Abstract—The physically centralized nature of current building automation systems increases complexity and has limited scalability and fault tolerance. A building automation system which can scale to encompass a future Internet of things, needs an architecture which is decentralized from the lowest to the highest layers. We present Brume, a building operating system which logically provides the same kind of service oriented architecture as a modern building operating system, while physically it runs on the same fabric as a future Internet of Things; large numbers of small and heterogeneous devices. This enables Brume to be highly horizontally scalable and fault tolerant through means of redundancy, while also being secure and simple to operate. To verify the expected benefits and the applicability of the platform for building automation, a prototype is developed for a real building setup. A set of micro benchmarks are run against the prototype to evaluate various aspects of the design, and two cases are used to show the platforms ability to support real-time control of building equipment and supervisory control based on neural network-based predictions.

1. Introduction

Modern day large commercial buildings contain a myriad of technological systems which are installed to ensure the safety, security, comfort and productivity of building users. Integrated control of subsystems improves overall occupant comfort and efficiency of the building, and for many years BASs (Building Automation Systems) have been deployed towards this purpose. While over 90% of buildings in the US are below 50,000 square feet, most of them are not equipped with a basic BAS [1]. Buildings remain one of the largest energy consumers, consuming about 70% of all electricity and 40% of total energy consumption in the United States [2]. Yet, two out of three occupants are not happy with the indoor climate [3]. According to some estimates, poor building control results in a 30% waste of overall energy in commercial buildings annually, making energy efficiency a compelling incentive for new investments in BASs [4].

The definitions of smart buildings vary throughout literature [5], [6], [7], but most of them seem to acknowledge that the smart building of tomorrow is different from the current generation of automated buildings in its ability to collect, analyze, react upon, present and share unprecedented amounts of sensed data. The combination of low cost and flexibility, Wireless Sensor Networks (WSNs) and Internet of Things (IoT) are enabling massive increases in the amount of data which can be collected in a building. Exploiting the possibilities created by this increase of data, requires a system capable of collecting, storing and analyzing it. Both cloud solutions and existing BOSs (Building Operating Systems) (e.g., sMAP [8], BOSS [3] and BuildingDepot [9]) create bottlenecks by requiring central points of storage and computation.

In this paper we will argue that the physically centralized nature of existing BASs and modern BOSs increases complexity and has limited scalability and fault tolerance. A BOS which can scale to encompass the future of IoT, needs an architecture which is decentralized from the lowest to the highest layers. We present Brume, a BOS which logically provides the same kind of service oriented architecture as an existing BOSs, while physically it runs on the same fabric as IoT; large numbers of small and heterogeneous devices. This enables Brume to be highly horizontally scalable and fault tolerant through means of redundancy, while also being secure and simple to operate. The contributions of this paper are as follows:

- **The Brume architecture** which is inspired by the fog computing paradigm and data center design, and can be described as a small data center running in a local fog of commodity hardware and building equipment.
- **A concrete set of services for building automation** which are capable of running on Brume’s architecture. These services are inspired by modern BOSs designs, in particular the work presented in [3], and deliver the same kind of service oriented environment for developing building applications.
- **Evaluation results** based on micro-benchmarks of the prototype system that provide evidence for the scalability of the system to a common office building without a BAS and provided fault tolerance. Results with concrete prototype applications which test the systems ability to handle both real-time and supervisory building control and demonstrate that the system is simple to operate.

2. Related Work

The low adoption of building automation systems indicates that current technology is not good enough compared to the cost of deployment, since otherwise the economic
Incentives (often the largest drivers for change) would have aided in widespread adoption. Current systems are expensive to install in part due to the heterogeneous nature of buildings which requires custom solutions, and are difficult to operate, maintain and extend throughout the evolution and long lifetime of buildings.

**Building Operating Systems** can integrate various building systems, and provide a platform for integrated building control and portable building applications. However, the scale of IoT and its distributed nature, challenges the centralized architecture of current BOSs. Physically BOSs typically run as a single server setup, or as a multi-server setup where each server has a dedicated function (like for instance a dedicated database server). This is true across research-driven BOSs (e.g. sMAP [8], BOSS [3] and BuildingDepot [9]), open source community projects (e.g. OpenHAB [10]) and industry driven efforts (e.g., HomeOS [11]). This centralized physical setup increases complexity and has limited scalability and fault tolerance.

- Running a single server might appear as the simplest solution from an installation and maintenance perspective, however it requires a complex additional network infrastructure to be maintained in order to establish physical and logical (translation between different protocols) communication links from this central point to all peripheral devices.
- Creating driver modules for the BOS capable of interfacing with heterogeneous building equipment can be successfully tackled as a collaborative effort on open source projects. However, each driver has to be instantiated and configured for each device of the given type in order to accommodate the specifics of the on-site network infrastructure.
- Having a large number of independently executing driver modules perform network operations to collect data and control a vast sensor/actuator network can quickly exhaust both the network bandwidth and computational resources of a centralized architecture. Yet, ever more resources are needed to perform data analysis and gather insights from the vast amounts of raw data generated by IoT devices.

A recent effort has proposed a BOS implemented using the BossWave system to address some of these short comings by providing a microservice architecture with a decentralized security model [12].

**IoT and Cloud Computing** together enable systems that can scale in amount of data which can be collected and analyzed, while other issues fundamental to building automation remain unsolved. Meanwhile IoT and cloud computing introduce a number of serious issues regarding security, privacy and fault tolerance.

- Traditional security models do not work at the scale of IoT, and IoT security remains an open issue. Meanwhile new stories about hacked devices are appearing daily [13].
- Privacy issues related to cloud computing have long been subject of discussions, and with the rise of IoT an unprecedented amount of data is placed in the hands of third parties.
- The dependency on remote services requires an always online connection to the public Internet which raises concerns regarding fault tolerance. The service provider might go down temporarily or maybe even for good (E.g. in case of bankruptcy), which might render parts of the system inoperable.
- Questions are raised as to whatever the current cloud model can actually handle the volume, variety, and velocity of raw data generated by IoT devices [14]. For applications which need to react in real time the latency associated with communicating with a remote datacenter is a major issue.

**Fog Computing** is an extension of the cloud towards the edges of the network, and places a layer of intermediary fog nodes between the IoT devices and the cloud. The fog-computing paradigm attempts to bring services, processing and storage closer to where data is produced, analyzed and acted upon. In theory any device with computing, storage, and network connectivity can be a fog node which shares those resources [15]. In fog computing a hierarchical model is often envisioned where tasks not requiring real-time response, or which require vast storage and processing capabilities such as in depth data analysis are moved towards the datacenter [14], [16], [17]. The ideas of fog computing also have roots in the area of Cyber foraging which is a technique where resource poor devices offload heavy work to stronger surrogate machines [18]. Current challenges in fog computing include how to coordinate and program a heterogeneous and distributed fog [15], [16], [19].

In comparison, we in this work present Brume, a new decentralized BOS which logically provides the same kind of service oriented architecture as a modern building operating system, while physically it runs on the same fabric as IoT; large numbers of small and heterogeneous devices. It provides a high-level programming model in order to enable the development of applications running on a large number of heterogeneous devices distributed over a wide area. This enables programmers to write building applications which automatically scale and adapt to the dynamic environment of a building, without drowning in boilerplate code handling the complexity of distributed computing.

### 3. System Design Principles

The overall goal for Brume is to experiment with a completely distributed and decentralized platform for building automation. Rather than having security implemented on top of inherently insecure structures, we want to make security a fundamental and ubiquitous property of every system component. Instead of centralizing functions and integrating everything into powerful devices performing central coordination, we opt for an architecture which enables unified building control through means of coordination between components following simple rules. We choose this direction because we see that many of the current challenges stem
from the need to centralize functions. An exponential growth in computing power, has now produced cheap embeddable SOC{s} (System on a Chip) capable of running fully fledged operating systems, which potentially enable a different kind of fog inspired architecture. We imagine Brume running on new thick sensor nodes with resources similar to common single-board computers or nodes of commonly found devices in a building. This has lead us to the following design principles for Brume.

1) **Horizontal scalability** as it enable the system to scale automatically with the size of a building. Having a very large building means more data generated by building equipment. However, it also means equally larger amounts of distributed commodity hardware which can run as Brume nodes. Brume’s architecture and the design of each component is optimized to support heterogeneous devices.

2) **Fault tolerance** because each component of Brume runs on many nodes throughout the cluster, fault tolerance is achieved through means of redundancy. Brume operates with no single point of failure because it is completely decentralized and coordination is done on a peer to peer basis rather than by a master node. Except for nodes physically controlling hardware, functions of a node can be replaced by any other node when a failure is detected.

3) **Security** as Brume nodes can not be expected to run on secure closed networks. Thus all communication between Brume nodes needs per default to exhibit privacy, authenticity and integrity. These properties must be ubiquitous in Brume to reduce the risk of application developers making mistakes which results in security vulnerabilities, and to reduce the amount of complex and error prone work associated with implementing strong security. Since Brume nodes are spatially distributed throughout a building, Brume has to protect against attacks which can be mounted by physically tampering with nodes.

4) **No dependency on third party services** as Brume attempts to run everything on local privately controlled hardware. This mitigates some of the issues related to privacy and security when using, for instance, cloud computing, and avoids depending on the service provider to be operational. It can lower the cost of running the system and it also increases fault tolerance as services can continue to operate in case access to the public Internet is lost.

5) **Integrated building control** is supported by Brume for both real-time and supervisory control loops. Real-time control loops can execute on Brume nodes controlling hardware. Supervisory loops which require more processing but are less latency sensitive can run distributed throughout the cluster. However, since all Brume nodes are fully connected, have access to all information within the cluster and mechanisms to coordinate with other nodes, it is possible to create more integrated building control loops which avoid vertical silos.

6) **Efficient application development** as the services provided by Brume are designed specifically for enabling efficient building application development. The programming model allows developers to easily create applications which are secure, automatically scale with the size of the fog, can access online metadata about the entire cluster, receive sensor values in real-time, access historic data and interact with hardware connected to any node.

4. **Brume In a Building**

To give an understanding of what Brume is and how it might work in a building, we provide an example of a potential setup in a building. This could be one of the many buildings in the US which are below 50.000 square feet which today is not equipped with a basic BAS [1]. This example setup is illustrated in figure 1.

The figure shows the outline of a building with a number of different devices some commonly found in a building and new thick sensor nodes with resources similar to common single-board computers running as Brume nodes. These devices are connected via heterogeneous network protocols and together form a secure end-to-end encrypted IPv6 overlay network (red outline) through which all devices can communicate. It is important to note that these devices are not connected to 'the Brume’ like devices might be connected to the cloud; rather they are the Brume, and through coordination and resource sharing these devices provide the four functionalities listed above the figure. Depending on the components running on a node, it can expand the clusters sensing and actuating capabilities, its computational performance, storage capacity and fault tolerance. This flexibility means that Brume’s physical manifestation can be hugely varied and unique for each building, while from a software perspective Brume node coordinate to provide a uniform service oriented environment.

Thick sensor nodes and sink nodes are excellent building blocks for Brume, as they would extend both sensing, processing and storage capability of a Brume installation and allow large amounts of data to be stored and processed locally. While IoT is focused on ultra low-powered sensors capable of running for years on coin batteries and energy harvesting, we imagine ‘thick’ sensor nodes packed with sensing, storage and processing would be more suitable in a building because here electricity outlets are fairly common everywhere. These nodes could also act as sink nodes as known from IoT WSNs. Software PLCs, such as the one

![Figure 1: Example Brume setup in a building](image-url)
shown in the bottom left of the figure, combine the features of a regular PLC with a fully fledged PC architecture. They could act to connect most existing sensor and actuator infrastructure from various bus protocols to Brume. Mobile devices of occupants could aid in extending the clusters occupancy sensing capabilities and computational performance, while providing users with native mobile applications for interacting with the building.

5. System Design and Components

Figure 2 presents a graphical overview of Brume. The left side of the figure show the structure of Brume on a single node, and the right side shows a cluster of connected Brume nodes. Brume consists of twelve components that are grouped in four separate layers, forming a layered architecture. Components at each layer only communicate with components within the same layer or the layer immediately below.

The layered architecture transcends a single computing device, and spans a cluster of connected Brume nodes. These devices can be very heterogeneous in nature, spatially distributed, and might run different subsets of Brume components depending on their capabilities (as illustrated on the right side of figure 2).

In traditional distributed systems individual components are usually instantiated once and might be assigned to different physical nodes. In more complex setups where high scalability or availability is important more instances of selected components might be configured and provisioned upon demand. Brume takes this approach to the extreme and allows each individual component to run simultaneously on any number of physical nodes. Brume is designed such that interfacing with one instance or another is completely transparent, because each component coordinates and aggregates data and functionality from sibling components on other physical nodes. In the next sections we will cover each component in more detail, motivate their purpose in a BAS and elaborate on the challenges of implementation in our prototype system.

5.1. Network layer

While connectivity is paramount for successfully operating a distributed BAS, configuring and operating a building wide network infrastructure is a complex and challenging task. For Brume we wanted a network technology which was easy to configure and operate to make node additions easy. We wanted Brume nodes to be able to operate on existing network infrastructures, thus sharing the same network as building occupants while still keeping all communication secure. We imagine that Brume nodes in a building are likely to be physically located within radio distance of adjacent nodes such that mesh networking can be utilized to improve the robustness of communication links.

To provide secure connectivity the network layer of Brume forms an encrypted IPv6 darknet (an overlay network which can only be accessed through specialized software), using the Cjdns network protocol [20]. Cjdns uses public key cryptography to enable end-to-end encryption, secure messages from tampering, protect against replay attacks and ensure that IPv6 addresses on the network can not be forged because they are derived from a nodes public key. Forging an address would require breaking the cryptographic key system of Cjdns, which is considered extremely hard [21]. Cjdns is sometimes compared to the better well known IPsec network protocol suite, as they have some overlapping functionality in respect to authenticating and encrypting every packet on a network. Cjdns was chosen for its unique features which allow it to run directly on the link-layer and to route packets in ad hoc networks. This allows much of the growing complexity involved in configuring IP based networks to be avoided (e.g. NAT, DHCP, DNS, gateways).

**Seamless connectivity** is performed by integrating all available interfaces and physical links to the network. Multiple links are used to boost connection speeds, while also increase robustness of the connection. Because Cjdns is a link layer protocol it enables Brume nodes to communicate over any available IP connections and also directly over Ethernet or 802.11 links. The Brume prototypes developed in this project use an IEEE 802.11s compatible WiFi module to create a wireless ad hoc mesh network.

**Peer discovery** plays an important role in reconnecting intermittently disconnected nodes, as well as easing expansion of Brume with new nodes. Brume nodes use Cjdns to enable auto-peering with other nodes using link layer beacons. While this enables peers to be discovered, Brume uses the Linux firewall to drop all traffic at the IP layer unless it comes from nodes which have been explicitly allowed to join Brume.

**Mesh routing** is used to deliver on requirements for reliable networking. Cjdns is used to relay messages throughout
the Brume, such that two nodes can be connected through intermediaries as long as some path through the connection graph exists.

5.2. Cluster layer

The cluster layer provides the necessary functionality to overcome the fundamental problem of coordinating nodes in a distributed system. While Brume is designed to be as decentralized as possible, it is necessary to have some form of coordination between nodes in order to perform many of the functions of a BAS. In order to ensure consistency, nodes need to communicate and make consensus, while the CAP theorem and the risk of creating a split brain, makes this task difficult. To find a suitable technology the potential of the Ethereum [22] blockchain was investigated. However, it was unclear how running a local blockchain would affect CPU, RAM and disk usage of the small devices, and we found that data center software for coordinating clusters of nodes offered more features which suited our use case better. We investigated Apache ZooKeeper [23], etcd [24] and Doozer [25] before deciding to use Consul [26] as the main technology for the cluster layer of Brume. Consul has low system requirements, operates decentralized and offers functionality to implement both quorum based strong consistency through the Raft algorithm [27] and weak consistency using a variant of the SWIM gossip protocol [28]. Other features include service discovery, health checking, a strongly consistent replicated key-value store and distributed locking mechanisms, all of which we found to be very useful for implementing a distributed BAS.

Coordination of services in Brume is largely based on the consistent key value store provided by Consul. This enables services to perform leader elections and distributed locking when strong consistency is needed, while information dissemination through gossip is used when weak consistency is acceptable. This enables the independently executing nodes and services to act as a unified entity.

Metadata provides mechanisms for discovering available services within the Brume cluster, and ensures that nodes have an updated registry of other nodes and their services. It disseminates up to date metadata about all nodes through the gossip protocol, allowing metadata to be ‘online’ rather than static.

Monitoring of services and nodes in Brume is performed with various health checks, ensuring that failed services are filtered out when service queries are performed. Health checking is a cornerstone of high availability setups, because a failed node or service has to be detected before corrective measures can be taken. Health information is part of the metadata about nodes, and monitoring of nodes also provides input to the ACL component which might block nodes based on health information.

ACL (Access Control Lists) keeps track of nodes and users who are members of Brume, and configures network security and access to services accordingly. Thus certain key spaces like, for instance, the list of Brume member nodes and their Cjdns public keys are locked down, allowing only administrator users to change it. The ACL component uses this information to configure Cjdns, and firewall rules on the node. To protect against physical tampering, Brume can be configured to disallow selected nodes to rejoin after a health failure has been detected, which could be invoked by designing the casing of a node such that opening it disconnects its power supply. A node which has failed and comes online will only initially trust a selected number of secure nodes and from here source the full list of trusted nodes.

5.3. Service layer

Five separate components together form the service layer, which provides the main services for intelligent building control and applications. Service layer components are made to be horizontally scalable and do coordination throughout all Brume nodes, making them run as a single entity despite being distributed across the entire cluster.

Brume DB is a distributed time-series database designed for Brume. While the performance of a single database server is important, for the case of Brume horizontal scalability is of equal importance. A number of existing time-series databases (e.g., LevelDb, InfluxDB, Riak, Cassandra and OpenTSdb) which offer horizontal scalability were investigated, but failed to meet the requirements of Brume. While the performance of a single database server is important, for the case of Brume horizontal scalability is of equal importance. A number of existing time-series databases (e.g., LevelDb, InfluxDB, Riak, Cassandra and OpenTSdb) which offer horizontal scalability were investigated, but failed to meet the requirements of Brume for one or more of the following reasons:

- Their system requirements were only suitable for data center hardware, and they were not designed to operate on resource constrained devices including common single-board computers.
- They were not suitable for dynamic environments (e.g. nodes with intermitted connectivity or moved nodes due to changes to building layout) as heavy re-indexing and relocation of data was performed every time nodes were added or removed.
- They assumed a topology in which all nodes and connections are homogeneous, which is a wrong assumption in a mesh network of heterogeneous hardware.
- They were not possible to integrate with Brume’s cluster layer, as they would essentially run an independent cluster replicating many of the functionalities from Brume’s cluster layer, and thus wasting resources.
- They were not completely decentralized, and required some kind of special coordinating nodes.

Brume DB was implemented as a fork of the open-source time-series database SiriDb [29], which is programmed in native C and showed good performance while consuming very little system resources. It has a powerful query language for retrieving time-series data and applying various transformations and aggregations in real-time. SiriDb can connect multiple nodes and allows making queries and insertions at any node as were it a single database.
To meet the requirements of Brume, SiriDb was modified and extended to store data close to its origin, while also caching data where it is consumed. This helps to avoid unnecessary data flows across many hops in a mesh setup, which could cause network congestion when the number of nodes increases. Another extension was made to use the gossip-based health checking from the cluster layer, rather than sending heartbeats between all nodes at fixed intervals.

In order to make Brume DB fault tolerant, integration with Brume FS (as explained below) was implemented which allows data to be stored redundantly across Brume. Brume DB uses the components from the cluster layer to automatically configure the database without any user intervention or master nodes, to detect failed instances and automatically reconfigure for uninterrupted operation and to ensure that data will be eventually consistent.

Brume FS is a distributed files system for Brume. While distributed and clustered file systems are not a new concept, we found that existing solutions (e.g., ClusterFS, SeaweedFS, LizardFS) have largely the same issues as described earlier for current time-series databases. Brume FS is a distributed and redundant file system which is implemented by using IPFS (Inter Planetary File System) for content distribution and the functionality from Brume’s cluster layer to coordinate the file-system operations.

IPFS can be compared to a Bittorrent swarm, where connected peers exchange blocks of data. This underlying functionality of IPFS is used for moving data between nodes, because the Bittorrent protocol has been proven very efficient and scalable [30]. Brume FS uses the cluster layer to perform service discovery and automatically configure the Bittorrent swarm. Consul’s distributed key-value store is used to map locations in the file system to IPFS block hashes, which serve as addresses in the IPFS swarm. When adding or updating a file in Brume FS, a redundancy level can be specified, or a default value will be used. The file is first added as a pending transaction, and other nodes are notified to start replicating the data blocks. Once the redundancy level is reached, the file update is committed. This ensures that a node does not update a file and then immediately goes down, making the file unavailable.

Brume FS coordinates the distribution of blocks in the network by considering the required redundancy level of a block and the amount of free storage on different nodes. Furthermore, the underlying IPFS swarm automatically caches blocks where needed, and these are again seeded to other nodes. This ensures that the data is stored in an efficient way in respect to the network topology and that it adapts to the changing data flows as the network evolves.

Brume Functions provide a programming and runtime framework for implementing both real-time building control logic and distributed applications. Brume Functions are inspired by the new advancements in serverless frameworks, also known as FaaS (Function as a Service). FaaS is suitable for Brume because it puts minimal responsibility on the developer in regards to computing infrastructure, enables code to run anywhere and allows the developer to easily develop for horizontal scalability. A Python SDK was developed to ease the development of functions which utilize the distributed resources of Brume.

To create a Brume function developers need to provide a python executable, an optional list of required python packages which are automatically installed in the virtual environment where the function is executed, and a set of configuration options specifying the trigger conditions for the function. These conditions can specify: 1) if the function should execute on a schedule or be triggered by arrival of data from any number of data streams; 2) if a single or more instances of the function can run simultaneously; and 3) if a filter should limit what physical nodes can execute the function. This filter can be based on any kind of available metadata about nodes, like for instance their available memory, processor speed or running services.

Brume Worker provides a computational framework, which can distribute heavy tasks across the heterogeneous computing resources in Brume. It provides a grid-computing inspired approach to distributing computations, which was chosen over traditional solutions like MPI (Message Passing Interface) [31], because it works better with heterogeneous nodes and is more fault tolerant in a dynamic environment where nodes might connect and disconnect intermittently.

Brume Worker is basically an extension of Brume Functions, and can be used through the Python SDK. It allows a function to create so-called work packages which are special kinds of functions that are only executed once as soon as an idle node can handle it. The filtering mechanism of Brume Functions can also be applied to work packages. Any number of worker packages can be created, and a callback function can be specified which executes once all the work packages have been executed. A callback function can create new worker packages and again provide a new callback, which enables a flexible system that allows many different kinds of distributed processing tasks.

Hardware Interaction provides interfaces to resource connected with individual Brume nodes, enabling interaction with building equipment following the driver concept established by other BOS platforms. We found that collector daemons originally designed to collect metrics from data center nodes, could be adapted to our use case of continuously collecting and reporting sensor values. We investigated collectd and statsd, before deciding to implement hardware interaction by extending the modular plugin-driven agent Telegraf with a number of custom modules. While Telegraf is geared to work with InfluxDb, we chose to use our custom BrumeDB database because the cluster functionality of InfluxDb is close sourced, requires a license and according to the specification is too demanding for hardware constrained devices.

Telegraf extensions to interface with new hardware can be created in GO, and we implemented functionality to automatically expose set-points and metadata for the connected hardware to Brume, making it accessible anywhere. This component continuously reads and archives sensor values to Brume DB as well as pushes them to subscribed listeners through the ZeroMQ library.

When applications or services want to interface with
hardware they must do it through the set-points which are exposed in the strongly consistent key-value store found in the cluster layer. When set-points are updated the hardware interaction component immediately invokes the responsible extension modules, which can translate the set-points to actuator commands. This design ensures that commands are persistent and can be executed in atomic transactions.

5.4. Application layer

Applications run in the native environment of devices including desktops, smartphones, smart appliances or servers. A device only need to have the network layer components running on the device for an application to have access to services running anywhere in the cluster.

The application layer contains no Brume components itself, but represents a layer of applications external to Brume, which might be running on the same physical device as other Brume components. These applications can interface with Brume components in order to perform their operations for instance a data visualization application for desktops or a mobile app for adjusting room temperature. The interface is via endpoints of the service layer components. The applications can make a service enquiry either via the service layer or a DNS lookup which makes it opaque which Brume node is serving the request.

6. Prototype

A prototype was developed consisting of five Brume nodes running our implementation of all the layers and components proposed in Section 5. The nodes were setup and connected in a way which simulates a small realistic building environment. Several applications were prototyped including real-time control logic and supervisory control with this prototype.

6.1. Hardware and Configuration

The prototype nodes were based on a Raspberry Pi 3 Model B mini computer. The onboard WiFi module of Raspberry Pi does not support the IEEE 802.11s standard for wireless mesh networking, thus each node was extended with a TP-LINK TL-WN722N Wireless USB adapter.

Three of the five nodes were extended with a GrovePi board, which connects an I2C (Inter-Integrated Circuit) bus to the Raspberry Pi's GPIO (General-purpose input/output) interface. The GrovePi board comes with a number of different plug-n-play sensors and actuators, of which the three prototype nodes had the following components attached: sound volume sensor, light intensity sensor, ultrasonic range sensor, combined temperature and humidity sensor, sound buzzer, a red and a blue LED and a button which can be either pressed or not pressed. Each node runs Raspbian Jessie Lite, which is a Raspberry Pi optimized version of the Linux OS, Debian 8 (Jessie).

The three nodes which have GrovePi extension boards, simulate three software PLCs connected to building equipment over three isolated I2C bus networks. The connected hardware enumerated in previous section simulates various building equipment like sensors (sound, light, temperature and humidity), lighting systems (LED diodes), alarms (sound buzzer) and control panels for equipment (button).

The other two prototype nodes simulate sink nodes which collect data from other IoT devices nearby. For instance, through BLE or ZigBee. However, in the prototype setup these nodes just collect CPU metrics.

6.2. Prototype Applications

In order to evaluate the feasibility of Brume as a platform for building automation, a number of applications were developed and deployed to the prototype Brume.

6.2.1. Real-Time Control Logic. Real-time control of the simulated building equipment was implemented using the Brume functions component. The following control logic assumes that the three nodes which have Grove Pi hardware attached, are controlling building equipment in three identical rooms.

- **Automated lights** which turn on the lights (blue LED) if motion is detected from the motion sensor, and the room is too dark based on readings from the light intensity sensor. The lights are turned off if no motion has been detected for 10 seconds.
- **Room locking** which make it possible to lock and unlock a room using a control panel (the button). When a room is locked, the automated lights system is disabled, and lights are turned off. The red LED is turned on to indicate that the room is locked.
- **Alarm** which is active if a room is locked. The alarm sounds in the entire building (all three sound buzzers) if somebody attempts to break into the locked room (the ultrasonic range sensor reports a value below a certain threshold). An authorized user needs to disable the alarm through the prototyped administrator application.

6.2.2. Supervisory Control. A case is designed to showcase the platforms ability to perform supervisory control based on predictions of a Neural Network (NN). The case revolves around using collected sensor data to predict indoor environmental properties 24 hours ahead. Specifically the time of day along with light, CO2 and temperature data was used as input to the network, while the output 24 hours ahead is used as the supervisory signal.

The NN was trained using The Island Model Genetic Algorithm. A Brume function triggers at certain time intervals to retrain the network, by first initializing a random population of genomes encoding synapse weights for the NN. It then uses the Brume Worker component to distribute the evaluation of the genomes. Each work package is an island in the genetic algorithm, and evolves the genomes
for a configurable number of generations. When all islands have reached the specified generation (e.g. all work packages complete), a callback Brume function is executed which performs the migration step of the algorithm, and then creates a new set of work packages to continue the evolution of subpopulations. This continues for a specified number of epochs until the final callback stores the results in Brume DB. The resulting NN can then be used to predict indoor environmental properties and schedule supervisory control.

6.2.3. Cluster administration. A web application was developed and installed on each node to ease installation and management of the Brume nodes. The goal for this application is to show that Brume from a user perspective is simple to operate, and that the distributed physical setup does not increase complexity for users.

When a new node is placed in a building it will create a Wi-Fi access point and automatically start searching for an existing Brume to join. Users can connect to the access point in order to initialize a new Brume. This step requires that users enter a username, a master token that must later be provided when performing administrative tasks and a Cjdns public key to associate with this user. A name can also be specified for the new Brume as well as a name for the specific node. The node will then initialize a Brume which can only be accessed by the initial user through Cjdns. Cjdns requires a few steps to configure, and once configured users can access Brume simply by providing their user name.

If users wish to expand Brume with new nodes, it is done by allowing them to join from within the Brume. Figure 3 shows how the link layer based peer discovery described in section 5.1 automatically discovers new nodes, which can be allowed to join Brume. Users must provide a name for the new node as well as a trust level which was described in section 5.2. The public key of the new node is automatically prefiled while the user must provide the master token required for administrative tasks. The new uninitialized node will also start making Cjdns link layer beacons on all connected interfaces to make its presence visible to existing Brume nodes.

Once users are connected to Brume via any node, they can access the web application of all nodes seamlessly. Clicking on a specific link might transfer the user to a different node, however the user might not even notice this. Figure 4 shows how the web application presents the hardware available on all nodes, based on metadata and set-points found in the cluster layer. Figure 5 shows the user interface for creating a Brume function. It simply requires users to upload a .zip file containing the python executables and configuring the trigger conditions directly on the website. The function is then ready to execute on any node, when the trigger conditions are met. The prototype web application provides more functionality which is not described here due to space restrictions. This includes interfaces to configure node metadata, hardware and network setting of nodes and for querying and visualizing time series data in Brume DB using SiriDB-http [29].

7. Evaluation

This section will present the results of several micro benchmarks that test the performance and scalability of Brume’s different components. The use of micro benchmarks enables us to provide results for the individual system design principles behind Brume. Furthermore, we will present an experimentally verified assessment of fault tolerance and an evaluation of the prototype applications.

7.1. Performance and Scalability

Network Layer scalability depends hugely on the available network hardware and setup. However, Brume envisions that configuration free mesh networking between nodes is the ultimate goal, and therefore we evaluate how Brume performs and scales in this setup. Cjdns is relatively new and while the vision is to operate a network at global scale, it is yet unknown how well its source routed mesh protocol scales to a global scale [32].

Table 1 and 2 show the average latency of 30 ping requests and transfer rates of a 100MB randomly generated file, across different number of hops through our prototype nodes. The nodes were placed in close proximity so some radio interface is to be expected with increasing traffic. We can observe that ping times increase with less than a millisecond per hop for both types of network hardware. In terms of transfer rates with Ethernet there is no impact with several hops where as with 802.11s we see a larger drop. We attribute this drop for 802.11s to increasing radio interference.

Another test was conducted to see how the encrypted overlay network of Cjdns would impact network performance. Table 3 summarizes the results, and shows how the raw transfer speed is impacted depending on the link type. Interestingly Cjdns performs better than the standard routing protocol of 802.11s (Hybrid Wireless Mesh Protocol (HWMP)) which in general support the choice of Cjdns.

Cluster layer performance and scalability mainly depends on how well Consul’s gossip protocol scales. A reason why Consul was chosen over other solutions is because its gossip protocol adapts to the network by considering RTT.

| TABLE 3: Impact of Cjdns on network performance |
|-----------------|-----------------|-----------------|
|                | Without Cjdns  | With Cjdns      |
| IPv4 network   | 11.8 MB/s       | 5.8 MB/s        |
| Raw Ethernet   | N/A             | 6.7 MB/s        |
| 802.11s        | 5.4 MB/s        | 5.7 MB/s        |


between nodes [33]. This consideration is important in a mesh setup because gossip between distant nodes is much more expensive, than nearby nodes. Based on a simulator of the underlying gossip protocol [33], we can estimate that a Brume containing 1000 nodes in a uniformly distributed mesh network where each node is connected directly to 4 other nodes, can converge (all nodes in the cluster have received a message sent from one node) in less than 1 second. This scenario would cover a medium sized office building with several nodes per room. Based on the simulation this requires connection links between nodes to be at least 437.5 kbps, which the previous network benchmark demonstrated was achievable even under severe radio interference for up to 2 hops. If health checking had been implemented with traditional heart beats between nodes, every node would have to send or receive a heartbeat from the 999 other nodes every second, or a central master node would have to be used. Both options would quickly drain the available bandwidth of a mesh network, because the amount of messages would be very high and they would need to travel many hops.

**Service Layer** scalability depends on Brume’s combined performance. To evaluate the scalability we developed a distributed video transcoding program using the Brume functions and Brume worker components. This benchmark simulates many real scenarios where a task involving large amounts of data needs to be processed, and where the work can be split into independent sub processes. We then benchmarked the program by transcoding a 2.5 min long 4k video file of 1.1 GB, to a ∼15MB 420p video file. For comparison, the same benchmark has been run on a commodity laptop. The laptop has the video file located on the disk, whereas in Brume the file is only located on one node which triggers the Brume function and then has to be shared over the network in order to distribute the task.

Table 4 shows a comparison of the hardware on the tested laptop and a single Raspberry Pi 3. Table 5 is a comparison using a processing benchmark tool called SysBench, which tests the raw CPU power in terms of number crunching. The laptop is ∼9 times faster in terms of raw numbers.

<table>
<thead>
<tr>
<th>TABLE 4: Hardware comparison of Raspberry Pi 3 and commodity Laptop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Threads per core</td>
</tr>
<tr>
<td>Core(s) per socket</td>
</tr>
<tr>
<td>Model name</td>
</tr>
<tr>
<td>CPU max MHz</td>
</tr>
<tr>
<td>CPU min MHz</td>
</tr>
<tr>
<td>BogoMIPS</td>
</tr>
</tbody>
</table>

The Brume function version of the transcoder works by creating a number of worker packages and giving each a specific interval to transcode. Once a worker is complete it will store the transcoded result in Brume FS, and when all workers complete the callback function will then concatenate all parts into a single file again. The measured time is from the first invocation of the Brume function, and until it has successfully created the output file.

Figure 6 shows the result from the benchmark, where the graph shows the computation time for the transcoding in relation to the number of nodes in Brume. The laptop is a horizontal line, as the transcoding time is constant in regards to the x-axis. Adding a node to Brume reduces the computation time by ∼50%, all the way up to the five nodes in our prototype. From this, we can deduce that it follows a power function, which is represented by the blue line in the graph. With five nodes, the laptop is only ∼30 seconds faster, and following our trend-line in theory ∼10 Brume nodes should be twice as fast as the laptop. However, distributing the video file from a single node might kick in at some point as a bottleneck.

**Brume FS** is tested for horizontal scalability by a test where one node adds a 100MB file with random content to
Brume FS. The system then measures how long it takes to synchronize with a number of other nodes. Table 6 shows the results of this test. From the results we can observe that all nodes finished retrieving the file almost at the same time, which is why only one number is shown. We see that increasing the number of nodes by a factor 2 only increases the time it takes to sync all nodes with a factor $\sim 1.3$. This is due to the swarm architecture of the underlying BitTorrent protocol used to distribute the data. When an initial set of chunks have been distributed between peers, those peers start exchanging chunks without waiting for the first node to finish downloading the entire file from the seed. This makes Brume FS highly scalable and allows a large replication factor without severely impacting performance.

**Brume DB** shows linear scalability, if insertions to a given time-series is localized to a single node and insertions are distributed evenly across the Brume. This will usually be the case when Brume nodes are installed as sink nodes or thick sensor nodes, in which case the speed with which data can be queried and inserted depends on the speed with which Brume FS can replicate data.

### 7.2. Fault tolerance

Each component of Brume was assessed in order to evaluate the fault tolerance of the system in various conditions. Our assessments were verified experimentally using our 5 node prototype Brume, where failures were simulated by cutting off power to nodes.

**Network Layer** fault tolerance is very dependent on the configured network infrastructure and the spatial topology of the Brume nodes. All services might be designed to handle multiple failed nodes. However, a single failed node in a weakly connected mesh topology can still leave Brume in a state with two disconnected components, resulting in an unhealthy state for either one or both components. This risk can, for instance, be mitigated by connecting a subset of the nodes through other channels like existing building wide networks (WiFi or LAN). Thus, these nodes are able to create a route between otherwise isolated subnetworks which are spatially too distant to communicate directly via mesh.

**Cluster layer** fault tolerance depend on Consuls RAFT protocol. While all Brume nodes participate in the cluster gossip to disseminate information in a weakly consistent way, only a few selected nodes participate in Consuls RAFT protocol which handles strong consistency. This is because each participant in the RAFT protocol significantly increases the network communication needed to achieve consensus. While most redundant systems can handle gradual failure effectively by automatically provisioning new nodes, it is difficult to automate this process for a RAFT quorum without introducing scenarios under which a split brain can occur. Thus, if a majority of the nodes participating in the RAFT protocol fail, Consul, and thereby Brume, goes into an unhealthy state where most of the services will stop working. However, an operator can freely select which nodes participate in the RAFT protocol using our web application for Brume, where it is also possible to manually handle gradual failures of RAFT participating nodes. For instance, choosing 5 nodes to participate in the RAFT protocol, allows two of these to fail without any significant effect on the cluster layer services.

**Brume FS** can handle gradual failures as long as the remaining nodes have storage capacity to hold all the data and there is enough time between node failures to make sure the data is replicated to other nodes. It can also handle sudden failures, as long as the failing nodes are not the ones containing all the replicas of a given data block. In this case the chosen replication factor determines how many nodes can fail before some data is lost, while other data which still has replicas will be accessible.

**Brume DB** relies on Brume FS for replication, thus behaves similarly in case of failures. While Brume FS can handle gradual failures down to a single remaining node, Brume DB cannot handle if more than 50% of nodes fail. This is related to our current implementation where a single Brume DB instance can only serve backed up data for one failed node at a time.

**Brume Functions and Brume Workers** can handle both gradual and sudden failures, as long as the executable python files and the associated configuration data remains available in Brume FS. In the prototype, if a node fails executing a function or worker package another node will take over within a few seconds.

**Hardware Interaction** can not handle gradual or sudden failures, because if a node controlling hardware fails that hardware becomes unavailable to Brume. However, this component can tolerate that the cluster layer becomes unhealthy as it can continue to read and buffer sensor values and execute local control loops which might be imple-
TABLE 7: Building Control

<table>
<thead>
<tr>
<th>Brume Function</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automated lights</td>
<td>37</td>
</tr>
<tr>
<td>Alarm</td>
<td>18</td>
</tr>
<tr>
<td>Room locking</td>
<td>27</td>
</tr>
</tbody>
</table>

TABLE 8: Neural Network Performance

<table>
<thead>
<tr>
<th>Device</th>
<th>Lines of code</th>
<th>1000 gen.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laptop</td>
<td>116</td>
<td>32 min and 18 sec</td>
</tr>
<tr>
<td>5x Brume Nodes</td>
<td>167</td>
<td>33 min and 27 sec</td>
</tr>
</tbody>
</table>

7.3. Prototype Applications

Implementing the real-time control logic described in section 6.2.1 required three Brume Functions to be created. The control logic responded in near real-time, with the bottleneck being the GrovePi hardware and not the Brume framework itself. The lines of Python code in each function can be seen in table 7. Each function handles a different subsystem (lights, alarm and locking) for all three simulated rooms, and is implemented in a generic way allowing it to handle the control of any number of similar rooms. However, we do rely on inexplicit knowledge about our chosen format for labeling sensors and actuators. While, the system provides a lot of metadata related to the state of Brume, a proper framework for handling building related metadata like for instance Brick [34] would be a great improvement.

For the supervisory control setup we have evaluated the NN described in section 6.2.2. While the resulting NN was able to distinguish night and day the average error was not overly convincing, e.g., when predicting light levels 24 hours ahead was around 200 lux. As the purpose of this evaluation was to demonstrate that Brume can execute the NN we did not focus on decreasing this error. Table 8 compares the time required to train the NN on Brume with 5 nodes, with a commodity laptop as also used for earlier experiments. The table shows that the five prototype nodes deliver on par computational time with the more powerful laptop, underlining our previous findings from the video transcoding benchmark. The table also compares the two implementations of the NN, showing that the version using Brume Functions only requires 51 extra lines of python code.

8. Discussion

With Brume we have proposed and designed a system that fulfill our six listed design goals. In this section we reflect on the evaluation results in regards to the design goals for Brume. Table 9 compares how Brume and existing centralized BOSs perform in terms of the six design goals. The table highlights the advantages of Brume in terms of scalability, fault tolerance, security and third party independence. Given the horizontal scalability one could argue that the benefits of Brume increase with the building size. Furthermore, the evaluation results provide evidence for that Brume supports integrated building control and efficient application development as also supported by existing centralized BOSs. In terms of distribution overhead the results demonstrate that Brume on multiple distributed nodes with low overhead could collaborate to handle tasks as efficient as a centralized node. In terms of maintenance a centralized BOS only has to be maintained on one node but connectivity to the many forms of building instrumentations has to be maintained. For Brume via the shown management interface multiple nodes need to be maintained but with directly connected building instrumentation which should allow for minimal maintenance with Brume.

In our presented evaluation we cover a set of prototype applications and evaluation setups to match common buildings without a BAS. In future work it would be relevant to perform a larger, more realistic and long-term deployment to further evaluate the benefits of Brume. In particular to document the performance overhead of distribution and maintenance needed over time. This also includes programming more complex applications and verifying the behavior of the system after unforeseen composition of new devices. It could also as part of such an evaluation be relevant to consider some extreme scenarios, e.g., earthquakes or other disasters of similar magnitude for stress testing the fault tolerance. An aspect that we did not cover in this work is metadata for building data and privacy protection of data across nodes. It would interesting to extend Brume in these regards building on recent work, such as, Brick [34] and PAD [35]. In regards to metadata it would also be relevant to enable Brume nodes to automatically spatially position themselves within the building.

9. Conclusion

In this paper we have addressed the physically centralized nature of current building automation systems which increases complexity and results in limited scalability and fault tolerance. We have presented Brume, a BOS which logically provides the same kind of service oriented architecture as existing BOSs, while physically running on the same fabric as IoT; large numbers of small and heterogeneous devices. This enables Brume to be highly horizontally scalable and fault tolerant through means of redundancy, while also being secure and simple to operate. To verify these expected benefits and the applicability of the platform for building automation, we presented a prototype deployed in a real building setup matching a small building without a BAS. A set of micro benchmarks were run against the prototype to evaluate various aspects of the design, and two prototype applications showed the platforms ability to support real time control of building equipment and supervisory control based on predictions from a neural network. By presenting Brume we hope to inspire the development of a new generation of building automation systems.
TABLE 9: Comparison of results for system design goals of Brume with existing BOSs.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized BOS</td>
<td>With server size</td>
<td>Poor</td>
<td>Special configuration per link</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Single point but connectivity to the many forms of building instrumentations has to be maintained.</td>
<td></td>
</tr>
<tr>
<td>(e.g., sMAP [8], BOSS [3] and BuildingDepot [9])</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Brume</td>
<td>With building size</td>
<td>By design</td>
<td>By design</td>
<td>Yes</td>
<td>Yes</td>
<td>Low overhead as documented by experiments</td>
<td>Maintenance of nodes with directly connected building instrumentations.</td>
<td></td>
</tr>
</tbody>
</table>

Acknowledgment

This work is supported by the Innovation Fund Denmark for the project COORDICY (4106-00003B).

References

[26] “Consul: https://www.consul.io/.”