditionally achieved through sensors and data loggers monitoring quantities such as vibrations, temperature, and humidity. However, these devices are typically cumbersome to deploy, as they require a nearby power outlet or extensive wiring. Therefore, their number is limited, and so is the monitoring: this is especially true in buildings containing works of art, due to the visual impact and physical encumbrance of the instrumentation.

In this context, wireless sensor networks (WSNs) enable radically different solutions overcoming the above limitations. Small, self-powered nodes relying on radio communication reduce the invasiveness of the system, allow the deployment of more devices, and enable experimenting with different configurations of the sensing infrastructure.

Torre Aquila. The above requirements were evident in Torre Aquila, where we conducted the study reported in this paper. Located in the city of Trento (Italy) close to the Buonconsiglio Castle, it is a 31 meter-tall medieval tower whose 2nd floor contains "Il ciclo dei mesi" ("The Cycle of the Months"), a series of internationally-renowned frescoes that represent a unique example of non-religious medieval painting in Europe, attracting thousands of visitors every year. The tower and some of the frescoes are shown in Figure 1.

The preservation of the frescoes is the main source of concern for the local conservation board. In ancient times Torre Aquila represented the main entrance to the city from the East: with the expansion of the city in the second half of the 19th century, most of the eastern city wall was demolished and the entrance to the city was moved a few hundred meters south of the original gate. Today this solution is inadequate for the increasing vehicular traffic. The solution to this problem, pursued by the Municipality of Trento, is to bypass the obstacle of the Castle compound with a road tunnel. The construction of the tunnel has been long delayed due to concern by the conservation board that construction work might cause unwanted settlement of the tower foundations. The timely estimation of the potential risk to the frescoes requires real-time monitoring and appropriate response models to reproduce the structural behavior of the tower.

Peculiarity of the WSN deployment. The use of WSN for monitoring the integrity of civil structures is not new, as we discuss in Section 2. Nonetheless, Torre Aquila poses peculiar challenges that are not usually found in the deployments reported in the literature:

- *Heterogeneity.* The system contains many kinds of sensors, whose operation is quite different. Deformation and environmental parameters can be sampled at a low rate, but vibration must be monitored at a high rate, which consequently demands efficient reporting of the resulting high volume of data. Both modalities must gracefully co-exist in the same sensing infrastructure.
- *Temporal span.* The time constants of the phenomena of interest require monitoring to span months or even years. In contrast, the systems found in the literature typically operate for at most a few weeks.
- *Online tasking.* The ability to change the behavior of the sensing infrastructure based on external input can be very useful. For instance, it is interesting to monitor vibrations when a visit by a large group of people is expected, or when strong winds are forecast.

Contribution. In this paper, we present the hardware/software solution we developed to efficiently address the above requirements for monitoring Torre Aquila.

The hardware core is based on TMote-like devices, customized as illustrated in Section 3. Deformation measurements are acquired by fiber optic sensors stretching the length of the tower. These sensors, developed especially for our deployment, required custom integration with the motes used to report the measurements. Moreover, high-rate sampling and reporting of vibration data demanded buffering into a short-term storage. The flash memory usually found in motes is ill-suited for this task, due to its high latency, energy consumption, and limited number of writes. Hence, we integrated on the mote a 32 Kbyte FRAM (Ferromagnetic RAM) chip, overcoming all of these problems. To the best of our knowledge, we are the first to use FRAMs in a WSN deployment.

Unlike the hardware, our software layer is not based on what can be considered a "standard" core. Instead of developing directly on top of the operating system, we chose to empower our developers with the higher level of abstraction provided by a WSN middleware, TeenyLIME [5]. We are unaware of studies reporting the use of a WSN middleware in the context of a real-world, long-running deployment. Moreover, as illustrated in Section 4, our use of middleware is not limited to the application logic: the lower-level services necessary to the system operation (i.e., data collection, data dissemination, and time synchronization) are all implemented directly on top of TeenyLIME.

Deployment details such as the placement of nodes and sensors are reported in Section 5. In the same section, we show and interpret data gathered during 4 months of operation, as an example of the insights gained about the tower status. After looking at our implementation from the end-user's perspective, Section 6 analyzes it from a system one. We evaluate the system performance w.r.t. data delivery and lifetime—often considered key metrics in WSN deployments—showing that our implementation achieves a data delivery close to 100% while working at very low power. Moreover, we discuss the benefits brought to development by the use of our middleware, in terms of reduction of programming effort and code reuse.

Section 7 contains brief concluding remarks along with our plans for future work.

2. RELATED WORK

In [10], the authors note that WSN deployments to date can be divided in two categories: environmental monitoring applications (e.g., [15]), designed with low-power operation allowing them to run for long periods, and high-rate, high-fidelity ones running only for a relatively short time. The deployment in Torre Aquila inherits challenges from both classes, as we must deal with high-rate data and yet the system is required to operate for long periods.

In general, although WSNs have been used for monitoring civil structures [10,3,13,7,22,4], the combination of requirements we must address is unique. For instance, only a handful of the systems surveyed in [13] can be tasked remotely, and in these cases (e.g., [4]) the implementation lacks support for low-power operations, hampering their use in long-running deployments. Similarly, most deployments deal only with monitoring vibrations [13], without the increased complexity due to heterogeneous sensors, as in Torre Aquila.

In some cases, the hardware is designed bottom-up for a given deployment. For instance, the work in [10] uses directional antennas, motivated by the peculiar shape of the target area. We cannot afford the luxury of fixing the network topology, as structural engineers are likely to relocate the nodes over time. Moreover, the nodes used by [10] cost ∼\$600 each, which in our case would make the WSN solution not cost-effective compared to a traditional one. Instead, one of our customized nodes costs ∼\$120.

On the software side, real-world deployments mostly feature ad-hoc implementations [10, 13, 22], which make very difficult extending or adapting their functionality to different scenarios. Moreover, where higher-level approaches have been proposed [7,4] the deployments targeted short-term use. Our middleware-based one sustains good performance over a long time span, and yet fosters component reuse in other scenarios, as we discuss in Section 6.

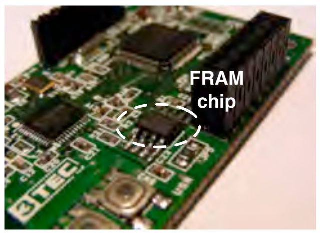
In summary, our goals set us apart from the state-of-the-art. We are neither confirming with a proof-of-concept "the *eventual* ability to cover a large civil structure with low-cost wireless sensors" [3] nor we are validating *already known* models using WSNs instead of conventional systems [10]. Our requirements, set by the structural engineers on our team, are instead to design, implement, and deploy an operational system that, by delivering good performance over a long period, helps them to assess the status of Torre Aquila. The rest of the paper describes how we achieve this goal.

3. HARDWARE

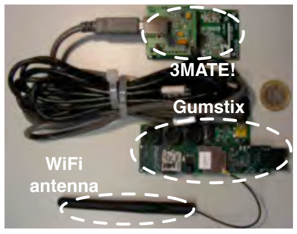
Our requirements demand customized hardware. We selected as the core platform 3MATE! nodes, developed by TRETEC (`www.3tec.it`), an easily extensible WSN node similar to TMotes [18], shown in Figure 2(a). The base 3MATE! is equipped with a TI MSP430 CPU, a ChipCon 2420 radio, and an inverted-F microstrip antenna. Differently from TMotes, the USB interface can be detached if not needed, reducing power consumption once deployed, and the board layout is designed to easily accommodate customized extension boards. Co-location with the manufacturer helped us to accommodate rapidly the needs of our deployment. The nodes have been customized differently according to their sensing goals, as described next.

Environmental nodes. We developed a 3MATE! extension board for environmental monitoring, equipped with simple analog temperature, relative humidity, and light sensors. In the deployment reported in Section 5, however, temperature was the only measure required by the end user. Sensitivity to temperature ranges from $-40°C$ to $125°C$ with a typical accuracy of $0.5°C$. This is sufficient to study phenomena such as temperature gradients across different floors.

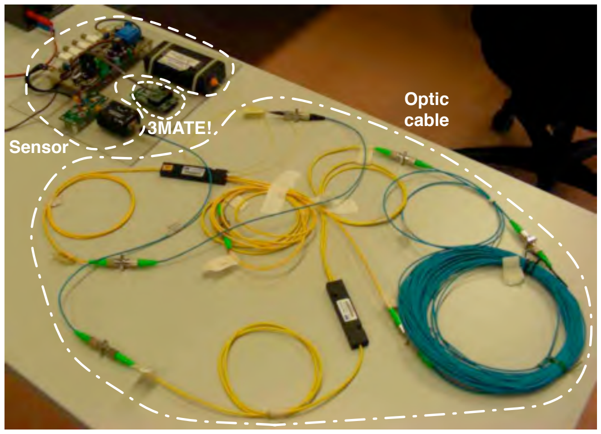
Deformation nodes. To study the tower deformation, we required a minimally-invasive solution with very high precision. We developed a dedicated Fiber Optic Sensor (FOS)



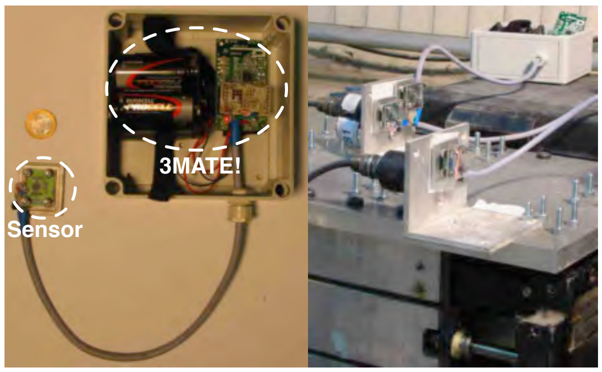
(a) 3MATE! node.



(b) Gumstix device as sink.



(c) Fiber optic sensor.



(d) Acceleration node and calibration.

Figure 2: Custom WSN hardware for Torre Aquila.

and the corresponding 3MATE! extension board, both shown in Figure 2(c). The sensor and its microcontroller-based control electronics, developed by TRETEC and University of Trento, work by differentially measuring the time taken for a laser pulse to travel through a pair of fiber optic cables wrapped around the monitored object. As the latter deforms, the cable stretches, modifying the travel time of the pulse. This solution is immune to electromagnetic noise and can be used to measure deformation on different physical scales, e.g., from individual walls to entire buildings.

The FOS is composed of a read-out unit with a synchronous laser pulser and a high-resolution optical receiver, and the optical path formed by fiber optic cables and splitters. Differently from all other sensors in Torre Aquila, the characteristics of FOS electronics require external power to ensure a stable measurement. The expansion board contains also a temperature sensor similar to the ones above, useful to correlate deformation with temperature in the same location.

Acceleration nodes. To measure vibration we used an analog, ultra-compact, tri-axial acceleration MEMS sensor (ST LIS3L02AL), integrated on a custom 3MATE! board connected through an extension cable that allows the sensor to be placed outside the node package, as illustrated in Figure 2(d). The sensor features a full range of $\pm 2\ g$ and is capable of measuring accelerations over a bandwidth of 1.5 KHz, with a resolution of 1 mg over 100 Hz bandwidth. We computed calibration coefficients for each sensor with induced vibrations at different frequencies and amplitudes using a shake table and piezoelectric accelerometers for seismic vibrations, shown on the right of Figure 2(d).

High-volume data such as vibration pose severe demands on buffering space. Some deployments [10] use the flash chip on the mote as a temporary buffer. However, this is a viable option only if the system operates for a limited time span, as the bound on the number of write operations eventually results in corrupted data. Instead, we equipped the 3MATE! with a FRAM chip, shown in Figure 2(a). Compared to flash memory, FRAM features lower power consumption, virtually unlimited write-erase cycles, and faster write speed, enabling higher sampling rates. In our experiments, the flash could sustain at most 500 Hz sampling, whereas the FRAM allowed up to 1 KHz. Nonetheless, the storage area provided by FRAM is generally smaller than flash. In our case this is not an issue, as we use our 32 Kbyte FRAM as a temporary buffer, freed progressively as data is reported to the sink, described next.

Sink node. In Torre Aquila, the sensed data converge from all nodes to a sink where they are collected and stored, requiring a computing device with enough storage space and processing power. Moreover, this device must double as a gateway to interconnect with the front-end, allowing remote users to interact with the system. Finally, the requirement to reduce invasiveness holds also for the sink.

To address these needs, we chose a Gumstix [8] device, shown in Figure 2(b). Gumstixs are easily customizable embedded PCs with a very small form factor. We equipped ours with a board to use Secure Digital (SD) storage cards, a WiFi card to reach the external network, and a USB board for connecting a 3MATE! to access the WSN. As shown in Figure 2(b), the space required for this configuration is very small: it uses the same packaging of the WSN nodes.

4. SOFTWARE DESIGN

The design of WSN software is often characterized by ad-hoc solutions built directly on top of the operating system. The consequence is that systems become difficult to maintain and reuse is hampered [19, 1]. In our deployment we took a different stand, and addressed since the beginning the challenge of designing the software layer through higher-level abstractions that simplify development and foster code reuse.

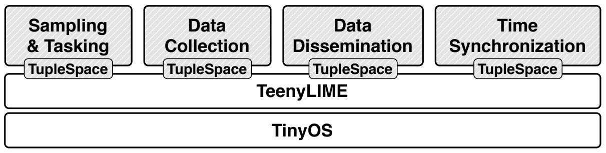


Figure 3: Software architecture.

Architecture. Figure 3 shows the high-level architecture of our software layer. The various macro-components interact *exclusively* through a shared memory space where data is read or written as tuples, sequences of typed fields. The tuple space abstraction is provided by a middleware called TeenyLIME [5], concisely described next. Its constructs are used to implement both application-level functionality (e.g., sensor sampling) and system-level mechanisms (e.g., routing and time synchronization), providing a unifying high level of abstraction throughout the software stack.

The reliance on this shared tuple space yields a highly decoupled software configuration, boosting code reuse both within and across deployments. For instance, the software deployed on acceleration nodes differs from that of environmental nodes *solely* in the sampling functionality, which inevitably depends on the quantity to sense. Moreover, it makes it easier to design alternative deployments by removing or replacing components, without affecting the others.

TeenyLime in a nutshell. As shown in Figure 4, in TeenyLIME each node hosts a tuple space *shared* among 1-hop neighbors: a node perceives its tuple space as containing the tuples stored locally plus those residing on its neighbors. Software components atop TeenyLIME interact locally or across nodes by reading/writing tuples from/to the shared tuple space. If needed, however, the read/write operations can be scoped to access directly the local tuple space of a neighbor. Read operations occur by requesting a match against a *pattern*: its fields express a constraint on the field type or value in the tuples being considered for matching. For instance, a pattern ⟨"foo", ?integer⟩ matches the tuple ⟨"foo", 20⟩ but not ⟨"foo", "boo"⟩. Moreover, TeenyLIME provides a form of data listener called a *reaction*, a code

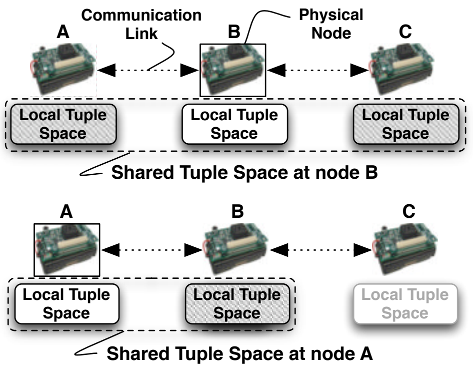


Figure 4: Tuple space sharing in TeenyLIME.

Node type	Operating parameters	Typical value
Environmental	Sampling period P	10 min
	# of sampling sessions N	infinite
Deformation	# of samples averaged per session A	10
	Sampling period P	10 min
	# of sampling sessions N	infinite
Acceleration	Sampling frequency F	200 Hz
	Sampling duration D	30 s
	# of sampling sessions N	infinite

Figure 5: Node types and their typical configuration.

fragment whose execution is automatically triggered upon the appearance of a matching tuple in the shared tuple space. This provides a very powerful way to increase the decoupling among different functionality. Other TeenyLIME constructs are described in the following, whenever appropriate. TeenyLIME is implemented in nesC on top of TinyOS. Therefore, operations are asynchronous and their result is signalled to the caller component through an event. A complete description of the middleware, including API and implementation details can be found in [5].

We now describe the design of the main components in Figure 3. In every case, we first highlight the requirements and challenges, and then report on the component design and implementation in TeenyLIME.

4.1 Sampling and Data Collection

Requirements and challenges. The deployment in Torre Aquila is characterized by heterogeneous sensor nodes whose sampling requirements and modalities vary greatly, as seen in Figure 5. This affects not only the local processing, but also the routing protocols employed for data collection, where reliability guarantees also play a key role. Based on our scenario, we identify two classes of traffic for data collection:

I. *Bursty, high-rate* data with *strong reliability* requirements, i.e., those coming from acceleration nodes. Large amounts of data are locally stored in a buffer whose elements are all sent in a burst after the sampling session. In this case, the loss of samples can impair the accuracy of the signal reconstruction, and therefore the analysis. Moreover, the volume of data generated requires compression, to reduce the amount of data transmitted and extend lifetime. This poses an additional reliability requirement, as it is impossible to decompress the stream if some of its packets are missing.

II. *Low-rate* data with *weak reliability* requirements, i.e., those coming from environmental and deformation nodes. Even if one sample is occasionally lost, a meaningful data analysis can still be carried out.

Our system also supports best-effort delivery of system data (e.g., battery status) whose loss is not critical. We could design a solution only for the most demanding class I, and use it for all data collected. However, this would constitute a waste of resources. Therefore, we designed a solution able to accommodate each of the above requirements efficiently.

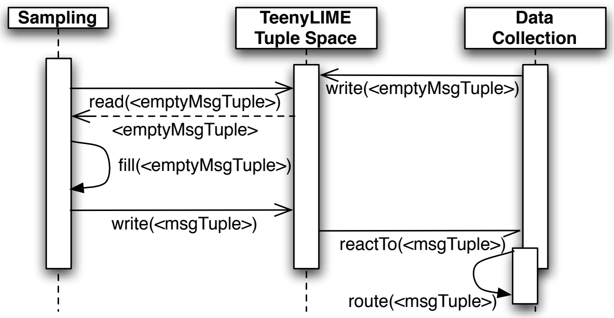


Figure 6: Handing sampled data over for routing.

Design and implementation. The sampling of environmental and deformation nodes is straightforward. The only peculiarity of deformation is that a single sample is usually not relevant, as values tend to fluctuate: thus, the data communicated to the sink is actually an average of the last A samples.

Instead, acceleration nodes add significant complexity due to the high volume of data sampled. Each of these nodes buffers the data of an entire sampling session on FRAM. The availability of the entire data set allows us to apply a Huffman [9] compression scheme to reduce the amount of data transmitted. It is important to note that, unlike other compression schemes mentioned in the literature (e.g., wavelets in Wisden [4]), Huffman is *loss-less* and therefore preserves the semantic richness of the vibration data [14]. The effectiveness of compression, however, greatly depends on the statistical properties of the data set. We observed that different nodes and acceleration axes produce data with different properties, which can be exploited in the Huffman scheme. Therefore, we developed a compilation tool-chain that, using as input the (real) uncompressed data from a node/axis, automatically generates the optimized compression code to be used on that node. This procedure requires an extra step during system deployment, but achieves remarkable improvements in the resulting compression, as discussed in Section 6.

At run-time, sampled data is encoded in a tuple that is shared by the sampling component, through TeenyLIME, with the data collection component of Figure 3. The coordination among the two takes place as shown in Figure 6. The sampling component queries the tuple space for an "empty" tuple, indicating the availability of a transmission slot: we describe next how and when this is generated. If such a tuple exists, it is removed from the tuple space, filled with the data to transmit, and output back to the tuple space. Through a previously-installed reaction the data collection component, notified of the presence of the data tuple, can withdraw it and begin the processing necessary for routing.

Our routing protocol builds a tree topology rooted at the sink. The tree is periodically rebuilt to account for connectivity changes. The process is performed by flooding a special control tuple. Each node re-propagates the tuple by writing

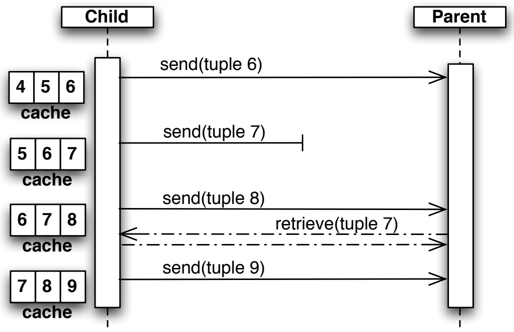


Figure 7: Hop-by-hop recovery example.

a copy of it in the tuple space of every node within communication range. There, the appearance of the tuple triggers a previously-installed reaction, which updates the tuple content with path cost information and repeats the process, eventually flooding the entire system. This flooding mechanism is reused also by other components, as mentioned later.

The reliability metric we use in optimizing the shape of the tree is a variant of [21], based on the Link Quality Indicator (LQI) provided by the radio chip. Interestingly, the LQI value is also accessed through TeenyLIME, using special tuples whose field values are materialized by the runtime, as described in Section 4.4. Finally, data forwarding occurs through the tuple space, by writing tuples to the tuple space of the current parent in the tree.

The reliability requirements of the aforementioned class I and II are dealt with through a hop-by-hop recovery scheme, intuitively described in Figure 7. Sent tuples are kept in the local tuple space, which effectively serves as a local cache, managed as a circular buffer. The receiving parent in the tree keeps track of the last tuple received from each child, thanks to a sequence number included in it. Upon recognizing a hole in the sequence, the parent pulls the missing tuple from the child's cache, using a read operation. The child node is totally oblivious of recovery: no dedicated processing is required, as the necessary operations are performed directly by the parent through TeenyLIME.

Since it is localized, fully distributed, and does not require system-wide flooding of recovery information, our reliable protocol enjoys lower latencies and far less network overhead than end-to-end, centralized solutions such as [10]. On the other hand, it might fail if a tuple is lost right before a node changes its parent. Consider a node C switching its parent from P_{old} to P_{new}. In this situation, P_{new} has no information about tuples previously sent by C, and cannot detect a tuple lost during the switch. These cases do occur in practice: Figure 8 shows a lab experiment where the tree is rebuilt every 2.5 minutes, and the occasional tuple losses occur *only* in coincidence with such tree reconfigurations.

Situations like the above must be avoided for class I traffic, which requires 100% delivery. They are taken care of in our protocol with a simple, yet effective, mechanism. Whenever the sink recognizes the beginning of a burst of class I traffic,

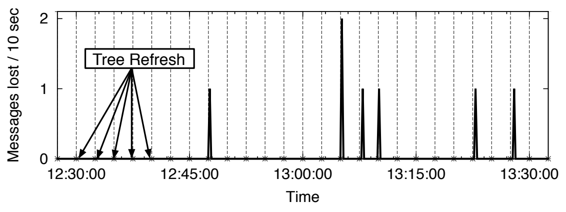


Figure 8: Lost tuples and tree refresh operations.

the time scheduled for the next tree rebuild is temporarily set to infinite. This effectively prevents the tree from changing while class I traffic is routed towards the sink, and thus removes the source of the problem.

Our implementation also considers transmission schedules. Traffic of class II is scheduled opportunistically. In the case of class I traffic, however, a network congestion may develop due to the high volume of data transmitted. To alleviate the problem, we employ a form of slow-start scheduling for class I traffic, varying the inter-message period at which the empty tuple representing an available transmission slot becomes available. When a transmission failure is detected, the inter-message is set to the highest value then slowly decreased, up to a configured minimum, as data are successfully forwarded to the sink. With a minimum inter-message interval of 1 s, the reporting of a 30-second compressed sampling session at 200 Hz takes around 8 minutes.

4.2 Time Synchronization

Requirements and challenges. To investigate the dynamics of Torre Aquila, the readings taken by different nodes must be correlated w.r.t. time. This is especially true for vibrations, e.g., to study how forces applied at the base of the tower propagate to the top floor. The samples must be aligned in time, with a worst-case time drift up to 1 ms [13].

Design and implementation. Several time synchronization protocols for WSN exist. To meet the requirement above, our solution is a modified version of [6]. The protocol works by creating a hierarchy among the network nodes, whose clocks are then synchronized with the root's clock. As depicted in Figure 9, synchronization is based on a round-trip tuple exchange between nodes at level i and $i - 1$ in the hierarchy. The nodes at level i record the time T_1, at which a synchronization request is issued, and T_4, at which the reply from a node at $i-1$ is received. This reply contains the times T_2 and T_3 at which the node at $i-1$ received the request and replied to it, respectively. These four values enable the nodes at the lower level i to evaluate clock drifts and propagation delays, and adjust consequently their local time w.r.t. nodes closer to (and therefore with a smaller drift from) the root at level 0. Since this process is performed at each hierarchy level, it eventually synchronizes all nodes to the root.

As with the other services, we implemented time synchronization using TeenyLIME. The hierarchy is built trivially by relying on the same flooding mechanism described for data

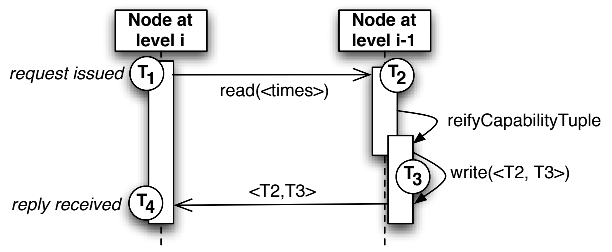


Figure 9: Time synchronization using capability tuples.

collection in Section 4.1. However, the information flooded (and therefore the resulting tree) is different, since data collection optimizes the tree shape w.r.t. link quality, while time synchronization minimizes the hop-count from the sink to reduce the impact of the link latency on the time estimate.

Instead, pairwise synchronization among nodes relies on one of TeenyLIME's unique constructs: *capability tuples* [5]. A capability tuple is essentially a placeholder for the actual data, which is generated on demand. As illustrated in Figure 9, when a read operation whose pattern matches a capability tuple is received, TeenyLIME does not simply return, as usual, the latter as result. Instead, it delegates its computation to the component that originally output the capability tuple, using a `reifyCapabilityTuple` event. This is handled by computing and outputting the actual content of the tuple, which is then finally delivered to the query issuer by TeenyLIME. This mechanism essentially enables a node to "advertise" the availability of data without the need to keep it up-to-date by periodically regenerating it—a waste of energy when not used by any query.

In our time synchronization component we use a capability tuple to produce on demand the values of T_2 and T_3, as illustrated in Figure 9. It is worth noting that most of the distributed processing is dealt with by TeenyLIME, greatly simplifying the implementation.

Of course, threats to accuracy may come from the unpredictability of processing and message transmission delays. Solving this issue actually led to extensions to the original TeenyLIME API. To alleviate the first problem, we enabled components to be notified when a given operation (e.g., a message send) is completed. This information is used by the synchronization component to periodically re-evaluate processing delays. Message transmission delays, instead, are kept under control by temporarily switching off the radio duty-cycling during a synchronization round. This is achieved by using a newly-designed *tuning interface*, which enables cross-layer interactions by giving developers direct control over the node hardware.

Evaluating precisely the accuracy of our protocol is difficult in the deployment environment. Therefore, we performed a number of lab experiments, using 12 nodes in a chain topology. We used a Tektronix TDS 220 two-channel oscilloscope to measure the time drifts between any two nodes in the network. As expected, the worst-case time drift happens between the root of the tree and the node at the opposite end of the chain. In this case, the time difference was 732 μs, still sufficient to perform meaningful analysis of vibration data [13]. Moreover, Section 6 reports that in the deployment we observed at most 6 hops between an acceleration node and the sink. It is therefore unlikely that time drift in Torre Aquila is higher than in our lab experiments.

4.3 Tasking and Data Dissemination

Requirements and challenges. The *ideal* configuration of the monitoring system deployed in Torre Aquila, in terms of acquisition rates and intervals, is not known a priori, as often happens when WSNs are employed to study a physical phenomenon for the first time. Moreover, in many cases an external event may suggest a different configuration. For instance, it could be of interest to monitor more frequently vibration and deformation when roadwork is being conducted nearby, people are present in the tower, or strong winds are present. The ability to remotely task the system must be supported by a mechanism that disseminates the new configuration *reliably*, and guarantees that the received data is *eventually consistent* across the system.

Design and implementation. The set of sampling parameters that can be modified remotely are those shown in Figure 5. There, we included the values suggested by the structural engineers on our team: each parameter, however, can be changed independently. In particular, the number of sampling sessions N can be a finite number, enabling monitoring of a given quantity only during a given time interval.

A parameter configuration is packed in a *task* tuple with an appropriate format. These tuples are generated on the sink upon a user request, issued through our graphical front-end, and disseminated using the protocol we describe next. On every node, the sampling and tasking component (Figure 3) registers a reaction matching task tuples and, upon receipt of a new one, updates the sampling parameters accordingly.

The task tuples must be disseminated reliably throughout the system, a widely studied problem in WSNs [11, 12]. We take inspiration from the state-of-the-art by adapting the Trickle [11] protocol. This achieves eventual consistency of the disseminated data by using monotonically increasing sequence numbers, used to determined if a node is up to date.

This dissemination scheme lends itself to a straightforward implementation on top of TeenyLIME. Task tuples are initially flooded by using the mechanism described for data collection in Section 4.1. Moreover, the management of missed tuples comes almost for free by using one of TeenyLIME's constructs: *node tuples* [5]. A node tuple represents the current state of a device, and is made available inside its 1-hop neighborhood. The format of the tuple and the rules for populating its field values are provided by the programmer, but the periodic update of these values and the tuple propagation to neighbors is carried out automatically by the TeenyLIME run-time. Therefore, checking whether a recovery is needed in our dissemination protocol is as simple as including the sequence number as a field in a node tuple; installing a reaction that fires whenever a neighbor's sequence number is newer than the local one; and recovering the missing tuple with a read operation on such neighbor.

4.4 TeenyLime: Deployment-driven Enhancements

The requirements of the Torre Aquila deployment brought the development of TeenyLIME one step ahead. We already mentioned some of the extensions we designed, e.g., the tuning interface in Section 4.2. Below is a summary of other enhancements to TeenyLIME motivated by our deployment.

Typed tuples and dynamic memory. In the presence of high-rate data such as vibrations, it is imperative to manage efficiently the available memory. To further optimize this aspect in TeenyLIME, we introduced a notion of *typed tuple*. Mimicking the generic data types in modern programming languages, developers instantiate tuples as:

```
tuple<uint8_t, uint16_t, float> temperature =
  newTuple(actualField(TEMPERATURE_TYPE),
           actualField(NODE_ID),
           actualField(temperatureReading));
```

where `actualField` indicates a field with actual data, as opposed to constraints on the field type or value. A preprocessor we developed inspects all TeenyLIME-based application components to gather a complete view of all tuples used. Based on this, it generates optimized data structures for storing and searching the data.

Typed tuples are managed at run-time by a component providing a form of dynamic memory based on *slabs* [2]. In our case, a slab is a chunk of memory meant to store tuples of the same size. Using slabs does not require de-fragmenting memory, which is difficult to implement on resource-scarce devices. In the application described here, the combination of the techniques above freed 80% of the memory allocated by our previous release of TeenyLIME.

Automatic field types. Our data collection component relies on LQI as a measure of link reliability. To relieve the programmer from the burden to explicitly query the operating system for similar low-level information, we make it available in the form of tuples by defining a number of special field types whose value is automatically materialized by TeenyLIME as part of the node tuples. For instance, in:

```
NodeTuple<uint16_t, lqi> myNodeTuple;
```

the value of the second field of the node tuple reflects the LQI value towards a particular neighbor. This way, low-level data becomes straightforwardly available to the application, greatly simplifying the development of routing protocols.

Reliable, low-power operations. In TeenyLIME, programmers can explicitly choose whether the execution of a remote operation is reliable or not. In our deployment, this feature is support by a dedicated reliable communication layer exploiting mixed software/hardware link-layer acknowledgements. This solution occupies only 252 bytes of program memory.

To provide low-power operations, we integrated in our run-time the Low Power Listening [20] layer available in the TinyOS distribution. TeenyLIME's operating parameters (e.g., the timeout for remote queries) are exposed to make them adjustable based on the expected message delays.

5. DEPLOYMENT

The tower contains four floors, the ground one isolated from the others and used as a public walkway. The plan is C-shaped 7.8 m × 4.5 m, and the height is 25.6 m. The 14th century enlargement closed the tower to the West and raised the gate by an additional storey. The two parts of the masonry body have completely different properties. The lower level walls consist of two 40 cm thick stone blocks, with an incoherent filling. At the upper levels, the older portion of the masonry is built of 80 cm thick stone blocks, while the most recent one is brick and blocks of varying sizes. Visitors enter the tower from the nearby Buonconsiglio Castle, arriving through a long corridor directly on the 2nd floor where the frescoes are.

Node placement. As shown in Figure 10, we deployed 16 nodes plus the sink #0. This is placed at the top floor, the only spot guaranteeing access to the external WiFi network.

The sensor position is chosen to detect early symptoms of deterioration of the structure. The joint between the ancient parts of the tower and the more recent ones is today perfectly visible (bottom of Figure 10), but the degree of structural

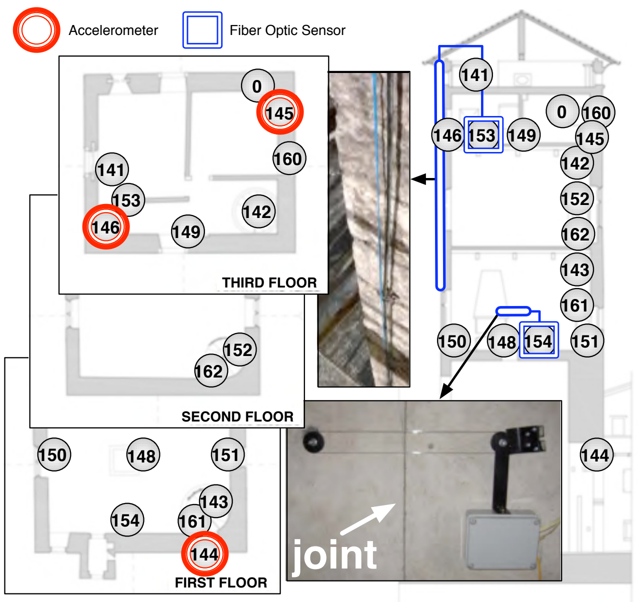


Figure 10: Deployment map.

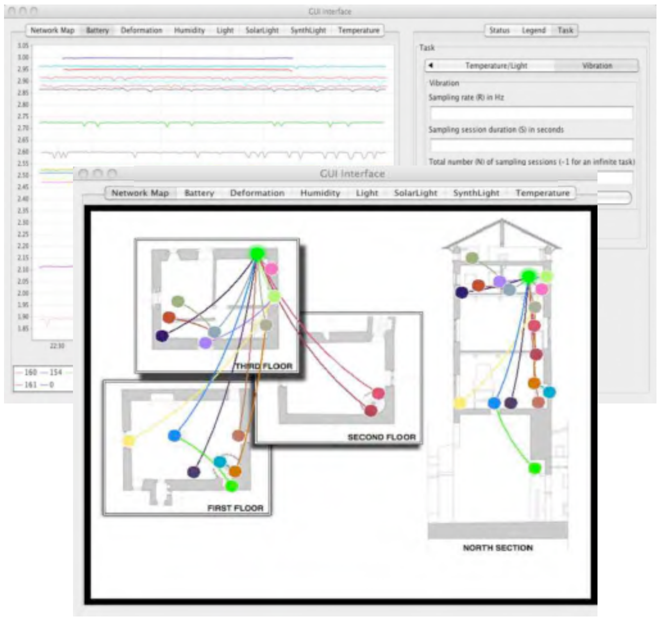


Figure 11: Graphical user interface.

connection of this joint is still a major point of uncertainty. The deformation across the connection is measured on the 1st floor by FOS #154. This is a 0.6 m gauge wrapped as an optical coil to magnify the sensor precision, and anchored to two expansion bolts at the sides of the joint, as shown in Figure 10. Another FOS is used to detect vertical elongation at the S-W corner of the tower, from level +5.7 m to +25.6 m. In this case the measuring path is a protected optical fiber loop pre-tensioned between two metal anchorings. An extension cable connects the sensor to node #153 at the 3rd floor.

The vibrations induced by traffic and, to a minor extent, by wind are recorded by acceleration nodes #144, #145, and #146, the first at the base and the others at the top of the tower. The analysis of acceleration readings allows to understand the dynamical behavior of Torre Aquila. Indeed, the vibration response of a building is not completely random, but concentrates mainly around some specific frequencies, know as natural frequencies. Daily and seasonal thermal excursions also affect the structural response of the tower, and the knowledge of these variations is needed to process and compensate the strain and acceleration signals recorded by FOS and accelerometers. This motivates the presence of a number of environmental nodes distributed all over the tower.

Data visualization and access. Effective access to the information gathered by the system is crucial in supporting the structural engineers in their analysis. To this end, we provide a custom graphical user interface, shown in Figure 11, implemented through a major re-factoring of Octopus [16]. The GUI shows the current network topology and serves as a control center from which the user can remotely task the WSN. Moreover, it displays the data collected, which are also persistently stored in a database.

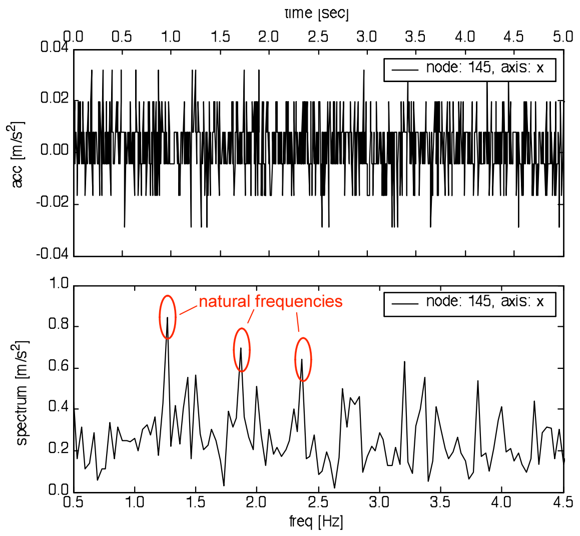


Figure 12: The acceleration signal from #145.

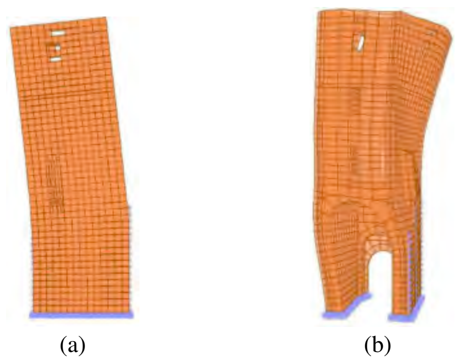


Figure 13: First (a) and second (b) vibrational modes.

Preliminary data analysis. The data collected is processed by a Bayesian algorithm that provides the user with the real-time probability of an ongoing structural disease. The algorithm can identify a hazardous condition many days in advance w.r.t. to the actual occurrence of the damage [23], and it has already been applied for risk analysis of historic buildings [24]. In the following, we provide a few examples of collected data and discuss the insights that the structural engineers on our team gained from them.

Figure 12 shows the acceleration measured on the X axis of #145. The top chart reports the time history over 5 s, while the bottom one shows the corresponding frequency spectrum. The peaks in the spectrum indicate possible natural frequencies of the structure, at 1.25 Hz, 1.80 Hz and 2.40 Hz. Every natural frequency follows a specific deflection shape, usually referred to as vibrational mode. For instance, Figure 13 shows the first two vibrational modes of the tower computed by a numerical model, respectively associated to natural frequencies of 1.25 Hz and 1.80 Hz[1].

Figure 14 reports the strain measured, in microstrains ($\mu\epsilon$),

[1]For sake of clarity, in the picture the amplitude of the vibrational modes have been artificially magnified.

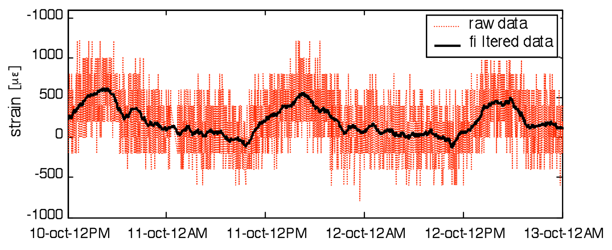


Figure 14: Strain measurements from FOS #154.

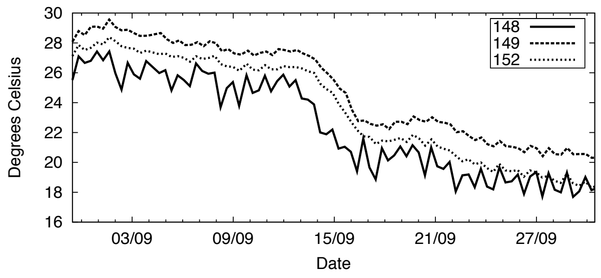


Figure 15: Temperature on three floors of Torre Aquila.

by the FOS #154 placed across the joint. To eliminate the high frequency instrumental noise, we applied to the signal a moving average filter with a 60-sample long window. The graph shows the well-known "breath" of the structure due to daily thermal variations. The joint is forced to open because of thermal expansion when sun rays hit the southern facade, and then closes during the night. We also note a delay between the maximum irradiation at mid-day and the maximum joint elongation, presumably caused by the thermal inertia of the walls. The analysis of this behavior also allows assessing the sensitivity of the joint to temperature. As for the latter, the temperature data shown in Figure 15 for nodes on different floors confirm the presence of a gradient along the tower, as well as significant seasonal changes. The daily strain variation (on the order of 500 $\mu\epsilon$) agrees with the numerical prediction under the assumption that the joint is fully released. To date, the strain response of the tower has not shown trends which may rise concerns about its stability.

The benefit of the above analysis is twofold: on one hand, in the short-term it permits identification of a reliable model for the structure response, and prediction of the behavior of the tower during exceptional events, e.g., earthquakes or subsiding. On the other hand, the data are stored in a database that will remain available in the long-term and constantly compared with more recent data, so that any change in the tower behavior can be detected, triggering specific analyses.

6. EVALUATION

In this section, we study the effectiveness of our design along two lines. We report first on the system performance in Torre Aquila, showing that our solution performs reliably and efficiently. Next, we consider the benefits of using a middleware during the development process.

6.1 System Performance

To assess the effectiveness of our middleware-based design we report on three key performance issues: i) reliable delivery of data, ii) effective compression of acceleration readings, and iii) energy consumption and system lifetime.

Reliable delivery. During the last four months of operation, the overall loss rate always remained below 0.01%. This performance is striking if compared to the average yield of long-running WSN deployments reported in the current literature [1], and even more so if we consider that ours is one of the few WSN deployments featuring high-rate data reporting for more than a few weeks.

The effectiveness of our reliability mechanisms for traffic of class I and II is exemplified in Figure 16, showing the cumulative loss rate (in log scale) over time. The loss rate for class I traffic generally remains an order of magnitude lower than that of class II traffic. In the morning of September 3rd a malfunctioning acceleration node lost a number of tuples, which generated the spike relative to class I traffic. Later on the same day we replaced the faulty node and performed a few maintenance operations on the sink, temporarily suspending its operation. This caused the spike in class II traffic. After these two events, the loss rate decreased steadily.

This performance is achieved in spite of the peculiar characteristics of the deployment scenario. Although Torre Aquila is not particularly tall, the thickness of its walls greatly hinders wireless propagation. As an indication of this, Figure 17 reports the percentage of time some nodes spent at

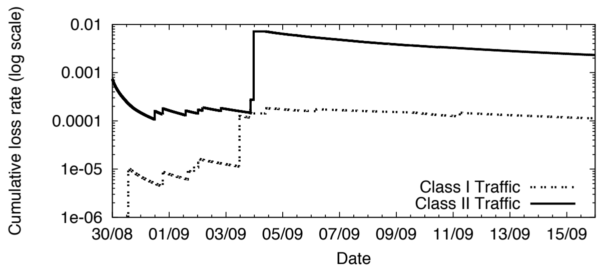


Figure 16: Cumulative loss rate.

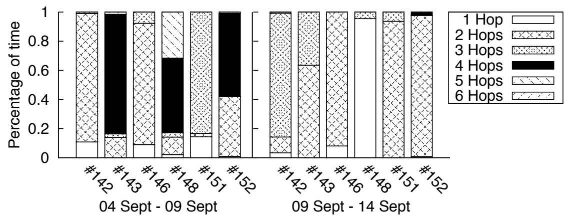


Figure 17: Distance in hops from the sink.

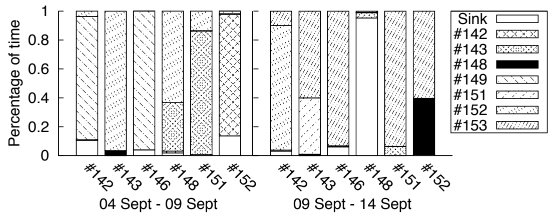


Figure 18: Time spent with a given parent node.

a given distance from the sink. Notably, the latter in some cases reaches the value of 6 hops. Moreover, we observed how small changes in the node placement drastically change the connectivity. Figure 17 shows two periods: in the latter we moved the sink because of some restoration work taking place in the tower. Although the sink was moved at most by 1 m, the topology drastically changed: for instance, #148 became able to reach the sink directly for most of the time, rather than through the 4-5 hops experienced previously.

In any case, our data collection protocol adapts effectively to topology changes. For instance, Figure 18 shows how, in the context of the same sink movement, nodes select a new, better parent. However, topology changes are more frequently induced by connectivity fluctuations caused by people visiting the tower and humidity gradients: the reaction to these common causes is equally effective. For instance, we observed nodes relying on up to 4 different parent nodes, according to the observed link reliability.

Compression. We used an Agilent 34411A digit multimeter to measure the processing time over 166 sampling sessions of 30 s at 200 Hz, for a total of ~1,000,000 raw acceleration samples. Although the code was not optimized for this data set, the worst compression time was 17.32 ms, which supports our choice of Huffman coding and confirms the efficiency of the compression code we generate automatically.

Our tool-chain also enables optimization of the compression scheme according to the specific node (i.e., position) and axis. In Torre Aquila, this brings considerable advantages w.r.t. a compression tuned using all acceleration samples regardless of their source and axis, as illustrated in Figure 19. Interestingly, the maximum improvement is achieved by generating the custom compression code for the Z axis. Indeed, this axis is subject to the gravitational field, and therefore its values are rather different from those of the X-Y axes: a dedicated compression scheme better captures the statistical properties of the corresponding data sets.

Energy consumption and lifetime. We observed that energy consumption essentially depends on the node functionality, as shown in Figure 20 using battery voltage. Acceleration nodes draw more current than environmental ones: not only are they used more intensively, but they must also continuously power the FRAM chip. Consequently, acceleration nodes deplete their available energy more rapidly.

Input Node	Input Axes	Compression Ratio	Reduction in Data Traffic
All	All	17.9%	17.9%
144	All	31.45%	27.7%
145	All	24.91%	
146	All	26.76%	
144	X-Y	47.11%	51.23%
	Z	69.34%	
145	X-Y	41.65%	
	Z	64.66%	
146	X-Y	43.56%	
	Z	62.43%	

Figure 19: Compression ratios with different input sets.

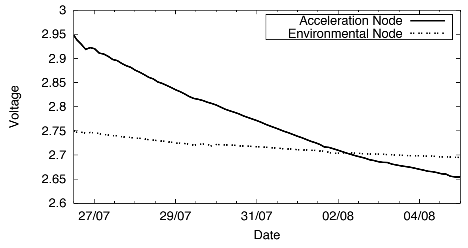


Figure 20: Battery voltage readings.

Component	Lines of code
Sampling & Tasking	235–962
Data collection	993
Data dissemination	339
Time synchronization	916

Figure 21: Lines of code for our core components.

Estimating the expected system lifetime of our system is tricky due to the non-linear behavior of commercially available batteries [17]. The first version of the system used a radio duty-cycle of 100 ms and used the on-board LEDs for debugging. Under these conditions, and using one pair of size C batteries, we observed *one* node dying after 3.2 months of operation. The system is currently operating with a radio duty-cycle of 250 ms, yielding the same reliability. Moreover, our packaging can accommodate two pairs of size C batteries. Assuming the single dead node as a worst case, we expect the system lifetime to extend beyond one year.

6.2 Beneficial Impact of Middleware

We discuss the impact of our middleware-based design on programming effort and re-usability.

Programming effort. Quantifying the programming effort is hard, as it is affected by factors difficult to measure (e.g., the complexity of the processing). Research in WSNs has hitherto considered the number of lines of code (LOC) as a simple indication. Figure 21 reports this metric for the core functionality of our system. It is interesting to compare these figures against similar functionality available in TinyOS libraries, where it is built directly on top of the OS. The CTP [20] collection protocol and the DIP dissemination protocol [12] have almost twice as many LOC as our solutions, and yet the former addresses only low-rate data. The original implementation of the time synchronization protocol [6] contains 80% more LOC than our version. We maintain that the significant reduction in LOC is achieved by delegating part of the processing to the middleware. For example, most of the recovery processing in our data collection component takes place within TeenyLIME, as described in Section 4.1. Parsing recovery requests, finding the message to be re-sent, and re-trying the transmission are captured by a *single* remote read operation.

Decoupling and re-usability. The use of TeenyLIME fosters *asynchronous* and *data-centric* interactions, which increases decoupling. As a result, the design for Torre Aquila can be easily extended to meet different requirements. For example, consider adding distributed data aggregation. This functionality is usually embedded within routing, resulting in the two becoming entangled. Instead, in our design this would require no modification to the data collection component. It is sufficient to tag differently the tuples carrying raw data, and make the new data aggregation component react to them. Aggregated data would then be output as message tuples triggering a reaction in the data collection component, as already happens in our current design. All these changes would not even require a wiring of nesC interfaces.

The high decoupling is also beneficial w.r.t. memory consumption. The size of the binary image installed on our nodes ranges from 37 KB (environmental nodes) to 47 KB (acceleration nodes). The latter is close to the 48 KB limit on TMotes, but it is the most complex as it also includes the compression code. Using components from the TinyOS libraries to provide similar functionality (i.e. CTP, DIP, and the implementation of [6]) would yield a binary of at least 51 KB, which would not fit the program memory.

7. CONCLUSION AND FUTURE WORK

The preservation of the valuable frescoes of Torre Aquila requires real-time monitoring of structural response and environmental conditions. We demonstrated that a WSN-based monitoring system can achieve this goal thanks to highly reliable data delivery sustained over an extended time span. In addition to the customized hardware, this result was achieved by means of reusable and extensible software services built on top of the TeenyLIME middleware, demonstrating the benefits of high-level abstractions in a real-world deployment.

The project's current focus is on finalizing the analysis of the data gathered. We expect that this will require again the relocation of some nodes and changes in the sampling configuration, further exploiting the versatility of our system. Our next step is to apply our hw/sw system to the monitoring of other heritage buildings, therefore verifying experimentally the flexibility and re-usability of our design.

A project Web site is available at `d3s.disi.unitn.it/projects/torreaquila`.

Acknowledgments. This work was partially supported by the Italian Ministry of Education (MIUR) under project PRIN06-2006084179_003. The authors wish to thank Christian Benoni, Massimo Cadrobbi for their work on time synchronization and the finite elements model, respectively, and Prof. Giovanni Soncini for his support in developing the FOS read-out system.

8. REFERENCES

[1] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli. The hitchhiker's guide to successful wireless sensor network deployments. In *Proc. of the* 6rd *Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2008.

[2] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, 1994.

[3] K. Chintalapudi, T. Fu, J. Paek, N. Kothari, S. Rangwala, J. Caffrey, R. Govindan, E. Johnson, and S. Masri. Monitoring civil structures with a wireless sensor network. *Internet Computing*, 10(2), 2006.

[4] K. Chintalapudi, J. Paek, O. Gnawali, T.S. Fu, K. Dantu, J. Caffrey, R. Govindan, E. Johnson, and S. Masri. Structural damage detection and localization using netshm. In *Proc. of the* 5th *Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2006.

[5] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco. Programming wireless sensor networks with the TeenyLIME middleware. In *Proc. of the* 8th *ACM/USENIX Int. Middleware Conf.*, 2007.

[6] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync protocol for sensor networks. In *Proc. of the* 1st *Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2003.

[7] O. Gnawali, K. Jang, J. Paek, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler. The Tenet architecture for tiered sensor networks. In *Proc. of the* 4th *Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2006.

[8] `www.gumstix.com`.

[9] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of IRE*, 40(9):1098–1101, 1952.

[10] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *Proc. of the* 6th *Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2007.

[11] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. of the* 1st *Conf. on Networked Systems Design and Implementation (NSDI)*, 2004.

[12] K. Lin and P. Levis. Data discovery and dissemination with DIP. In *Proc. of the* 7th *Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2008.

[13] J. P. Lynch and K. J. Loh. A summary review of wireless sensors and sensor networks for structural health monitoring. *Shock and Vibration Digest*, Mar 2006.

[14] J. P. Lynch, A. Sundararajan, K. H. Law, A. S. Kiremidjian, and E. Carryer. Power-efficient data management for a wireless structural monitoring system. In *Proc. of the* 4th *Int. Wrkshp. on Structural Health Monitoring*, 2003.

[15] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proc. of the* 1st *Int. Wkshp. on Wireless Sensor Networks and Applications*, 2002.

[16] Octopus Home Page. `http://csserver.ucd.ie/~rjurdak/Octopus.htm`.

[17] C. Park, K. Lahiri, and A. Raghunathan. Battery discharge characteristcs of wireless sensor nodes: An experimental analysis. In *Proc. of the IEEE Int. Conf. on Sensor and Ad-hoc Communications and Networks (SECON)*, 2005.

[18] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *Proc. of the* 5th *Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2005.

[19] B. Raman and K. Chebrolu. Censor networks: a critique of "sensor networks" from a systems perspective. *SIGCOMM Comput. Commun. Rev.*, 38(3), 2008.

[20] TinyOS Official Source Tree. `www.tinyos.net`.

[21] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proc. of the* 1st *Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2003.

[22] N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *Proc. of the* 2nd *Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2004.

[23] D. Zonta, M. Pozzi, and P. Zanon. Manging the historical heritage using distributed technologies. *Int. Journal of Architectural Heritage*, 2(3), 2008.

[24] D. Zonta, M. Pozzi, P. Zanon, G. A. Anese, and A. Busetto. Real-time probabilisitc health monitoring of the portogruaro civic tower. In *Proc. of the* 6th *Int. Conf. on Structural Analysis of Historical Constructions*, 2008.