

# Automated Testing in the Internet of Things

This article presents a novel approach to testing distributed systems. Our automated test environment (ATE) is created to validate the BACnet/IT building automation protocol and is easily adaptable to other domains. During development of the new BACnet/IT reference implementation, we had to face several testing challenges. Based on that, we derived requirements for the ATE. The result is a flexible and lightweight test environment, which consists of only a few interacting components. Our ATE is able to simulate real-life situations like a power outage or a replacement of a BACnet/IT device. Further, it allows manipulating the behavior of BACnet/IT components during runtime. With such a test environment it is possible to automate tests in a straightforward and efficient way.

Thomas Dobler, Artem Khatchatourov, Christoph Stamm, Wolfgang Weck | wolfgang.weck@fhnw.ch

We use more and more digital devices in everyday life. When you turn on the lights, a traditional switch actually closes an electrical circuit, but in many modern buildings the switch sends a digital message over a communication network to the light, or even to several lights in the same room. The advantages are plenty. There is not just one light switch next to the door, but lights can be turned on and off or even dimmed from several places. In our lecture rooms, for instance, one such place is the speaker's desk. Also, lights might be switched by automatisms, for instance, turned off whenever sensors detect that the room is empty or that there is enough daylight coming in through the windows.

Switching lights is just one example of the upcoming Internet of Things (IoT). A rapidly increasing number of digital devices may increase comfort and use energy more efficiently through automatization and mutual interaction. With the IoT, many relatively simple and small devices will exchange short messages with each other, partially with real time constraints. Using Internet technology for data exchange reduces cost by avoiding dedicated cables and by sharing software infrastructure, such as name and addressing services and security mechanisms.

From an engineering point of view there is an important paradigm shift included. IoT takes us from system architectures with a central service embedding all intelligence and being contacted by client devices to fully interconnected networks where every device can contact any other device. A system's complexity is not embedded (and encapsulated) in a central node anymore, but spread across the network of a large number of interacting devices, each of which by itself can be quite simple, though.

As engineers we want to have and to provide evidence that our constructions meet their requirements. Next to systematic (partially formal-

ized) construction methods, an important tool for this is systematic testing of new or modified machinery. For single-node computer systems there are established test methods and tools. Programmers deploy unit tests to their code. When software systems are built from source code automated tests are run, and so forth. Servers can be tested through automatically simulated clients and vice versa. However, these approaches only poorly cover situations, when important system properties rely on the cooperative interaction of hundreds or thousands of devices and the connecting infrastructure.

In this paper we describe our automated testing environment for IoT in the application domain of building automation. This testing environment is one contribution of our institute in a joint CTI project<sup>1</sup> with FHNW's Institute of Automation and Siemens Switzerland, Building Technologies Division. Our part in the project is reviewing, prototyping, and evaluating the draft standard of the new internet-based instance of the BACnet protocol: BACnet/IT [BAC16]. In order to be able to evaluate our implementation of the communication stack and with it the new BACnet/IT draft standard, the need for an automatic tool for testing has arisen.

First of all we introduce building automation and BACnet. Then we look at two exemplary application scenarios, raising specific testing demands. The two scenarios shall illustrate requirements to testing tools for BACnet/IT specifically and distributed IoT systems in general. In practice, there are many more requirements and corresponding test cases that can be served with the same set of tools. Finally, we present the elements of our ATE.

## Building Automation

Building automation is the (centralized) control of a building's heating, air conditioning, lighting

<sup>1</sup> CTI Project: Convergence of Building Automation and IT World, KTI-Nr. 16841.1 PFES-ES

and other systems through a building management system.

Building automation of the future requires flexible communication solutions. Due to the development of costs and the spread of Internet technologies, standard IT solutions are increasingly being used. IT infrastructures and services are currently undergoing a process of adaptation to the new requirements of IoT by providing suitable protocols for the integration of devices (CoAP, RPL, 6LoWPAN). Building automation, which uses IT infrastructure, is also subject to this adaptation process.

### BACnet/IT

BACnet is a well-established standard among building automation manufacturers. It has been originally defined for proprietarily wired infrastructure to connect sensors and actors in a building [BAC]. With the ubiquitous Internet of today, migrating BACnet onto standard Internet mechanisms is advantageous.

BACnet/IP allows communication via IP-based networks, but uses IP only as a data link and often communicates with IP broadcasts. Based on the data link, BACnet/IP uses its own BACnet-specific protocols on upper layers. This leads to massive acceptance problems in the IT world and prevents easy interoperability and integration into other domains. These problems have been recognized by the standardization committee and led to a new specification.

The new BACnet/IT draft standard describes how to implement the established application layer based on standard Internet protocols such as HTTPS, TLS, WebSocket, and standard Internet services, such as DNS and DHCP providing a basic communication layer for BACnet messages [BAC16].

With BACnet/IT, building automation no longer runs its own communication infrastructure, but becomes a guest among others on a network with standard Internet technologies. This cuts down operation cost and can even raise reliability if part of the saved budget is used to operate the common standard infrastructure with extra redundancy. As a trade-off, building automation must get along with configurations and restrictions set up by the network operators. Consider for example a bank or another company with high security standards as a building's tenant. Following today's best practices, the network will be configured in zones shielded against each other by firewalls, possibly using network access control, and so forth. Assignment of network addresses and device names may have to follow specific rules. Thus, the building automation devices need to acquire essential information from responsible IT services under more difficult conditions than in a proprietary network.

So, one of the specific challenges is that BACnet/IT devices must be able to cope with all kinds of restrictions a network operator may put on them. At the same time they must not rely on high service availability, because there may be no professional network operation at all. The latter is the case with smaller companies, when IT networking is neither business critical nor part of the core competence. The BACnet/IT draft standard tries to cover all relevant situations within this broad spectrum of possibilities. It defines how devices react to specific situations.

### Example 1: Speed Test

In this and in the next section we describe two exemplary application scenarios, raising specific testing demands. Both scenarios shall illustrate requirements to testing tools for BACnet/IT specifically and distributed IoT systems in general.

Some IoT applications have significant real-time requirements. In building automation these are, for instance, those involving human observable reaction to human activity or alarm transmissions. Consider again the introductory light switch example: An actor triggers a light switch and the light in the room turns on. The light bulb (or more precisely the whole system) needs to react fast enough, so that humans don't experience a delay, not even in a whole corridor with lots of individual light bulbs. Hence, there are tight time constraints specified with such scenarios. Ensuring compliance with these requirements calls for measuring time under varying circumstances, e.g. different network loads.

Figure 1 illustrates this test case. There is a flow of messages between two BACnet/IT controllers. One controller is connected to a light switch and the other to a light source. The two delta mark time spans we want to measure.

Measurements must be repeated under different conditions with varying network load and bandwidth or in different topologies. This allows comparing results to determine how network conditions affect the performance. Having to run the same tests many times raises the need to program

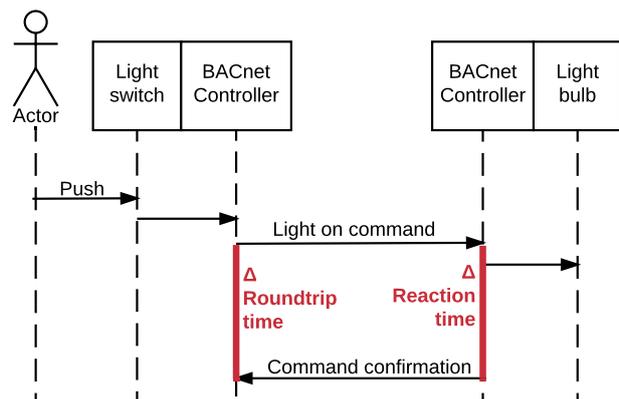


Figure 1: Measuring reaction and roundtrip time in a light switching scenario

the scenario to be run and measured so that it can be executed automatically over and over again. Further, the various environments in which the scenario shall be run must be set up and configured automatically. Without such support, broad band testing would simply be too expensive.

### Example 2: Power Failure

Rarely occurring scenarios as a power outage, for instance, may have high impact, ranging from considerable cost to life threatening. Hence, it is especially important to simulate such situations during testing rather than waiting for them to occur in a live setting. As an example, you may consider a temporary power outage in a building. Some devices of the building automation system will stop to operate, while others will continue to work, based on resilient power supply, such as a local battery. Further, the network infrastructure may be inhibited so that communication channels between devices close down.

It is up to the application programs being run by the individual – temporarily disconnected – devices to cope with such a situation and to prevent major disaster. This, however, is not the topic we are concerned with in this example. We are interested in what happens, when power comes back again and both the network and some temporarily powerless devices start to recover. The problem then is, that many devices will restart their communication at the same moment, synchronized by electricity becoming available again simultaneously everywhere.

In such moments, specific network services become bottlenecks, because they receive requests from each device trying to integrate itself into the system. Consider for instance, name services, responsible for mapping device and service names to actual network addresses. The BACnet standard requires devices (re-)entering a network to announce their availability together with the application domain services they offer. This involves sending a registration message to a specific BACnet directory server (BDS), which cooperates with a standard domain name services (DNS).

Usually, devices are integrated into and removed from the system one by one, so that the BDS can easily handle these registrations. Having many devices synchronized through simultaneous power up will put the BDS and the network under unusual stress. Figure 2 illustrates this specific situation.

It is fairly easy to imagine similar stress situations with other IoT applications also, just because of the sheer number of devices interacting. Forced synchronization, can have various causes, e.g. a fire alarm that triggers a whole number of devices like sprinklers and safety lighting.

Of course, the BACnet/IT draft standard has foreseen such situations and prescribes count-

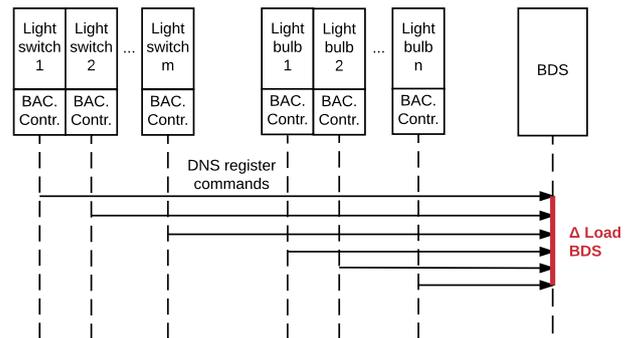


Figure 2: BDS load after a power outage

er measures, such as deferring the registration messages for a randomly selected time span. Our protocol stack implements this. Still, engineering standards require not only to verify such measures but also to quantify the load on bottleneck units as well as the duration until the system recovers to normal operation. On top of this, it is of interest, how many registration requests are dropped by the BDS due to overload.

To run tests that can produce relevant data requires many (such as hundreds) devices to simultaneously send out a message. Connecting a hundred physical devices to a single power socket and power them up together might be possible but would be a costly and inflexible solution. Virtualization is simpler and cheaper. Simulating the synchronized power up situation requires automatically triggering many such virtual devices to send out specific messages at the same time. This process must be programmed (scripted) and invoked during the test. In addition, a stored test procedure will also lead to reproducible test results.

### Requirements to a Test Environment

The two examples above illustrate a set of requirements to an automated test environment for IoT systems in general and our specific BACnet case especially:

- *Devices must be manipulated:* Many test scenarios require some degree of invasive action on the devices forming the system under test. Some scenarios involve devices entering or leaving a system, such as described in the power failure context above, so they must be deployed and run with specific configurations or halted. An actual application must be mocked to initiate message transmissions. To measure time intervals, respective instrumentation must be injected.
- *Tests must be controlled from a single workplace:* Testing an IoT system involves operating several devices in coordination. A small number of devices could be put together in a rack or on a table and be controlled physically. This does not work anymore when many devices are

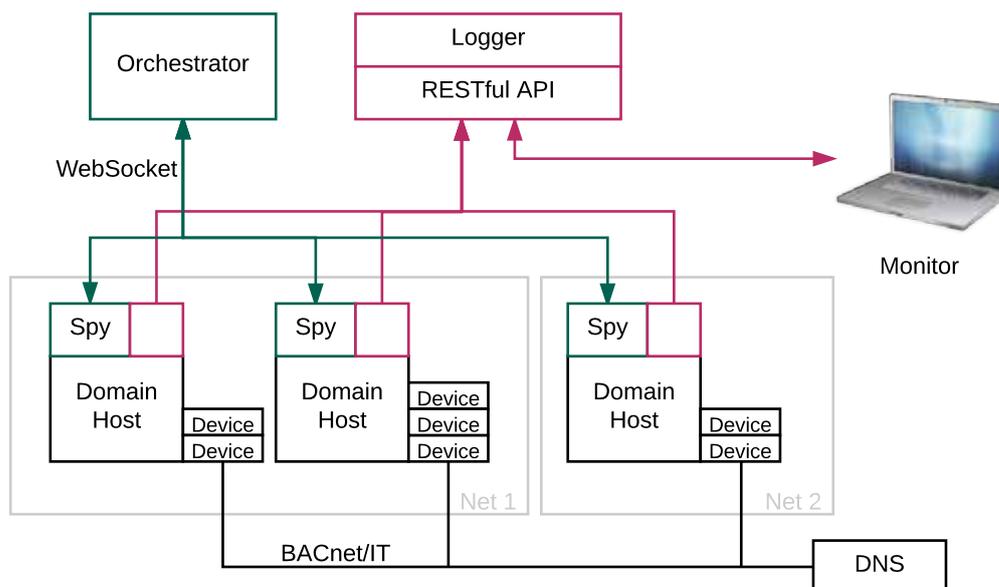


Figure 3: Overview of the components of our Automated Test Environment (ATE)

involved. Some tests and measurements, such as load testing, require coordinated access to a very large number of devices. This should be controlled from a single controlling workplace, managing all devices on the network including nodes in the infrastructure.

- *IoT systems under test must be scalable:* Depending on the application area, IoT systems vary in size. Even building automation systems may range from a few controllers to large amounts of devices. Depending on what to test or to measure, infrastructure from a private home LAN to an enterprise network with dedicated DNS and an aggressive firewall setup must be set up. The testing environment needs to scale from a few – maybe even physical devices – to large amounts of virtual entities.
- *Reproducible tests must be programmed:* There are two reasons, why test procedures must be repeated. First, the same tests may have to be run with varying parameters, such as a different number of devices, different network capacity, etc. Second, tests are not just a one-time shot to prove an implementation to be correct, but they are used as regression tests during further development and change. If for each test a lot of steps have to be repeated manually, the process becomes cumbersome and error-prone. Hence, test procedures should be written as automatically executable programs.
- *System data of devices must be accessible:* As the examples show, there is a need to collect various data on individual devices during testing. For instance, testing with the power outage scenario, we would like to measure CPU load, memory usage, number of open connections, and so on. This requires accessibility of the respective system data. For the first example from above, we need to measure time be-

tween specific network events. These events can be protocolled in a log file, which can later be pulled from the device to the central workplace for evaluation.

#### Related Work

Surprisingly, there are only a few frameworks available for testing distributed networks:

- *Java Device Test Suite [JDTS]:* It is designed for testing embedded mobile devices. It is rather a monitor for an embedded device on the network than a framework for distributed testing of these devices.
- *TETWorks [TET]:* It is designed for distributed testing and has support for different languages and platforms. Unfortunately, it does not allow reading of system parameters, which is imperative for our needs. However, the software is well documented and its approach is a good base for an extended tool.

None of these tools suits our needs, so we decided to implement our own *Automated Test Environment (ATE)*. It is designed specifically for BACnet/IT, but our approach can be used in any building automation domain. We decided to base our environment on the approach of TETWorks. So, if you are familiar with TETWorks, then you will recognize some of its components in our new test environment.

#### Automated Test Environment

Our ATE depicted in Figure 3 allows us to deploy, start and control virtual devices in the cloud as well as in a laboratory and it simplifies logging and visualization of logging data. It consists of five major components: An *Orchestrator* to manage operation of BACnet/IT devices on the network, a *Logger* to collect status and event updates from these devices visualized by a *Monitor*, and a

network node including the two other major components: *Domain Host* and *Spy*.

A network node might be a virtual or physical machine containing one or more Domain Hosts (e.g BACnet/IT hosts). Each of these Domain Hosts can contain one or more BACnet/IT devices.

Virtual BACnet/IT devices usually do not produce traffic on the network because they are not programmed to play the role of a real device. Thus, communication must be invoked manually by the Orchestrator through the Spy, which serves as an intermediate interface. In the following sections we describe each of these components in more detail.

The Orchestrator plays an important role in simplifying automated testing in an IoT environment. It has been designed to fulfill some of the requirements listed above: running and controlling reproducible tests from a single workplace. With dozens of network nodes it is cumbersome to log in individually and to deploy devices manually. This is a common problem in cloud administration and there are many solutions that solve this problem for general use. However, our Orchestrator is more powerful than common cloud administration tools, because it is able to access Domain Hosts even after their deployment.

Some tests require coordinated teamwork from up to hundred BACnet/IT devices on the network. This calls for automated test procedures, which force devices to exchange messages in a fixed sequence. A test written as a procedure can be uniquely identified for documentation purposes and used in other network environments with different hosts. Doing so will produce comparable results, which can be used to compare setups and implementations.

### Orchestrator

The *Orchestrator* is a centralized tool, which manages physical or virtual devices on the network. It controls these devices by sending commands of two different types over Websocket connections:

- *Maintenance Commands* trigger operations such as starting a *tcpdump* process or deploying a new Domain Host with given configuration. These commands are run by Spies and do not concern any BACnet/IT communication.
- *BACnet Commands* are directed towards the Domain Host. These commands trigger BACnet/IT devices to send messages to other physical or virtual BACnet/IT devices in the same domain.

The Orchestrator has two modes of operation:

- *Interactive mode*: A command line interface provides commands to control the devices and other nodes on the network. This mode of operation is not suited for large tests.
- *Test Procedures* are Groovy scripts of more complex command sequences. Such scripts au-

tomate coordination between BACnet/IT devices on the network. These test procedures have to be independent from each other to avoid complications, e.g. identifier collisions. So it is good practice to terminate all used devices after a test is completed.

The test procedures are divided into three main stages:

- *Setup* deploys all required devices for the test with a given configuration.
- *Test* contains the commands that must be executed to run the test including the evaluation of its results.
- *Teardown* phase contains the cleanup routine for the test.

### Spy

The Spy is a Java application running on a virtual or physical machine. It is an interface for the Orchestrator to communicate with the Domain Host. Additionally, it manages operations on the node. These include deploying and halting of Domain Hosts and executing shell commands for reading CPU and memory usage and starting *tcpdump*, which is used to write network communication to a file. Afterwards, this file is sent to the Orchestrator for evaluation.

In essence, the Spy is a program that boots with the computer and controls some of the operations in the host system. It is similar to malware on a compromised computer in a botnet. To mitigate potential damage, we have limited a Spy's scope of shell commands to a few that can not do any harm to the host. This however, does not mean that no damage can be done to the network: A Spy can easily flood the network with traffic produced by a BACnet/IT device. Indeed, a DDoS attack on one BACnet/IT device conducted by other BACnet/IT devices is a relevant test case.

### Domain Host

To run a test environment with distributed interacting devices a well-defined setup is needed. For example, a BACnet/IT system includes a BDS, a directory service, and a number of devices. These devices maintain objects and properties, which have to be configured. Further behavior has to be configured as well. For example, the configuration tells a device it has to announce itself to the BDS.

During development and testing of the BACnet/IT, the involved components have to be in an initial state. Thus, we have to simplify and automate the process of making the system ready for further tests.

A *Domain Host* has to fulfill two main tasks:

- It configures and starts the devices at a specific node according to a given configuration, which defines for example the number of devices and the value of their objects and properties, the IP address of the DNS, the usage of

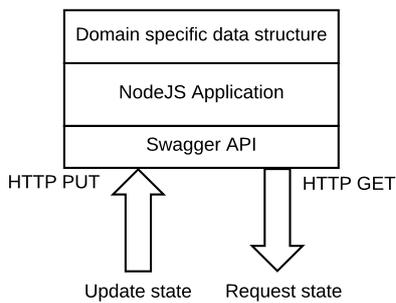


Figure 4: Simple architecture of the Logger

TLS, BasicAuth or CORS during communication and so on.

- A Domain Host acts as an interface between a Spy and the BACnet/IT devices on a node. As described above, it receives commands sent by a Spy, passes the commands to BACnet/IT devices, and forces them to communicate according to the received command.

### Logger

The *Logger* receives and maintains logging data from all Spies, Domain Hosts, BACnet/IT devices and other BACnet/IT system components. It is a nodeJS application keeping track of the events and state changes (see Figure 4). For example, devices announce themselves with the BDS, send messages to each other, and may change the value of their properties. Besides the devices, the communication stack of a device has also a changing state and sends logging information to the Logger, and even the zone file entries of the DNS changes are reported to the Logger.

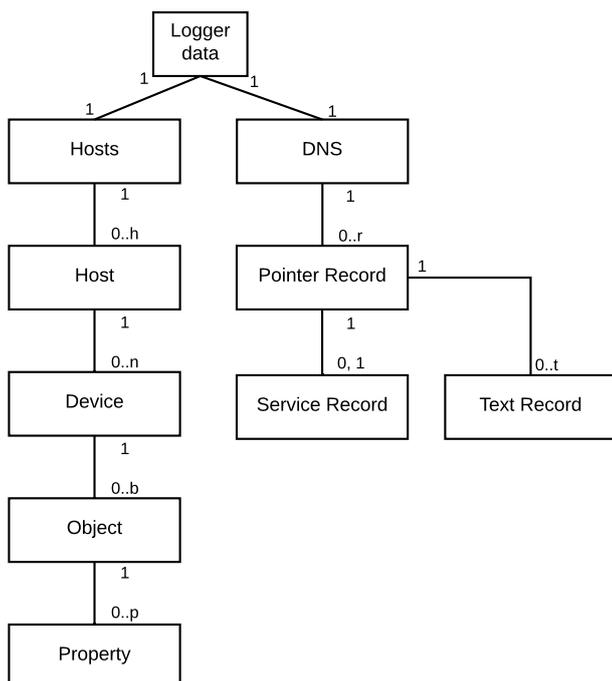


Figure 5: Data structure of the Logger applied to the domain of BACnet/IT

Every component of the system (e.g. a Spy, a Domain Host or even a BACnet/IT device) decides when and what information it will send to the Logger. After a message from another device is received, a device may inform the Logger about that incoming message.

Getting an overview of all the notable events and state changes is a challenge. Observing several standard output logs at the same time is arduous and confusing. Thus, one could adjust the logging level to standard output to get a proper system overview, but this is inconvenient, because the logging level can differ per component and test case. Heading this problem, our solution is oriented towards a notification mechanism. Whereby one designated Logger collects all the information it receives. Communication with the Logger is through a RESTful API.

For the definition and implementation of our RESTful API we use *Swagger* [SWG]. Swagger provides a number of tools to design a RESTful API and offers simple client and server implementations in different programming languages.

A typically RESTful API based on Swagger is defined as follows:

```

/route/{entity}:
  [HTTP METHOD]:
    parameters:
      - name: [name]
        in: path
        type: string
    responses:
      200:
        description: [text]
      404:
        description: [text]

```

Our Logger stores the received data in a domain specific data structure. For BACnet/IT, for example, we use the data structure described in Figure 5. With this data structure, we are able to keep track of all the important and notable events and state changes during the runtime of our BACnet/IT implementation.

The same RESTful API is also used by a *Monitor*. In our understanding a Monitor is just an application that analyzes and visualizes data from the Logger.

### Monitor

Any Monitor implementation can request the current state from the Logger in JavaScript Object Notation (JSON). In our implementation of the Monitor we use D3 to create sunburst charts [D3JS]. D3 is a popular JavaScript library for data visualization. Sunburst charts fit very well to our 1-to-m entity relations of the data representation and let one zoom into areas of interest.

In Figure 6 we display information we care about during development and testing of the BACnet/IT system. On the innermost ring we see *Hosts*

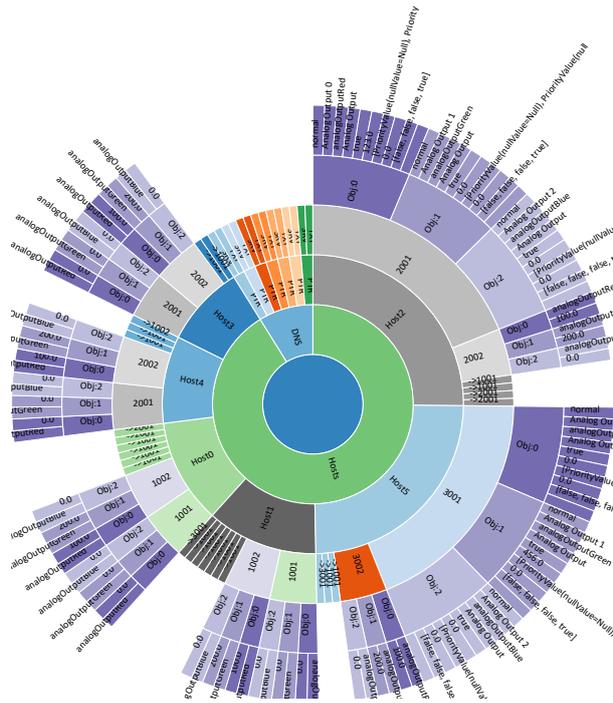


Figure 6: Sunburst visualization of Logger data in Monitor

and DNS. A host contains several devices, e.g. *Host2* contains devices 2001 and 2002, a device may have several objects, e.g. device 2001 has objects Obj:0-2, and finally an object consists of several properties (visualized in the outermost ring). In the upper part of Figure 6, the current state of the DNS of the system is fanned out.

**Example 1 using the ATE**

The diagram in Figure 7 describes how the ATE can be applied to the *Speed Test* example presented previously. For the test we need two instances running Ubuntu with a Spy installed on each

one. One instance will host a BACnet/IT device simulating a light switch (A1) and the other will pose as an Internet connected light bulb (B2). The goal is to measure the roundtrip time of a request (BACnet/IT message) from A1 to B2 including a response from B2 to A1 confirming a successful execution of that request. The test is stored in a Groovy script (called procedure). It can be run not only in local environment but over different networks or even over Internet. Also, additional traffic may be generated on the network to simulate real-world conditions.

The test consists of the following steps:

- A. The Orchestrator deploys the Spy on a remote machine via SSH in case the Spy is not already deployed. The SSH tunnel is closed afterwards and all further communication is carried out over WebSocket.
- B. Maintenance Commands signal both Spies to deploy a Domain Host with a configuration delivered as a serialized object.
- C. Maintenance Commands to Spies invoke the tcpdump process.
- D. A BACnet Command to the Spy on instance 1 enters the Domain Host and triggers a BACnet message dispatch from device A1 to B2 on instance 2. The message destination is only provided as a BACnet/IT device identifier. A1 doesn't know the destination IP address yet and resolves the identifier via DNS. Finally, this message is sent to the returned IP address.
- E. When a response has been received, the test is completed, the tcpdump processes are terminated, and the tcpdump file is delivered by the Spy to the Orchestrator, which analyzes both files and computes the test results.

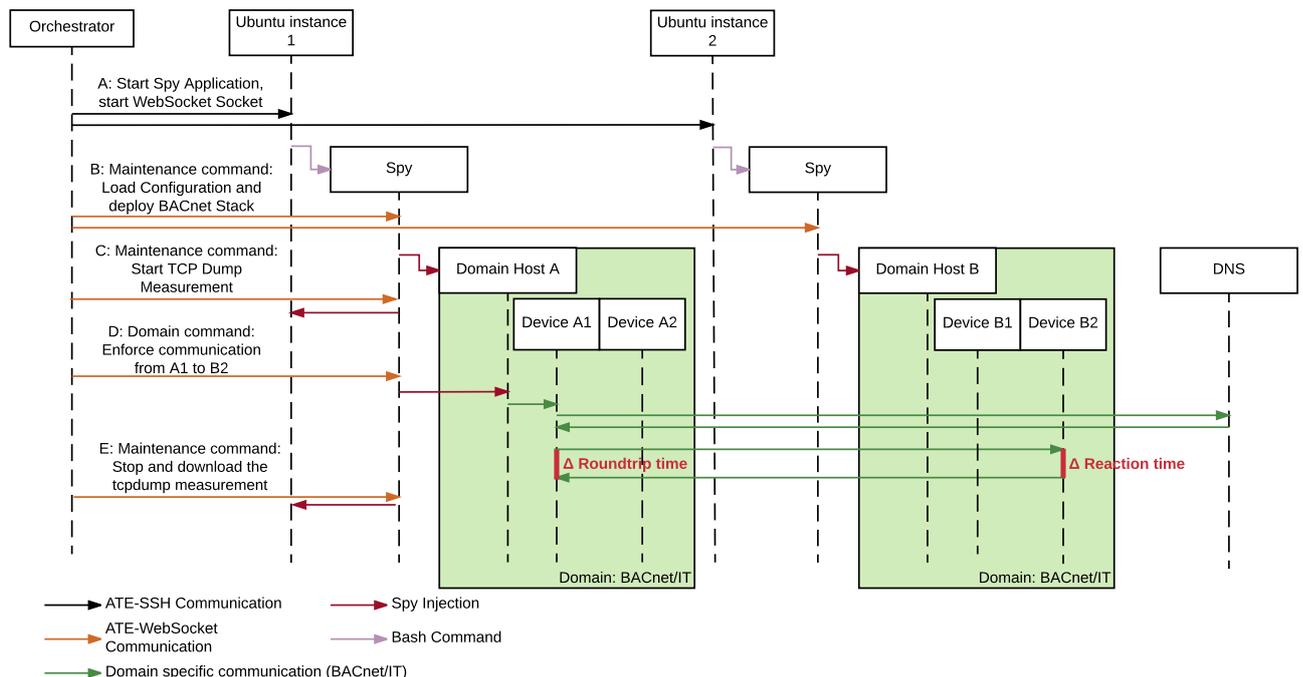


Figure 7: Communication flow initialized by the Orchestrator in example 1 (time measurements)

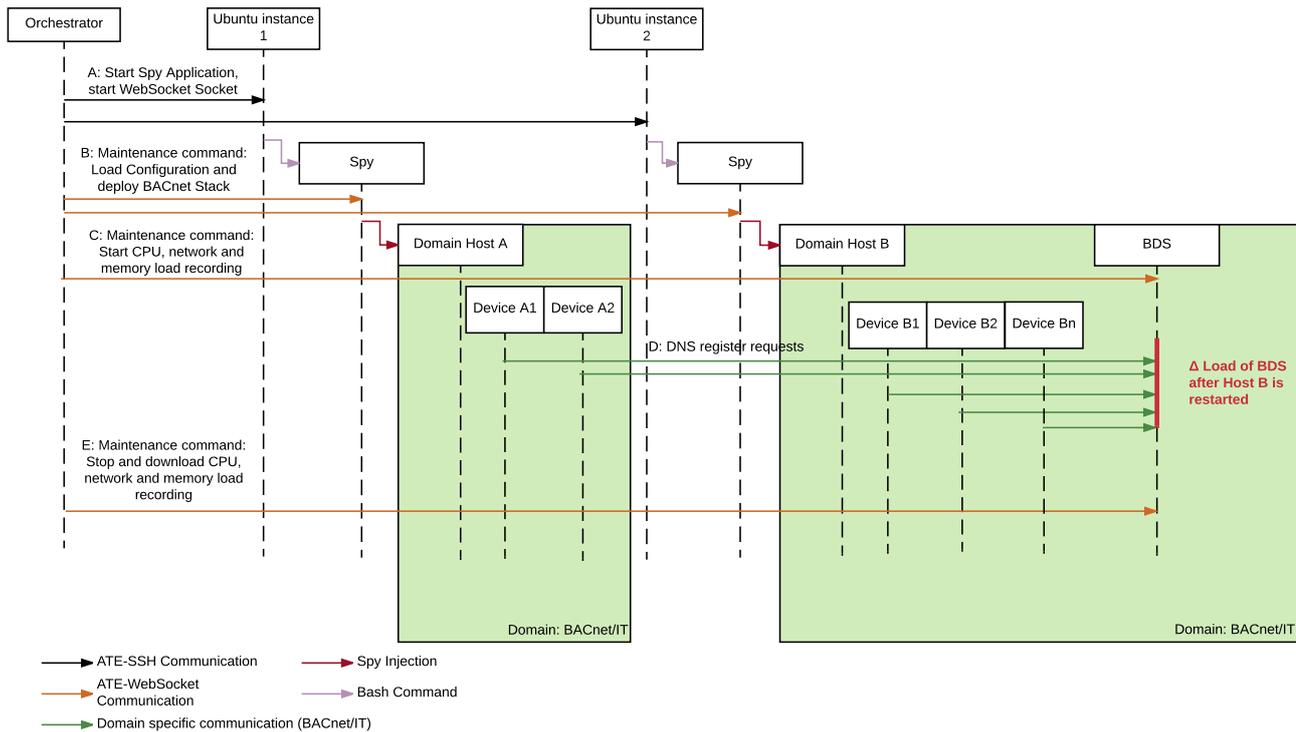


Figure 8: Communication flow initialized by the Orchestrator in example 2 (power outage)

### Example 2 using the ATE

Let us pick up the second test example explained previously that occurs when all devices in a building power up at the same time and announce their availability to the BDS (Figure 8). During this process the BDS is flooded with registration requests from hundreds of devices residing on dozens of hosts. One can describe this massive load on the BDS as a DDoS attack.

During the test, we deploy a BDS device and ten Domain Hosts with ten BACnet/IT devices each. Depending on the configuration these devices may send a registration message immediately or wait a random time span before sending a registration message to BDS. This deferred registration helps to reduce the peak load on the BDS and avoids any dropped messages. The load is indicated by CPU and memory usage, which is recorded during the test. After recording, the CPU and memory usage is evaluated, and the evaluation results lead us to an improved device configuration for this network.

The test consists of the following steps:

- A. The Orchestrator starts a Spy on each instance over SSH.
- B. CPU and memory usage recording is started on the instance hosting the BDS.
- C. For the main test, Domain Hosts are deployed on every instance. During startup, they deploy their BACnet/IT devices and automatically register them using the BDS.
- D. The test is completed when all hundred devices have been registered.

- E. The Orchestrator collects recordings from the BDS and evaluates the results.

### Conclusions and Outlook

The Automated Test Environment presented in this article allows us to simulate real life scenarios like a power outage, BACnet/IT component failures, malfunction of involved network components or a complete rearrangement of the existing IT network. Without this ATE we wouldn't be able to automate tests in IoT systems in such a straightforward and efficient way.

In the future we want to improve the stability of our ATE and develop a first adaption to another use case beyond building automation. Further, continuous integration plugins and automated system sanity checks would extend the functionality and a graphical user interface could improve the ATE experience.

### References

- [BAC] BACnet, official website: <http://www.bacnet.org/>
- [BAC16] Proposed Addendum bj to Standard 135-2016, BACnet® - A Data Communication Protocol for Building Automation and Control Networks, December 2016. [http://www.bacnet.org/Addenda/Add-135-2016bj-apr1-draft-2\\_chair\\_approved.pdf](http://www.bacnet.org/Addenda/Add-135-2016bj-apr1-draft-2_chair_approved.pdf)
- [D3JS] Data-Driven Documents: <https://d3js.org/>
- [JDTS] Java Device Test Suite: <http://www.oracle.com/technetwork/java/embedded/javame/javadevice-140362.html>
- [SWG] SWAGGER open source framework: <http://swagger.io/>
- [TET] The Test Environment Toolkit, TETWorks on the web: <http://tetworks.opengroup.org/>