A Distributed and Decentralized IoT Data Exchange

Shrey Baheti, Yogesh Simmhan
Department of Computational and Data Sciences
Indian Institute of Science, Bangalore, India
shreybaheti@iisc.ac.in, simmhan@iisc.ac.in

***Abstract*—One of the ways in which city can improve the livelihood of its citizen is by creating a local economy around the sharing of data from IoT devices. People, communities and companies are not yet ready to widely share their data collected by IoT devices with public service provider due to privacy, security and trust issues. Fairness in exchange and monetization of data are also factors. In this report, we address some of the technical challenges involved in realizing such a data exchange. We leverage emerging distributed ledger technologies to build decentralized, trusted, transparent and access controlled architecture for IoT data exchange. We discussed a hierarchical structure to create IoT data exchange for smart community application. We have also discussed design requirements and system designed for general IoT data exchange and will be prototyping for smart community application meeting those requirements.**

I. Introduction

Internet of Things (IoT) data are increasingly viewed as a new form of massively distributed and large scale digital assets, which are continuously generated by billions of devices. IoT devices are capable of sensing the environment and send collected data to centralized servers for processing, analysis and subsequent actions. Gartner, Inc. forecasts that 20 billion connected devices will be in use worldwide by 2020[1]. Within the IoT domain, there are several types of applications, such as smart cities, smart communities, and smart home. We see smart city (SC) is the one improving the lives of its citizen by data-driven services provided by the municipal government or by the companies or by the individuals.

To enable these data-driven services like [1], there is a need of data sharing among communities to build a smart city. Most of these data is currently owned by individuals, companies or governments, and held privately. For example, in a public transport network like the Delhi metro, the density of personal metro cards swipes over time at individual metro stations may be useful not only to transportation authority, but also to taxi companies, which can benefit from the knowledge of any anomalous passenger traffic pattern, i.e., by placing their fleet on the right stations at the right time. Perera et. al presented in [2], sensing as a service model for smart cities by IoT. This data sharing8 requires interested parties to cross their trust boundaries to share their data with other communities. In this way, data can be viewed as a new tradeable digital asset. However, the lack of trust and incentive in trading such assets is hindering their larger availability from data producers to consumers.

The limitation of IoT devices in terms of compute, memory, network and power leads to challenges in designing secure IoT data exchange. Some of these challenges are heterogeneity, trust, reliability, availability, and much more [3]. Traditional *centralized solutions* such as Government database[2] and Microsoft Azure Data Marketplace[3] are designed to provide a single platform where data owners can buy, sell and share data with other members. However, centralized solution would be a costly option with intermediary fees and charges to support the rapid rise in connected devices. It is also important to note that these solution is a single point of mistrust, failure, hacking and compromise. The data owners needs to accept their terms and conditions in order to access their data exchange platform which may result in weak privacy guarantees. There have been cases in the past, where the private information of users is shared with third parties without their consent. More details covered in the Related Word, in Sec. II-A.

In contrast to centralized approach, *Peer-to-Peer (P2P)* or decentralized approach came into existing. P2P has gained advantage over traditional centralized solutions by decentralizing the Data Exchange platform such as the BitTorrent protocol. These platforms do not need a trusted third party to provide a data exchange and they remove the intermediary charges. But, P2P lacks in the functionality of access control, data immutability and provenance of data sharing. More details covered in the Related Word, in Sec. II-B

Hence, there is need of a hybrid solution that can provide the platform for IoT data exchange that can encompass the benefits of centralized and P2P approaches. We need a P2P system where data is distributed and is not centralized in the sense that single website (cloud server) holding all the data making it a bottleneck.

We propose a distributed and decentralized architecture for IoT data exchange that offers a solutions to the following inherent questions:

1) How can IoT data producers exchange their private data securely with consumers in a trustless, distributed environment, for monetary or attribution benefits?
2) How can consumer trust the validity of data exchanged?
3) How data producers can claim ownership of data?

[1]https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016

[2]https://data.gov.in/

[3]https://azuremarketplace.microsoft.com/en-in/marketplace/apps/Microsoft.DataCatalogGalleryPackage?tab=Overview

4) How can we establish provenance of data for authenticity?
5) How can we support streaming data, with freshness and scaling?

The rest of this report is organized as follows. Section III formulates the problem definition with an example application. Section II introduces the related work which covers background of Centralized, Decentralized Data exchange, Distributed Ledgers and Blockchain technology along with some existing IoT data exchanges. Section IV mentions the design requirement categories into two subsections as System and security requirement. Section V introduces the high level overview of architecture, entities and their roles, system design and sequence diagram of our distributed and decentralized IoT data exchange. Section VI describes our previous work with some experiments. Section VII concludes the work with some suggestions for the future development.

II. Background and Related Work

A. Centralized Data Exchange

Centralized Data Exchange relies on brokered communication models, known as client/server paradigm. They make use of huge processing and storage capacities of cloud servers to store the data provided by device owners. The device owner needs to register their devices on to central server in order to perform data sharing. In order to communicate between the devices, the communication has to exclusively go through the cloud, even if they are few feet apart. For a decade, this model has connected generic computing devices and small scale IoT networks, it will not be able to respond to the rapid rise in IoT ecosystem. With billions of connected devices, having a centralized platform to govern and manage heterogeneous devices can be a bottleneck and single point of failure [4]. You can expect an increase in the intermediary charges as the demand grows.

Such Centralized Data Exchanges are governed by large corporations as they have huge infrastructure to support large data storage and transfer. These companies act as a trusted third party, and inherit the single point of failure drawback of relying on third party services. Even if centralized Data Exchange provides the availability and reliability of data with access control, may still suffers from censorship of data, which leads to access denied to their own data. Data owners also do not have a clear idea of where and how the data they provide is going to be used. Most of these systems do not provide the audit trail of their data's usage.

B. P2P Data Exchange

Decentralized approach to IoT data Exchange would solve many of the questions above. In order to enable communication between devices, Decentralized Data Exchange adopts a Peer to Peer (P2P) communication model. This approach removes the need of installing and maintaining large centralized data centers and distribute compute and storage needs across connected devices forming an IoT network. This will prevent failure in any single peer in a network from bringing the entire network to a halting collapse. The decentralized Data Exchange also removes the need of trusted third party to enable data sharing, as devices can directly communicate with data owners.

However, enabling P2P communication presents its own challenges, main among them the issue of security. The data is available on public network and can be downloaded by anyone part of the network, there is no access control policy mechanism available. This would lead to our problem of claiming ownership of data. Existing P2P IoT data exchange are designed to incentivize the data producers on sharing their data. Also there is no support for audit trail of data consumption by different consumers. Such P2P data exchanges are BitTorrent, used for file exchanges without any trusted third party. The data will be available as long as the copy of data is available with a peer active on the network.

C. Data Exchange using Distributed Ledgers

Here, we explore some of the alternative decentralized technologies that we use in solving our IoT data exchange problem. Here, we have discussed some background about those technologies.

1) Distributed Ledgers: A distributed ledger is a database that is spread across several peers or computing devices. Each peer replicates and saves an identical copy of the ledger. Each participating peer of the network updates its ledger independently. The feature of this ledger is that it is not maintained by any central authority. To make an update to ledger, peers vote on the update and ensure that majority agrees with the conclusion reached. This voting and agreement on single copy of the ledger is called consensus, and is govern by a consensus algorithm. Once consensus has been reached, the peers updates their ledger with the latest agree-upon version of the ledger separately. The consensus algorithm will work as long as majority of the peers are honest and reduces the cost of trust. Distributed ledgers presents a paradigm on how data can be exchanged among the individuals, companies and government.

2) Blockchain Technology: Blockchain is one form of distributed ledger technology that underpins Bitcoin [5]. Not all distributed ledgers employ a chain of blocks to provide a secure and valid distributed consensus. Blockchain is the mechanism that allows transactions to be veried by a group of unreliable peers. It provides a distributed, immutable, transparent, secure and auditable ledger. Blockchain consists of two types of elements:

1) Transaction are the actions created by the peers of the network.
2) Blocks record these transactions and make sure they are in the correct sequence and have not been tampered with. Blocks also record a time stamp when the transactions were added.

When someone wants to append a transaction to the chain, all the participants in the network will validate it. They do this by applying an algorithm to the transaction to verify its validity. What exactly is understood by valid is defined by the

	Public	Consortium	Private
Consensus participants	All miners (permissionless)	Selected set of nodes (permissioned)	One organization (permissioned)
Read Permission	Public	Public or restricted	Public or restricted
Efficiency	Low	High	High
Data security	Very high	Could have data breach	Could have data breach
Example	Bitcoin, Ethereum	R3 Corda, B3i	MultiChain

TABLE I: Types of Blockchain Architectures

blockchain system and can differ between systems. Then it is up to a majority of the participants to agree that the transaction is valid. A set of approved transactions are then bundled in a block, which gets sent to all the nodes in the network. They in turn validate the new block. Each successive block contains a hash, which is a unique fingerprint, of the previous block.

The blockchain allows access to all transactions that have occurred since the genesis transaction of the system, and can be veried by any peer of the system. A blockchain is distributed across and managed by peer-to-peer networks. Since it is a distributed ledger, it can exist without a centralized authority managing it, and its data quality can be maintained by database replication and computational trust. When someone wants to add a transaction to the chain, all the peers in the network will validate it. They do this by applying an algorithm to the transaction to verify the validity. The algorithm is defined by blockchain system and can differ between the systems Then, it is up to a majority of the participants to agree that the transaction is valid. A set of approved transaction on a blockchain is grouped together and organized in blocks. The blocks are then linked to one another and secured using cryptography. A blockchain is essentially a continuously growing list of records. Its append-only structure only allows data to be added to the database: altering or deleting previously entered data on earlier blocks is provably impossible. Blockchain technology is therefore well-suited for recording events, managing records, processing transactions, tracing assets, etc.

Three main types of blockchains exists as of now, as summarized in Table I:

1) In a *Public Blockchain*, everyone can read or write data. Some public blockchain may limit the access to just reading or writing. Some characteristics of public blockchain:
   - Code is open source.
   - Anyone can download the code and join the network as peer.
   - All peers validates the transactions in the network and thus participates in the consensus process of validating block to be added to chain.
   - Anyone can send a valid transaction through network abd expect to see it included in the blockchain.
   - Everyone can read transactions recorded on the blockchain via explorer.
   - Transactions are transparent, but pseudonymous.
   - Transaction confirmation rates is slowest as compared with other blockchain implementations.

   Examples are Bitcoin [5], Ethereum [6].
2) In a *Private Blockchain*, all the participants are known and trusted. This is useful when the blockchain is used between companies that belong to the same legal parent entity. Some characteristics of private blockchain are:
   - Write permissions are kept centralized to single organization.
   - read permissions can be public or restricted depending upon the use case.
   - Transaction confirmation is fastest.

   Examples are Multichain [7], Hyperledger Fabric [8]
3) In a *Consortium Blockchain*, all participant are known and trusted, but few of them act as stakeholder to govern the consensus of the blockchain. Some characteristics of the consortium blockchain are:
   - read/write permissions are kept within the group of selected organizations.
   - consensus process is controlled by pre-selected set of peers.
   - Transaction confirmation is faster and provide more transaction privacy as compared to public blockchain.

   Examples are B3i (Insurance), R3 corda (Banks) [9]

D. Existing IoT Data Exchanges

The idea of considering data collected from IoT devices as tradeable assets is not new. The author in [10] provides a centralized broker based solution where the device owners register their sensors along with standardized description and data consumers query to find the relevant data depending upon their budget to procure data. Their model does not take in consideration that people are not willing to share their their data on a public site. one example of a real-time IoT data marketplace is presented in [11]. It aims to provide a real-time marketplace with a single website containing information about all sellers and their data. They have designed it using centralized pub-sub broker, with access control of data buyers turned (on/off) based on their payment.

There have been work done in the field of IoT data sharing where authors tries to use Blockchain for privacy, security, availability, accountability [12], authorization [13], and trust [14]. The author proposed blockchain-based sharing services in [15].

Our work is closely related to [16]. The authors presented a decentralized broker based mechanism of sharing IoT data. Their model is based on honesty of both parties sharing the details of number of messages communicated in order

claim the success of trade. These approach can lead to more transaction failure due to transiency of IoT devices. In partial data transfer, this approach will not provide any reward to data owner as data consumer can always lie about the receipt of messages. This is prime requirement of secure data sharing in exchange of reward which will be supported by or design.

[17] proposes a lightweight blockchain-based architecture for IoT. The model removes the proof-of-work and concept of coins from conventional blockchain to reduce the computational overhead and energy consumption. As a proof of concept the provide a use case in [18]. The author discuss the case study of smart home to justify IoT security and privacy claims. They used CoojaSimulator to simulate the smart home architecture and succeeded in reducing traffic, processing time and energy consumption keeping confidentiality, integrity and availability intact.

Authors of [19] presents and compare different forms of distributed ledger potentially suitable for front-end application development including Ethereum, Hyperledger Fabric and IOTA. The author shows based on the experiments that a full Ethereum node is not reliable to run on constrained IoT device. Therefore, the architecture with remote ethereum client is a viable option.

Conoscenti, et al. [20] discuss the various use cases of the blockchain and which factors affect integrity, anonymity and adaptability of blockchain technology. They focus on leveraging blockchain and peer-to-peer approaches for a private-by design IoT where data produced by devices are not entrusted to centralized companies. They also mention that large blockchain system like Bitcoin are the most secure, but not scalable. hence not suited for IoT.

Our work is complementary to above approaches in the way we have modelled our architecture. The above solutions are either focusing on purely centralized or decentralized solution. In this way, they cannot serve as a solution for complete IoT data exchange. We propose a hierarchical structure where private ledger can be seen as centralized and as you go up the network transforms into decentralization. Thus, you get better privacy control of your sensitive data.

III. Problem Formulation

We first introduce the base formulation of IoT data exchange and will try to formulate the 5 questions mentioned above. Then, we describe the example application of smart communities and how it can be benefited by a decentralized and distributed IoT data exchange.

A. IoT data exchange

In essence, the objective of the IoT Data Exchange is to share data from device owners who owns IoT devices to users who consumes them. The sets of *device owners*, *data consumers*, and *IoT devices* are denoted by *DO*, *DC* and *S*, respectively. Note that the user may act as both a *DO* and *DC* at the same time. IoT devices owned by DO_i can be denoted by $S_i = \{s_{i1}, s_{i2}, \cdots, s_{i_n}\}$. Each s_{ij} is associated with its meta-data, denoted by m_{ij}. Here, meta-data describes the IoT device relevant information such as sensor type, location.

The distributed ledger, denoted by *L* is created which keeps track of all the device owners and their sensors meta-data information available. The data consumers *dc* can search through the L, and identify the most relevant IoT device's data *s* owned by device owner *do* they want to consume. Then, dc will request the do to share the data in exchange of price per unit data (mentioned in meta-data of IoT device) associated with it. If parties agrees on terms of data exchange, the transfer of data will start. The transfer will continue as long as dc has some credits.

1) Secure Data Exchange: The primary requirement of IoT data exchange is to provide a mechanism to transfer data from data owner to data consumer in exchange of some incentive. The transfer of data must be secure enough to support confidentiality, Integrity and Availability, known as CIA triad. Since the environment is distributed comprising of trust-less peers, it is necessary to implement security CIA triad.

2) Validity of Data Exchanged: When Data owner and data consumer agrees on exchanging data, there is a need of validating the data exchanged. It is to ensure that the data which is transferred to consumer is generated by the same device from which it has requested.

3) Ownership of Data: In decentralized IoT data exchange, claiming the ownership of data by a particular sensor is in itself a challenge. the ownership of data will help data owner to take control of permissions related to that data. In open P2P data exchanges, data available on distributed storage is not owned by anyone. But, for our use case, we need a mechanism to provide data owners to claim ownership of data so that they can control the access of data consumers for the data.

4) Provenance of Data: The provenance of data is the mechanism of keeping track of data. It starts with sensors sensing the environment and producing the raw data. The raw data is primarily owned by data producer who owns the sensor. With the help of IoT data exchange, the data can have multiple consumers and they may derived some insights based on the raw data. The linking of these data movement can provide insights about who, when and how got access to our data. The provenance helps in providing authenticity on origin and journey of data.

5) Streaming Data: The data generated by IoT devices is valuable only if it can be processed. These data loses its value as time progresses, commonly known as fast data. Timely deliver of fast data is possible only if streaming data is supported by our IoT Data Exchange. The streaming data can be seen as a subscription based model, where consumers request for data from an IoT device in advance with an expiration date. The producer will be charge per unit of data shared with consumer periodically.

B. Example Application

We present the use case of a distributed and decentralized IoT data exchange using Blockchain, *Smart communities*. We proposed a hierarchical model based on distributed ledger

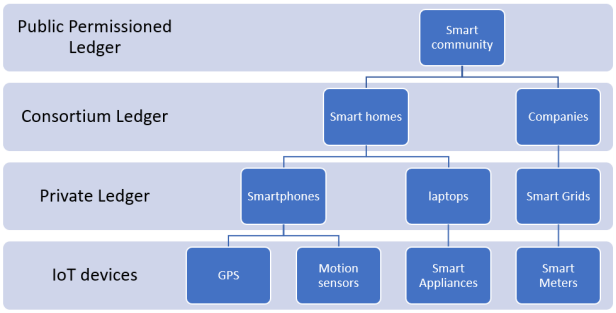


Fig. 1: Hierarchical Structure for smart community

technology. The high level overview of architecture is shown in fig. 1. The bottom layer comprise of IoT devices which includes Sensors or actuators. These devices are the one which generates data by sensing the environment like location, temperature, etc. Theses devices remain private to device owners. The smart home can have 1 or more IoT devices which are under access control of the device owner(s). The smart home can be consider as the private ledger where device owner can grant/revoke permissions given to different devices which are part of the same network. The distribute ledger could help in maintaining auditable ledger as a proof when asked by consumer to show the validity of the data collection. These smart homes can combine to form a P2P network among themselves and form a consortium ledger. In this all the smart home owner can act as miner considering they have unconstrained resource devices to run mining node. This owner can also publish curated data collected from their sensor or other sensors of different smart homes. The smart homes belong to same community will share the common network so we can use private permissioned ledger. Then, group of smart communities distributed all over the city(ies) can join and form public permissioned ledger where sharing data could help in improving the life of the smart citizens.

Managing resources in an effective way especially during peak-demand periods is essential for maintaining reliable and sustainable resources in a smart community. The pressing challenge here is privacy-sensitive of user's data , analyzing one's energy usage information can leak their household appliances used, time of use, etc. There are existing techniques to provide privacy to data.

With regards to smart communities data sharing, a consortium blockchain comes out to be better fit. As per the smart home owner is concerned, first he/she needs better livelihood within their community and then within city. A group of smart communities sharing data among themselves could benefit by analyzing the usage pattern and preparing for the peak demand in advance.

The above IoT architecture is hierarchical which gives us fine grained control over the participants to join in the network and share/sell their data. All devices are kept private to device owners. Device owners can decide which device data they want to share and which not. The device owners (smart home owners) can join and build wide area network based private permissioned blockchain network to improve the scaling factor and reduce transaction cost. These second level of private blockchain will make the smart communities. Smart communities could benefit from their trusted residents to get the resource usage information through sensors data sharing and in exchange could give assets/coins tradeable on the same platform.

IV. Design Requirements

An IoT data exchange must support buying and selling activities of data providers and data consumers, such as listing and discovering of data, pricing the data, permissions (access control) set by owner, making payments and transferring of data, security and privacy. The below requirements are based on the survey on [21]

The IoT data exchange design requirements can be categorized into two:

A. System Requirements

Here, we present high level system requirements for a decentralized and distributed IoT data exchange, based on the needs of data producers and data consumers.

1) *Producer and Consumer:* We assume that there are set of producers of data in a city with one or more static or streaming data collected from IoT devices (these data could be anything from room temperature to health), and buyers interested in their data makes them the fundamental participants of IoT Data Exchange.
2) *Data listing and discovery:* We need to provide a mechanism for producer and consumer to find each other, and for consumers to know the type of relevant data is available, without merely posting them on a single website (traditional centralized approach to create data exchange). Well P2P approach has provided an search and query mechanism in unstructured overlay networks as well as indices based on distributed hash tables in structured overlay networks. We make use of the P2P approach to search for relevant data. We can ask producers to post their sensors meta-data information (sensor type, data rate, latency, pricing, location, server address, etc.) on the common ledger where everyone can see.
3) *Low latency:* The sensors data has its highest value when it is delivered from producer to consumer with minimum latency overhead. Edge computing could play the major role as devices close in proximity can provide data at minimal latency. The total time from the creation of data at sensors to the point where the data is delivered to consumers must be reasonably small so that it can be processed and timely decisions to react to the current situation can be made.
4) *Data Streams:* The data transfer could easily be delivered from producers to consumer in form of stream events.
5) *Data Store:* Since IoT devices are resource constrained, we need a data store to store generated data for later utilization or transfer. Data can be stored on centralized servers or decentralized servers. Centralized servers easy

to scale but suffer from latency, weak privacy guarantees, and single point of failure whereas Decentralized servers are not affected by the weaknesses of centralized servers.

6) *Access control and Identity management:* It may be required to add an identity layer for role-based access control. It should be possible to provide the control to device owner to choose at device level to grant/revoke permissions rights for any data consumers.
7) *Data Transfer and Payments:* Once a consumer learns about the producer and make a request to exchange data, it can use the designated protocol to collect data in exchange for payments or other incentive. one can make use of existing MQTT protocols to provide data transfer integrated with payment channel and a distributed ledger to store the key transactions records such as contract, invoices.
8) *Privacy:* It is essential to ensure that the privacy-sensitive data is not leaked/logged/store on to the public ledger/domain/Exchange. One way to ensure this is to keep the data that is exchanged on separate, private , encrypted, data channels between producers and consumers. Or put the sensitive information on off-chain storage system with the hashed address pointing to actual data. That hashed pointer can only be access after the successful transaction.

B. Security Requirement

Here, we present high level security requirements required to ensure security and privacy for a decentralized and distributed IoT data exchange. Security and privacy are fundamental principles of any information system. We refer to safety as the combination of integrity, availability, and confidentiality. Typically it is possible to obtain security using a combination of authentication, authorization, and identification. These concepts are defined below:

1) *Security Threats*: The platform must be secure enough to handle known security threats and should not affect the data loss. The security threats include DDoS, man-in-the-middle, Sybil attacks, snooping, sniffing. The data exchanged end-to-end might need to be encrypted.
2) *Integrity*: it is the certainty that the information has not been altered, except by those who have the right to make these changes. In our context, the data is stored on a distributed ledger, and can only be modified by the permitted users by invoking transaction to publish data. The transaction is verified for the authorization of the user who tries to publish data.
3) *Availability*: it ensures that users of a given system will be able to use it whenever necessary. In other words,the service is always active when requested by a legitimate user, and this requires the communication infrastructure and the database.The ledger achieves this objective by allowing peers to establish connections with multiple peers and to maintain the blocks in a decentralized way with various chain copies on the network. If the data is large, off-chain solution can be used to provide availability.
4) *Confidentiality*: it is the guarantee that the unauthorized persons will not obtain information. That is, only those with the rights and privileges will be able to access the information, whether it is in processing or transit. To ensure this principle, Streams (defined in later section) provide a natural way to support encrypted data on a blockchain, as follows:
   a) One stream is used by participants to distribute their public keys for any public-key cryptography scheme.
   b) A second stream is used to publish data, where each piece of data is encrypted using symmetric cryptography with a unique key.
   c) A third stream provides data access. For each participant who should see a piece of data, a stream entry is created which contains that datas secret key, encrypted using that participants public key.

   This provides an efficient way to archive data on a distributed ledger, while making it visible only to certain participants.
5) *Authentication, authorization, and auditing*: this seeks to verify the identity of who performs a specific function in a system, check what rights that user owns, and store usage information for that user. Since the streams can be only be created by those who have permissions. When a stream is created, it is open or closed. Open streams are writable by anybody who has permission to send a transaction, while closed streams are restricted to a changeable list of permitted addresses. In the latter case, each stream has one or more administrators who can change those write permissions over time. All the transactions maintained in a distributed ledger visible to all peers.
6) *Non repudiation*: it guarantees that a person cannot deny an action in a system. The non repudiation provides evidence that a user performed a specific action such as transferring money, authorizing a purchase, or sending a message. As all transactions are signed, a user cannot deny that he has done it.

V. System Design

A. High level Overview Architecture

As presented in Fig. 2, IoT devices and Device Owners forms a private network where device owner can initiate a distributed ledger as admin and grant access to devices such as send/receive transaction. This distributed ledger is used to control and audit the communications and provide access control between devices. The device owner has the full control of IoT devices within its private network and is responsible for dealing with all communications inside and outside of the private network. In this scenario proof of work is delegated to known peers and are responsible to maintain the network. The assumption here is that a device owner is equipped with the unconstrained resource device capable of running private blockchain as admin such as laptop. The IoT devices

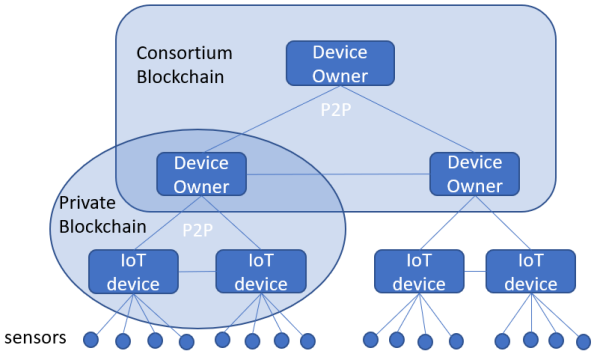


Fig. 2: Proposed IoT architecture using Blockchain

generating data uses stream to publish/subscribe data on to distributed ledger. This enables device to share data among peers even if some of them are offline.

B. Entity and their roles

We have defined our entities in Section III.

Each entity will have to generate its set of public private keys pair, denoted by PK_x, RK_x, respectively. These keys are used as identity for the devices and device owners and gives pseudonomity to devices and their owners. The access control (Permissions) list, denoted by ACL, are defined for each device by there device owner. The permissions could be *connect*, *send*, *receive*, *issue*, *create*, *mine*, *activate* and *admin*.

Each IoT device is associated with 1 or more permissions to participate in network. The IoT devices generates stream of data in real-time and these streams can be shared among the connected devices. The device owner creates a device stream to push data on to distributed ledgers. Streams provide a natural abstraction for distributed ledger use case which focus on general data retrieval, timestamping and archiving, rather than the transfer of assets between participants. These can be considered as the what, when and who of a shared database.

C. System Interfaces

Below are some of the operations available with device owner to register, access control IoT devices.

- RegisterDevice (<device_identity>): A device owner (DO_i) needs a public key of an IoT device to register it on a private network. Transaction T having public key $PK_{s_{i1}}$(device_identity) and basic permissions (connect, send, receive) are passed as inputs is triggered. After successful validation of Transaction T, the device is eligible to connect to private network anytime, anywhere as long as they are within the same network and miner is available to validate future transactions. By default, the one who creates a chain act as Admin and miner to create blocks and confirm transactions. There can be multiple admins and miners.
- DeregisterDevice (<device_identity>): A device Owner (DO_i) can deregister device at anytime and stops all communication with the other devices and ledger.
- GrantPermissions (<device_identity>, permissions_list[]): A device owner (DO_i can grant permissions and append new permissions to it permissions list and record it on distributed ledger for later retrieval, if a device goes out of the network and rejoins the network.
- RevokePermissions (<device_identity>, permissions_list[]): A device owner (DO_i can revoke permissions and remove permissions from the permissions list and record it on distributed ledger for later retrieval, if a device goes out of the network and rejoins the network.
- ListPermissions (<device_identity>): returns current permissions of IoT device.
- device_id[] ListDevices(<device_owner_identity>): return list of devices owned by device owner DO_i (device_owner_identity)
- txid CreateStream(<device_identity>, stream name, stream type): When device owner wants to store data on network, it creates a stream and stores data on it. The data stored on stream can be optionally encrypted and is visible to permitted users. The function takes input IoT device identity, name of stream and type of stream (open/closed). The function returns the transaction id (txid) of the transaction creating the stream.
- txid DeleteStream(<device_identity>, stream name): when device owner wants to delete the stream of IoT device, he/she can call the function with device identity and stream name belonging to device. The function return the transaction id (txid) of the transaction deleting the stream.
- GrantAccess (<device_identity>, stream name, access list): Device owner DO grants access to publish or subscribe or both to IoT device S1 on stream. The access list contains the list of permissions such as publish,subscribe or both on stream, to be requested by S1.
- RevokeAccess (<device_identity>, stream name): function will revoke access of the device identity from the permitted users list of stream. This function will be successful only if the user has the permission to revoke the access of the other users of streams. By default, creator of stream has all the permissions and can delegate this permissions to more users, if needed.
- PublishData(<device_identity>, stream name, data): IoT devices can push MBs of data by invoking this function on to stream. The transaction will be verified by the miners to ensure that the device has the right to publish data on the stream. The data will be appended to the stream, if transaction is confirmed else transaction will fail.
- string SubscribeStream (<device_identity>, stream name): The device needs permission from device owner to access the stream. If allowed then the function will return the data encrypted using subscribers public key. The input parameter is the subscribers public key as the identity and name of stream. It returns string of encrypted IoT device data.

Each operation can be thought as of transaction on the distributed ledger and making changes to the current state of the nodes (device owners / IoT devices) and their state variables (permissions/streams).

D. Sequence diagram

Fig. 3 describes the sequence of operation executed by different entities during device setup of data producer s1 and data exchange between s1 and s2 IoT devices.

The sequence of operations followed for device setup are:

1) IoT device s1 collects data from sensors sensing the environment.
2) s1 requests device owner do1 to register device on ledger. It also requests to create stream to publish data with write permission to stream.
3) do1 contacts smart contract and call predefined functions to register device, create contract and grant permission to s1
4) After successful validation of request and confirmation, the transaction is recorded. Now, device is successfully registered with write access to stream.
5) The collected data from sensors is published onto stream. The data is published periodically and thus support streaming data.

The sequence of operation followed for data exchange between 2 IoT devices are:

1) IoT device s2 query the distributed ledger to get the identity of device owner of s1 stream.
2) The contract returns the public identity of do1 who owns s1 to s2.
3) s2 generates multi-party signature to purchase the subscription of data stream of s1 in exchange of some coins/assets as per the metadata information available.
4) Smart contract contact do1 to get its approval of granting read permission to s2 and also need to generate transaction which will be granting read access to stream produced by s1 to be consumed by s2.
5) The miner validates all the necessary checks and update the ledger about the read access given to s2.
6) s2 can now subscribe the stream and reads data. s2 cannot write in the same stream as it does not have write access.
7) After expiration time or number of messages read by s2 exhausted, the do1 revokes access of s2 to read stream of s1. This signifies that the subscription of s2 to access s1 is over and the data is shared for the required amount of time or for number of messages.

Note All the steps needs a transaction to invoke a function and is recorded on the distributed ledger.

VI. Preparatory Work

In this section, we describe experiment results from our initial preparatory work on VIoLET, that was published in a peer-reviewed conference paper titled *A Large-scale Virtual Environment for Internet of Things (VIoLET)* [22]. VIoLET can be used to define and launch large-scale IoT deployments within cloud VMs. The IoT deployment is configurable for compute and network bandwidth and network latency. Users can also configure synthetic sensors for data generation on the IoT devices. Recently, we have have incorporated models for CPU resource dynamism and failure-recovery of devices, to allow realistic environment.

VIoLET will serve as the IoT virtual test bed within which to evaluate our contributions to the IoT data exchange using blockchain.

	Deployment→		**D400**	
Device	Cores	CMark	Count	∑CMark(k)
Pi 2B	4	11,557	255	2,947
Pi 3B	4	15,457	96	1,484
Pi 3B+	4	17,888	47	841
NVidia TX1	4	27,070	1	27
Softiron	8	77,800	1	78
		Total		**5377**
Standard_D32_v3 *(host)*	32	3,01,269	20	6,025

TABLE II: Device Perf., Device Counts and Host VM Counts used in Deployments

A. Experimental setup

We use *Raspberry Pis – Pi 2B* with 4×900 MHz ARM32 cores, *Pi 3B* with 4×1.2 GHz ARM64 cores, and 1 GB RAM each and *Pi 3B+* with 4×1.5 GHz ARM64 cores, and 1 GB RAM each. In addition, we have two resources – a *Softiron 3000 (SI)* with AMD A1100 CPU with 8×2 GHz ARM64 cores and 16 GB RAM, and an *NVidia TX1* device with 4×1.7 GHz ARM64 cores and 4 GB RAM. We use Microsoft's Azure *Standard_D32_v3* VMs that has 32 cores with 128 GB RAM. The Dv3-series features the 2.4 GHz Intel Xeon E5-2673 v3 (Haswell) processor. The Standard_D32_v3 costs Rs. 111.05/hour.

B. Results

1) CPU Resource Dynamism: We ran the experiment for 2 hours on D400 deployment. The control interval is set to be 30 seconds means every 30 seconds all devices will go through probabilistic CPU dynamism. The cpu_var_period is set to 600 seconds. This will enforce that all the device must undergo dynamism on and average every 10 mins. Another parameter set is cpu_var_max to 0.2 which enforces the maximum cpu variability to be 20% deviation from the device original cpu/CoreMark. To get the observed CoreMark we have run a python script which runs CoreMark for 2 hours for each device.

Fig. 4a shows violin plot of *deviation%* of the observed CoreMark for D400 deployment with CPU dynamism, where $deviation\% = \frac{(Observed - Expected)}{Expected}\%$. Here the updated CoreMark set by dynamism is used to calculate the deviation%. We see that mean and median of devices are within the expected range $\pm 4\%$ except for SI. The median (in red) and mean (in purple) deviation % for Pi2B is 0.09% and 0.09%, for Pi3B is 1.36% and 1.53%, for Pi3B+ is 1.09% and 1.21%, for

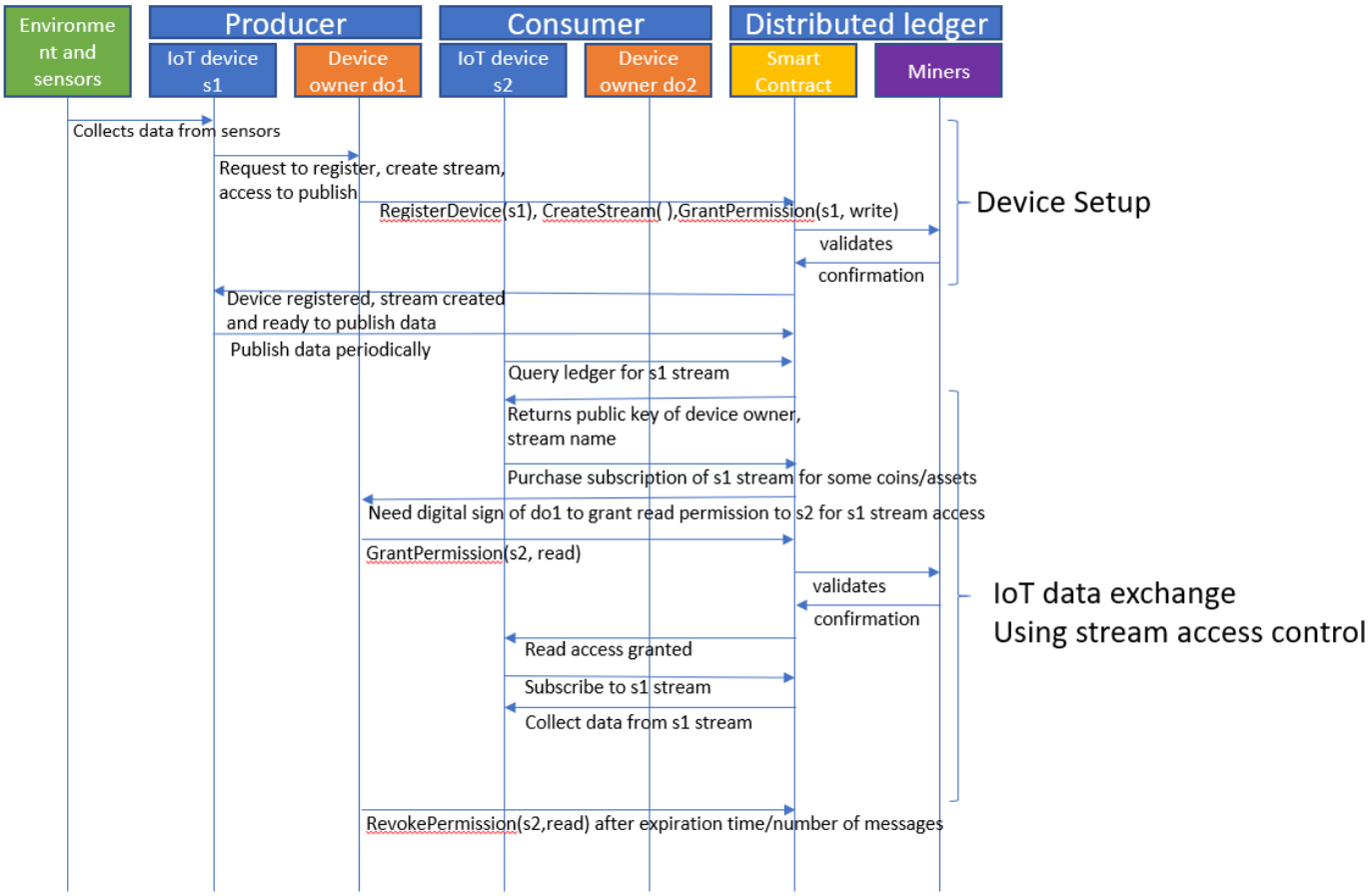


Fig. 3: Sequence Diagram for device setup and data exchange between 2 IoT devices via streams.

TX1 is -3.35% and -4.00%. The SI has a mean and median deviation% of 51.07% and 50.40%. But for the rest of devices there maximum and minimum deviation value lies within the expected range. Hence, we are able to update the compute of device instantaneously.

Fig. 4b shows violin plot of *deviation%* of the observed CoreMark for D400 deployment without CPU dynamism. We see that mean and median of devices are within the expected range $\pm 4\%$ except for SI. The median (in red) and mean (in purple) deviation % for Pi2B is -0.32% and -0.09%, for Pi3B is -0.43% and -0.18%, for Pi3B+ is -0.85% and -1.85%, for TX1 is -16.47% and -15.16%. The SI has a mean and median deviation% of 35.98% and 35.44%. Since SI was allocated on a single VM, we are seeing a over-performance.

When compared Figs. 4a 4b, we can clearly see that the deviation of device types follows the similar pattern for each device types. TX1 shows more consistent result in case CPU dynamism but was poorer in for SI.

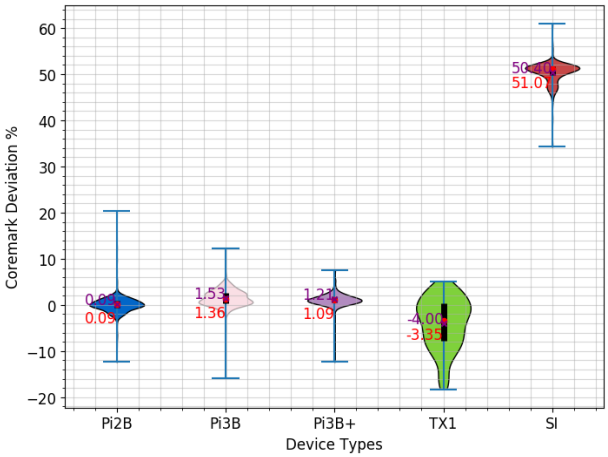
2) Failure and Recovery: Another dynamism application we have designed is to simulate the failure and recovery of devices. The Failure of devices is simulated using MTTF = 600 seconds (Mean-time-to-failure) and recover based on MTTR = 300 seconds (Mean-time-to-recover). We ran the experiment for D400 deployment for 2 hours and results are shown in Figs. 5. The Figure shows the scatter plot for all devices location on the plot of MTTF and MTTR as the axis. It also has the violin plot for the same points along with it. The dynamism is triggered in every 30 seconds (control interval) to check for each device that it fails or not. It is expected that across time for all devices and across devices per interval both gives approximate MTTF and MTTR, where observed $MTTF = \frac{\Sigma DeviceUpTime}{\#DeviceFailures}$ and observed $MTTR = \frac{\Sigma DeviceDownTime}{\#DeviceRestarts}$.

For all devices across time shown, we have the median and mean MTTF for D400 in fig. 5a as 622 and 691 seconds. Similarly for MTTR, we have median and mean for D400 in fig. 5a as 304 and 326 seconds.

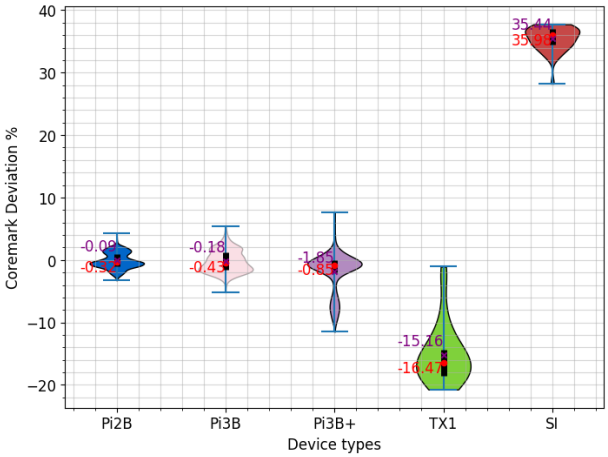
For all devices per interval of 600 seconds (equivalent to MTTF duration), we have median and mean MTTF for D400 in fig. 5b as 622 and 691 seconds. Similarly for MTTR, we have median and mean MTTR for D400 in fig. 5b as 313 and 313 seconds.

VII. Conclusion and Future Work

In this report, we have first formulated a problem that can be used to represent an IoT data exchange in smart communities. Then, we define the system and security requirements that are necessary for IoT data exchange. And then, we have proposed our system design using blockchain. We are working on

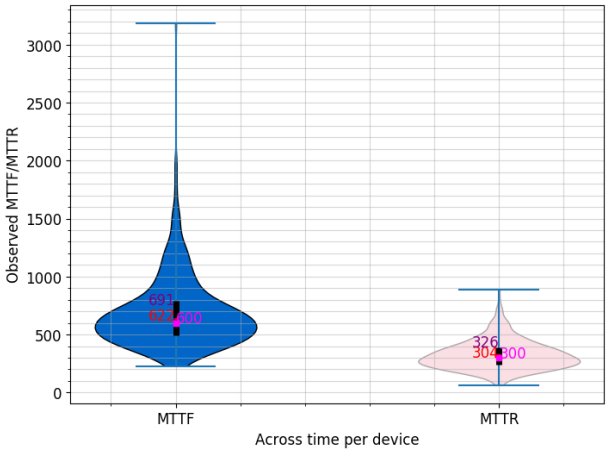


(a) D400: with CPU dynamism

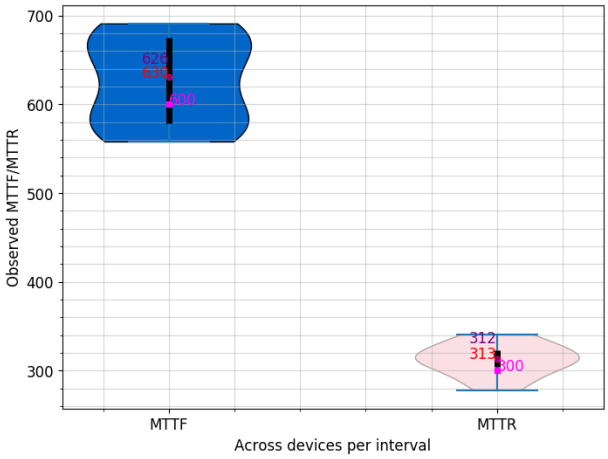


(b) D400: without CPU Dynamism

Fig. 4: Violin plot of *CoreMark Deviation%* of device types for CPU Dynamism.



(a) D400: MTTF/MTTR for All Devices



(b) D400: MTTF/MTTR per interval

Fig. 5: Scatter and Violin plots for MTTF/MTTR dynamism

implementing a prototype based on smart community use case that our solution can answer the problems associated with IoT data exchange. In our previous work, we have built a Large-scale Virtual Environment for Internet of Things (VIoLET) which will be used to prototype our solution.

References

[1] F. Leccese, M. Cagnetti, and D. Trinca, "A smart city application: A fully controlled street lighting isle based on raspberry-pi card, a zigbee sensor network and wimax," *Sensors*, vol. 14, no. 12, pp. 24 408–24 424, 2014.

[2] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Sensing as a service model for smart cities supported by internet of things," *Trans. Emerg. Telecommun. Technol.*, vol. 25, no. 1, pp. 81–93, Jan. 2014. [Online]. Available: http://dx.doi.org/10.1002/ett.2704

[3] A. Reyna, C. Martn, J. Chen, E. Soler, and M. Daz, "On blockchain and its integration with iot. challenges and opportunities," *Future Generation Computer Systems*, vol. 88, pp. 173 – 190, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17329205

[4] P. De Filippi and S. McCarthy, "Cloud computing: Centralization and data sovereignty," *European Journal of Law and Technology*, vol. 3, no. 2, 2012.

[5] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Bitcoin white paper*, 2008.

[6] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[7] G. Greenspan, "Multichain private blockchainwhite paper," *URl: http://www. multichain. com/download/MultiChain-White-Paper. pdf*, 2015.

[8] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: ACM, 2018, pp. 30:1–30:15. [Online]. Available: http://doi.acm.org/10.1145/3190508.3190538

[9] R. G. Brown, J. Carlyle, I. Grigg, and M. Hearn, "Corda: An introduction," *R3 CEV, August*, 2016.

[10] K. Miura and M. agar, "Data marketplace for internet of things," in *2016 International Conference on Smart Systems and Technologies (SST)*, Oct 2016, pp. 255–260.

[11] B. Krishnamachari, J. Power, S. H. Kim, and C. Shahabi, "I3: An iot marketplace for smart communities," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2018, pp. 498–499.

[12] R. Neisse, G. Steri, and I. Nai-Fovino, "A blockchain-based approach for data accountability and provenance tracking," in *Proceedings of the*

12th International Conference on Availability, Reliability and Security, ser. ARES '17. New York, NY, USA: ACM, 2017, pp. 14:1–14:10. [Online]. Available: http://doi.acm.org/10.1145/3098954.3098958

[13] A. Ouaddah, A. Abou Elkalam, and A. Ait Ouahman, "Fairaccess: a new blockchain-based access control framework for the internet of things," *Security and Communication Networks*, vol. 9, no. 18, pp. 5943–5964, 2016.

[14] R. Di Pietro, X. Salleras, M. Signorini, and E. Waisbard, "A blockchain-based trust system for the internet of things," in *Proceedings of the 23rd ACM on Symposium on Access Control Models and Technologies*, ser. SACMAT '18. New York, NY, USA: ACM, 2018, pp. 77–83. [Online]. Available: http://doi.acm.org/10.1145/3205977.3205993

[15] J. Sun, J. Yan, and K. Z. K. Zhang, "Blockchain-based sharing services: What blockchain technology can contribute to smart cities," *Financial Innovation*, vol. 2, no. 1, p. 26, Dec 2016. [Online]. Available: https://doi.org/10.1186/s40854-016-0040-y

[16] P. Missier, S. Bajoudah, A. Capossele, A. Gaglione, and M. Nati, "Mind my value: A decentralized infrastructure for fair and trusted iot data trading," in *Proceedings of the Seventh International Conference on the Internet of Things*, ser. IoT '17. New York, NY, USA: ACM, 2017, pp. 15:1–15:8. [Online]. Available: http://doi.acm.org/10.1145/3131542.3131564

[17] A. Dorri, S. S. Kanhere, R. Jurdak, and P. Gauravaram, "Lsb: A lightweight scalable blockchain for iot security and privacy," *arXiv preprint arXiv:1712.02969*, 2017.

[18] ——, "Blockchain for iot security and privacy: The case study of a smart home," in *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, March 2017, pp. 618–623.

[19] M. Pustišek and A. Kos, "Approaches to front-end iot application development for the ethereum blockchain," *Procedia Computer Science*, vol. 129, pp. 410–419, 2018.

[20] M. Conoscenti, A. Vetro, and J. C. De Martin, "Blockchain for the internet of things: A systematic literature review," in *Computer Systems and Applications (AICCSA), 2016 IEEE/ACS 13th International Conference of*. IEEE, 2016, pp. 1–6.

[21] E. F. Jesus, V. R. Chicarino, C. V. de Albuquerque, and A. A. d. A. Rocha, "A survey of how to use blockchain to secure internet of things and the stalker attack," *Security and Communication Networks*, vol. 2018, 2018.

[22] S. Badiger, S. Baheti, and Y. Simmhan, "Violet: A large-scale virtual environment for internet of things," in *Euro-Par 2018: Parallel Processing*, M. Aldinucci, L. Padovani, and M. Torquati, Eds. Cham: Springer International Publishing, 2018, pp. 309–324.

VIoLET: A Large-scale Virtual Environment for Internet of Things

Shreyas Badiger, Shrey Baheti and Yogesh Simmhan

Indian Institute of Science, Bangalore India
shreyasb@IISc.ac.in, shreybaheti@IISc.ac.in, simmhan@IISc.ac.in

Abstract. IoT deployments have been growing manifold, encompassing sensors, networks, edge, fog and cloud resources. Despite the intense interest from researchers and practitioners, most do not have access to large-scale IoT testbeds for validation. Simulation environments that allow analytical modeling are a poor substitute for evaluating software platforms or application workloads in realistic computing environments. Here, we propose VIoLET, a virtual environment for defining and launching large-scale IoT deployments within cloud VMs. It offers a declarative model to specify container-based compute resources that match the performance of the native edge, fog and cloud devices using Docker. These can be inter-connected by complex topologies on which private/public networks, and bandwidth and latency rules are enforced. Users can configure synthetic sensors for data generation on these devices as well. We validate VIoLET for deployments with > 400 devices and > 1500 device-cores, and show that the virtual IoT environment closely matches the expected compute and network performance at modest costs. This fills an important gap between IoT simulators and real deployments.

1 Introduction

Internet of Things (IoT) is expanding rapidly as diverse domains deploy sensors, communication, and gateway infrastructure to support applications such as smart cities, personalized health, and autonomous vehicles. IoT is also accelerating the need for, and the use of edge, fog and cloud resources, in a coordinated manner. The need comes from the availability of large volumes of data streams that need to be analyzed closer to the edge to conserve bandwidth (e.g., video surveillance), or of fast data streams that need to be processed with low latency [16]. Edge gateway devices such as Raspberry Pi and Smart Phones have non-trivial resource capabilities, and can run a full Linux stack on 64-bit ARM processors. Fog devices such as NVidia's TX1 and Dell's Edge Gateways have power-efficient Atom processors or GPUs to support the needs of several edge devices [3, 19]. At the same time, edge and even accelerated fog devices may not have the elastic and seemingly infinite on-demand resource capacity that is available in the cloud, and necessary for processing by certain IoT applications.

Besides production deployments of IoT, there is also active research at the intersection of IoT, and edge, fog and cloud computing that is investigating application scheduling, resiliency, big data platforms, and so on [8, 9]. However,

a key gap that exists is the ability to validate these research outcomes on real or realistic IoT environments. Research IoT testbeds may have just 10's of devices, and simulation environments make too many idealized assumptions and do not allow actual applications to be deployed. Manually launching and configuring containers is time consuming and error-prone. Even planning of production deployment of IoT, edge and fog resources are based on analytical models or simulations, which may not hold in practice [11,14,18].

What is lacking is a virtualized IoT environment that offers the computing and network ecosystem of a real deployment without the need to purchase, configure and deploy the edge, fog and networking devices. Here, we propose *VIoLET, a Large-scale Virtual Environment for Internet of Things*. VIoLET offers several essential features that make it valuable for researchers and planners. It is a virtualized environment that uses containers to offer comparable compute resources as edge, fog and cloud, and can run real applications. It allows the easy definition of diverse network topologies, and imposes bandwidth and latency limits between containers. VIoLET also allows the definition of virtual sensors that generate data with various distributions within the containers. It runs on top of cloud VMs or commodity clusters, allowing it to scale to hundreds or thousands of devices, provided cumulative compute capacity is available on the host machines. All of these help setup and validate an environment that mimics the behavior of city-scale IoT deployments in a fast, reproducible and cost-effective manner. *VIoLET v1.0* is available for download from `https://github.com/dream-lab/VIoLET`.

The rest of this paper is organized as follows. We motivate various requirements for VIoLET in Section 2, describe its architecture design that meets these requirements and its implementation in Section 3, present results on deploying and scaling VIoLET for different IoT topologies in Section 4, compare it with related literature and tools in Section 5, and finally present our conclusions and future work in Section 6.

2 Design Requirements

Here, we present high-level requirements for a *Virtual Environment (VE)* like VIoLET, based on the needs of researchers and developers of applications, platforms and runtime environments for IoT, edge, and fog resources.

Compute environment. The VE should provide the ability to configure computing resources that capture the performance behavior of *heterogeneous IoT resources*, such as edge devices, gateways, fog and even cloud resources. Key resource capabilities to be controlled include CPU rating, memory and storage capacity, and network. Further, a *compute environment* that can host platforms and run applications should be provided within these resources. Virtual Machines (VM) have traditionally offered such capabilities, but are too heavy-weight for the often light-weight and plentiful IoT devices. *Containers* are much more light-weight and offer similar capabilities. One downside is the inability to change the

underlying Operating System (OS) as it is coupled with the Linux kernel of the host machine. However, we expect most IoT devices to run a flavor of Linux.

Networking. Communication is central to IoT, and the networking layer is sensitive to various deployment limitations on the field. Wired, wireless and cellular networks are common, each with different *bandwidth and latency characteristics*. There is also a distinction between *local and wide area networks*, and *public and private networks* – the latter can limit the visibility of devices to each other. These affect the platforms and applications in the computing environment, and can decide who can connect to whom and if an indirection service is required. The VE needs to capture such diverse network topologies and behavior.

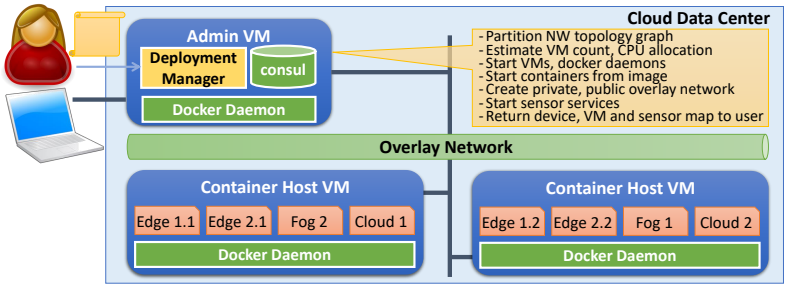
Sensing and Data Streams. Sensors (and actuators) form the third vital component of IoT. These are often connected to the edge computing devices by physical links, *ad hoc* wireless networks, or even on-board the device. These form the source of the distributed, fast data streams that are intrinsic to IoT deployments. The VE should provide the ability to simulate the generation of sensor event streams with various sampling rates and distributions at the compute devices for consumption by hosted applications.

Application Environment. IoT devices often ship with standard platforms and software pre-loaded so that potentially hundreds of devices do not have to be reconfigured across the wide area network. The VE should allow platforms and application environments to be pre-configured as part of the deployment, and the setup to be ready-to-use. Users should not be forced to individually configure each compute resources, though they should have the ability to do so if required.

Scalable. IoT deployments can be large in the number of devices and sensors – ranging in the 1000's – and with complex network topologies. A VE should be able to scale to such large deployments with minimal resource and human overheads. At the same time, these devices offer real computing environments that require underlying compute capacities to be available on the host machine(s). Hence, we require the VE to *weakly scale*, as long as the underlying infrastructure provides adequate cumulative compute and network capacity for all the devices. The use of elastic cloud resources as the host can enable this.

Reproducible. Simulators offer accurate reproducibility but limit the realism, or the ability to run real applications. Physical deployments are hard to get access to and may suffer from transient variability that affects reproducibility. A VE should offer a balance between running within a realistic deployment while being reproducible at a later point in time. This also allows easy sharing of deployment recipes for accurate comparisons.

Cost effective. Clouds are able to offer a lower cost per compute unit due to economies of scale at data centers. But IoT devices while being commodity devices are costlier to purchase, deploy and manage. Having VEs offer comparable resource performance as the IoT deployment but for cheaper compute costs is essential. They should also make efficient use of the pay-as-you-go resources. Further, they should be deployable on-demand on elastic resources and release those resources after the experiments and validations are done.



(a) Architecture Design

(b) JSON describing devices, sensors, VE deployment and host VMs.

Fig. 1. VIoLET Architecture and deployment documents

Ease of Design and Deployment. Users should be able to configure large IoT deployments with ease, and have them deploy automatically and rapidly. It should be possible to mimic realistic real-world topologies or generate synthetic ones for testing purposes.

3 Architecture

We give the high-level overview architecture of VIoLET first, and then discuss individual components and design decisions subsequently. Fig. 1a shows the high-level architecture of our framework. Users provide their IoT VE as *JSON deployment documents* (Fig. 1b) that declaratively capture their requirements. A `devices.json` document lists the devices, their types (e.g., Raspberry Pi 3B, NVidia TX1) and their CPU performance. Another, `sensors.json` document list the virtual sensors and their configurations available. Lastly, the actual deployment document, `deployment.json` lists the number of devices of various types, the network topology of the device inter-connects, including bandwidths and latencies, and optionally the virtual sensors and applications available on each device.

VIoLET takes these documents and determines the number of cloud VMs of a specified type that are required to host containers with resources equivalent to the device types. It also decides the mapping from devices to VMs while meeting the compute capacity, and network bandwidth and latency needs of the topology, relative to what is made available by the host VMs.

Then, containers are configured and launched for each device using *Docker*, and the containers are inter-connected through an overlay network. This allows different private and public networks to be created in the VE. Further, Traffic Control (TC) and Network Address Translation (NAT) rules are set in each container to ensure that the requested network topology, bandwidth and latency limits are enforced.

Virtual sensors, if specified, are then started on each device and their streams available on a local network port in the container. Application environments or startup scripts if specified are also configured or launched. After this, the user is provided with a mapping from the logical device names in their deployment document to the physical device IPs of the matching container, and the VMs on which the containers are placed on. Users can access these devices using the Docker `exec` command. Further, the port numbers at which various logical sensors streams are available on each device is also reported back to the user. Together, these give access to the deployed runtime environment to the user.

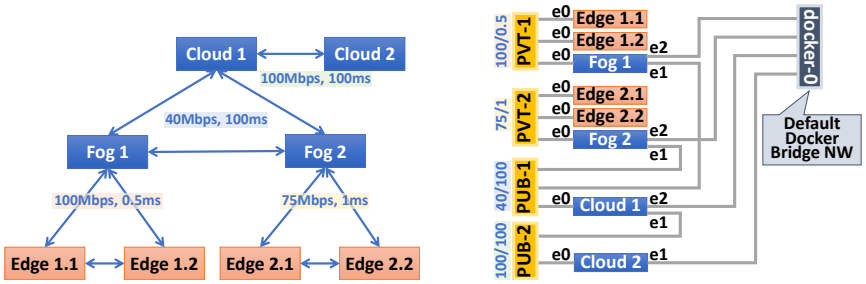
3.1 Compute Resources

Containers are emerging as a light-weight alternative to VMs for multi-tenancy within a single host. They use Linux kernel's `cgroups` feature to offer benefits of custom software environment (beyond the OS kernel) and resource allocation and isolation, while having trivial overheads compared to hypervisors. They are well-suited for fine-grained resource partitioning and software sand-boxing among trusted applications.

Computing devices in VIoLET are modeled as containers and managed using the *Docker* automation framework. There are two parts to this: the *resource allocation* and the *software configuration*. Docker allows containers to have resource constraints to be specified [1]. We use this to limit a container's capacity to match the CPU and Memory available on the native device. We use CPU benchmarks on the native device and the host machine to decide this allocation. The commonly used *CoreMark* [2] is currently supported for an integer-based workload, while *Whetstone* [3] has been attempted for floating-point operations. One subtlety is that while we use the multi-core benchmark rating of the device for the CPU scaling, this may map to fewer (faster) cores of the host machine.

A container's software environment is defined by the user as an image script (`Dockerfile`) that specify details like applications, startup services, and environment variables, and allow modular extensibility from other images. Public

[1] Docker Resource Constraints, docs.docker.com/config/containers/resource_constraints
[2] y, Embedded Microprocessor Benchmark Consortium (EEMBC), coremark.org
[3] Whetstone Benchmark History and Results, roylongbottom.org.uk/whetstone.htm



(a) Sample Topology Description (b) Bridges in Overlay to Achieve Topology

Fig. 2. Network Topology and Docker Overlay Network

Docker repositories have existing images for common IoT platforms and applications (e.g., Eclipse Californium CoAP, Microsoft IoT Edge, RabbitMQ, Spark). VIoLET provides a base image that includes its framework configuration and allow users to extend their device images from this base with custom software configuration. This is similar to specifying a VM image, except that the users are limited to the host device's Linux kernel OS [4]. Hence, defining a compute device in VIoLET requires associating a device type for resources, and a device image for the software environment.

3.2 Network Topology

Users define the network topology for the devices based on three aspects: the public network or a private network the device is part of; the visibility of devices to each other as enforced by firewalls; and the bandwidth and latency between pairs of devices. IoT networks are usually composed of numerous private networks that interface with each other and the public Internet through gateways. We allow users to define logical private networks and assign devices to them. These exist in their own subnet. Each private network has a gateway device defined, and all traffic to the public network from other devices is routed through it. All gateway devices are part of one or more public networks, along with other devices that are on those public networks.

For simplicity, all devices in a private network by default can access each other, and have a common latency and bandwidth specified between pairs of devices by the user; and similarly for all devices connected to a public network. By default, devices on different public networks can reach each other. However, users can override this visibility between any pair of devices, and this is directional, i.e., $D1 \rightarrow D2$ need not imply $D1 \leftarrow D2$.

[4] Docker recently introduced support for Windows and Linux containers hosted on Windows Server using the Hyper-V hypervisor. But this is more heavy-weight than Linux containers, and not used by us currently.

We implement the bandwidth and latency between devices using *Traffic Control (TC) rules* offered by Linux's `iproute2` utility, and the `network` service that we start on each container using `systemd` [5]. Here, every unique bandwidth and latency requirement gets mapped to a unique virtual Ethernet port, and the rules and enforced on it. This Ethernet port is also connected to the bridge corresponding to the (private or public) network that the device belongs to. The bridges physically group devices that are on the same network, and also logically assign a shared bandwidth and latency to them. All devices on public networks are also connected to a common `docker-0` bridge for the VM they are present on, and which allows all to all communication by default. Restricting the routing of traffic in a private network to/from the public network only through its gateway device is enacted through `ip` commands and *Network Address Translation (NAT) rules*. These rules redirect packets from the Ethernet port connected to the private network, to the Ethernet port connected to the public network.

Docker makes it easy to define connectivity rules and *IP addressing* of containers present in a single host machine using custom bridges defined on the Docker daemon running on the host. However, devices in VIoLET can be placed on disparate VMs and still be part of the same private network. Such communication between multiple Docker daemons requires custom Docker *overlay networks*. We create a standalone *Docker Swarm pool* which gives us the flexibility to set network and system parameters [6]. For this, the host machines must be able to access a shared key-value store that maintains the overlay networking information. In VIoLET, we use the *Consul* discovery service as our key-value store that is hosted in a separate container on an *admin VM*.

E.g., Fig. 2 shows a sample network topology, and the Ethernet ports and bridges to enact this in VIoLET. Here, the edge devices E1.1, E1.2 form a private network PVT-1 with the fog device F1 as a gateway, and likewise E2.1, E2.2 and F2 form another private network, PVT-2. Each device can have sensors enabled to simulate data streams with different distributions. The bandwidth and latency within these private networks is uniform: 100Mbps/0.5ms for PVT-1, and 75Mbps/1ms for PVT-2. F1 and F2 fog devices go on to form a public network PUB-1 along with the cloud device, C1, with 40Mbps/100ms. Similarly, the two cloud devices form another public network PUB-2, with 100Mbps/100ms. All these devices are on a single VM, and the public devices are also connected to the `docker-0` bridge for that VM. While the edge devices are connected to a single overlay network, the fog and cloud devices can be connected to multiple overlay networks, based on bandwidth and latency requirements.

As can be seen, configuring the required network topology is complex and time consuming – if done manually for each IoT deployment. Having a simple declarative document that captures the common network patterns in IoT deployments helps automate this.

[5] Traffic Control in Linux, tldp.org/HOWTO/Traffic-Control-HOWTO

[6] Multi-host networking with standalone swarms, docs.docker.com/network/overlay-standalone.swarm

3.3 Sensors and Virtual Observation Streams

Edge devices are frequently used to acquire IoT sensor data over hardware interfaces like serial, UART or I2C, and then make them available for applications to process and/or transfer. Experiments and validation of IoT deployments require access to such large-scale sensor data. To enable this, we allow users to define virtual sensors that are collocated with devices. These virtual sensors simulate the generation of sensed events and make them available at a local network port, which acts as a proxy for a hardware interface to the sensor. Applications can connect to this port, read observations and process them as required.

We support various configuration parameters for these sensors. The values for the sensor measurements themselves may be provided either as a text file with real data collected from the field, or as the properties of a statistical distribution, such as uniform random, Gaussian, and Poisson from which we sample and return synthetic values. In addition, the rate at which these values change or the events are generated is also specified by the user. Here too we can specify real relative timestamp or a distribution.

We implement each sensor as a Python script that is launched as part of the container startup. The script starts a *Flask* application server that listens on a local port. It takes the sensor's parameters, and internally starts generating observations corresponding to that. When a client connects to this port and requests a measurement, the service returns the *current* reading. For simplicity, this is reported as a CSV string consisting of a user-defined logical sensor ID, the observation timestamp and a sensed value, but can be easily modified.

3.4 Resource Mapping and Deployment

The *admin VM* runs a service that receives the user's deployment document as a REST request and enacts the deployment on cloud VMs in that data center. The default resource hosts are Amazon EC2 VMs but this can easily be extended to resources on other cloud providers or even a private cluster. All AWS EC2 VM instances belong to a same Virtual Private Cloud (VPC) and the same subnet. On receipt of the deployment request, VIoLET builds a graph of the network topology that is used to deploy the devices onto host resources. Here, the vertices of the graph are the devices and are labeled with the device's CPU requirement, given in the CPU benchmark metrics, e.g., iterations/sec for *CoreMark*, and MWIPS for *Whetstone*. An edge exists if a source device can connect to a sink device, and this is labeled by the bandwidth and latency for that network link. E.g., a private network where all devices can see each other will form a clique.

We then make a gross estimate of the number of underlying resources we require. This is done by adding the vertex weights, dividing by the benchmark metric for the host (cloud VM) and rounding it up. This is the least number of identical host resources, say n, needed to meet the compute needs of all devices.

Then, we partition the graph across these n hosts using `gpmetis` such that the vertex weights are balanced across hosts and the sum of edge cuts between hosts, based on device bandwidths, is minimized. This tries to collocate devices

Table 1. Device Perf., Device Counts and Host VM Counts used in Deployments

	Deployment→		**D105**		**D408**	
Device	Cores	CMark	Count	∑CMark (k)	Count	∑CMark (k)
Pi 2B	4	8,910	50	445	200	1,782
Pi 3B	4	13,717	50	685	200	2,743
NVidia TX1	4	26,371	4	105	7	184
Softiron	8	76,223	1	76	1	76
		Total		**1,311**		**4,786**
m4.10XL *(host)*	40	371,384	4	1,485	13	4,827

with high bandwidth inter-connects on the same host. We then check if the sum of the bandwidth edge cuts between devices in each pair of hosts is less than the available bandwidth capacity between them, and if the sum of benchmark metrics of all devices in a host is smaller than its capacity. If not, we increment n by 1 and repeat the partitioning, and so on.

This greedy approach provides the least number of host resources and the mapping that will meet the CPU and bandwidth capacities of the deployment. For now, we do not optimize for memory capacity and latency, but these can be extended based on standard multi-parameter optimization techniques.

4 Evaluation

We evaluate VIoLET for two different IoT deployment configurations: **D105** with 105 edge and fog devices, and **D408** with 408 edge and fog devices. The configuration of each of the devices, their CoreMark CPU performance and the deployment counts are shown in Table 1, along with the number of AWS VMs required to support them. CoreMark v1.0 is run with multi-threading enabled.

We use two generations of *Raspberry Pis* as edge devices – *Pi 2B* with $4 \times$ 900 MHz ARM32 cores and *Pi 3B* with 4×1.2 GHz ARM64 cores, and 1 GB RAM each. In addition, we have two fog resources – a *Softiron 3000 (SI)* with AMD A1100 CPU with 8×2 GHz ARM64 cores and 16 GB RAM, and an *NVidia TX1* device with 4×1.7 GHz ARM64 cores and 4 GB RAM (its GPU is not exposed). We use Amazon AWS *m4.10XL* VMs that have 40×2.4 GHz Intel Xeon E5-2676 cores, 160 GB RAM and 10 Gbps network bandwidth as the host. Each costs US$2.00/hour in the US-East (Ohio) data center. As we see, the D105 deployment with 424 ARM cores requires 3 of these VMs with 120 Xeon cores, and D408 with $1,636$ ARM cores requires 13 of these VMs with 390 Xeon cores. These deployments cost about US$6/hour and US$26/hour, respectively – these are cheaper than a single Raspberry Pi device, on an hourly basis.

4.1 Results for D105 and D408

The network topology for these two deployments is generated synthetically. D105 is defined with 5 private networks and 4 public networks, while D408 has 8

Table 2. Configuration of private and public networks in D105, and Deviation% between Observed and Expected Bandwidth and Latency per network.

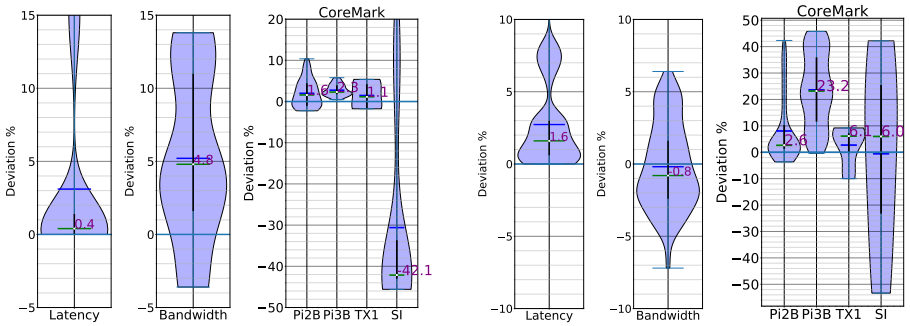
Network	Expected		Obs. BW Dev.%		Obs. Lat. Dev.%	
	BW (Mbps)	Lat. (ms)	Median	Mean	Median	Mean
PVT-1	5	25	11.0	11.0	0.6	0.5
PVT-2	5	75	13.8	13.8	0.0	0.0
PVT-3	25	1	4.8	4.8	15.0	15.5
PVT-4	25	25	4.0	3.7	1.0	1.1
PVT-5	25	50	1.6	1.4	0.0	0.0
PUB-1	25	75	-3.6	-3.6	0.0	0.0
PUB-2	25	75	-3.6	-3.6	0.0	0.0
PUB-3	25	75	-3.6	-3.5	0.0	0.0
PUB-4	25	75	-3.6	-3.6	0.0	0.0

Table 3. Configuration of private and public networks in D408, and Deviation% between Observed and Expected Bandwidth and Latency per network.

Network	Expected		Obs. BW Dev.%		Obs. Lat. Dev.%	
	BW (Mbps)	Lat. (ms)	Median	Mean	Median	Mean
PVT-1	100	5	-2.6	-2.4	6.0	5.2
PVT-2	75	5	-1.1	-1.3	3.0	4.9
PVT-3	75	25	-4.1	-4.0	0.6	1.0
PVT-4	50	5	0.0	0.1	4.0	4.9
PVT-5	50	25	-1.8	-2.0	0.6	0.8
PVT-6	25	25	-1.8	-2.0	0.6	0.8
PVT-7	25	5	2.8	3.2	0.6	0.8
PVT-8	25	50	4.8	5.0	0.6	0.8
PUB-1	25	75	-3.6	-3.6	0.0	0.0
PUB-2	25	100	-7.0	-7.0	0.0	0.0

private networks and 2 public networks. A fog device serves as the gateway in each private network, and we randomly place an equal number of edge devices in each private network. Their respective network configurations are given in Tables 2 and 3. Each network has a fixed bandwidth and latency configuration, and this ranges from $5-100$ Mbps bandwidth, and $1-100$ ms latency, as specified. All devices in the public networks can see each other. Edge devices in the private network can access the public network, routed through their gateway, but devices in the public network cannot access the devices in the private network. It takes about 8 mins and 24 mins to launch these two topologies on VIoLET.

Once deployed, we run four baseline benchmarks to validate them. The first does `fping` between $2n$ pairs of devices in each private and public network, where n is the number of devices in the network, and measures the observed latency on the defined links. Next, we sample a subset of $\frac{n}{2}$ links in each private and public network and run `iperf` on them to measure the observed bandwidth. Since `iperf` is costlier than `fping`, we limit ourselves to fewer samples. Third, we run `traceroute` to verify if the gateway device configured for each device



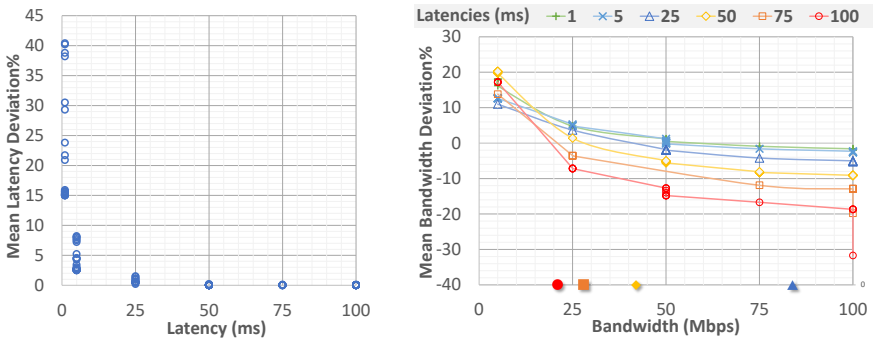
(a) D105 Deployment (b) D408 Deployment

Fig. 3. Violin plot of *deviation%* for network latency, bandwidth and CoreMark CPU.

matches the gateway of the private network, as a sanity check. These network sanity checks take ≈ 3 *mins* per network for D105, and runs in parallel for all networks. Lastly, we run multi-core CoreMark concurrently on all devices.

Figs. 3a and 3b show a violin plot of the *deviation%* of the observed network latency, bandwidth, and CoreMark performance from the expected metrics for the two deployments, where $deviation\% = \frac{(Observed - Expected)}{Expected}\%$. The median value is noted in purple text. We see that the median latency and bandwidth deviation% are within $\pm 5\%$ for both the D105 and D408 deployments, with latency of 0.4% and 1.6%, and bandwidth of 4.8% and -0.8%, respectively. This is within the margin of error for even real-world networks. The entire distribution in all these cases does not vary by more than 15%, showing a relatively tight grouping given the number of devices and VMs. We analyze these further for diverse network configurations in the next section.

We run the CoreMark CPU benchmark on all the devices concurrently and report the violin plot for the deviation% for each of the 4 device types. The median CoreMark value for each device is included in the violin, except for the SI fog where we report values from all the trials since there is just one such device in each deployment. We see that for the two Pis and TX1 – the three slowest devices – the median CoreMark deviation% is within $\pm 2.5\%$ for D105, and the most deviation is $+10\%$ for Pi2B. These indicate that the observed performance is marginally higher than expected, and there is little negative deviation for these three devices. However, we see that the single SI fog device, which is the largest device, has a median deviation% of -42.1% from 40 trials of CoreMark that were run on it. The distribution is also wide, ranging from -45% to $+21\%$. This indicates that the concurrent multi-threaded CoreMark runs on 10's of containers on the same VM is causing the largest device container to have variable performance. In fact, the sum of the observed CoreMarks for all the deployed devices in D105 is $1,319k$, which is close to the sum of the expected CoreMark from the devices of $1,311k$. So the small over-performance of many small devices is causing the under-performance of the large device. D408 shows a



(a) Latency

(b) Bandwidth, at different Latencies. Bottom row shows ideal bandwidth for latency.

Fig. 4. Variation of *deviation%* for different latency and bandwidth configurations.

different behavior, with Pi3B showing higher positive deviations, with a median of 23.2%, while the other devices show a smaller positive deviation of 2.6–6%. SI however does show a wider distribution of the deviation% as before.

Besides these baseline network and CPU metrics, we also run two types of application workloads. One of them starts either an MQTT publisher or a subscriber on each device, and each connects to an *Eclipse Mosquitto MQTT broker* on its gateway. A publisher samples observations from a local sensor and publishes it to a unique topic at its gateway broker while a subscriber subscribes to it. This tests the network and process behavior for the common pub-sub pattern seen in IoT. While results are not plotted due to lack of space, we observe that the median end-to-end latency for each message is $\approx 50\ ms$, which loosely corresponds to the two network hops required from the publisher to the broker, and broker to subscriber.

Another workload that we evaluate is with the ECHO dataflow platform for edge and cloud [15]. Here, we incrementally launch 100 Extract-Transform-Load dataflows using the Apache NiFi engine on distributed devices and observe the latency time for deployment and the end to end latency for the dataflows. This is yet another use-case for VIoLET to help evaluate the efficacy of such edge, fog and cloud orchestration platforms and schedulers.

4.2 Analysis of Network Behavior

Being able to accurately model network behavior is essential for IoT VEs. Here, we perform more detailed experiments that evaluate the impact of specific bandwidth and latency values on the deviation%. Specifically, we try out 19 different network configurations of the D105 deployment while varying the pair of bandwidth and latency values on these networks. These together form 143 different networks. In Fig. 4b, we plot the deviation% of the mean bandwidth, as the

bandwidth increases for different latency values, while in Fig. 4a, we plot the deviation% of the mean latency, as latency increases.

It is clear from Fig. 4a that the latency deviation is sensitive to the absolute latency value. For small latency values of 1 ms, the deviation% ranges between $15 - 40\%$, and this drops to $2.6 - 8\%$ for 5 ms. The deviation% exponentially reduces for latencies higher than that, with latencies over 50 ms having just 0.1% deviation. The latency between VMs is measured at 0.4 ms, while between containers on the same VM is 0.06 ms. Hence, achieving a latency better these is not possible, and the achieved latency depends on the placement of containers on the same or different VMs. Since our network partitioning currently is based on bandwidth and compute capacity, and not latency limits, it is possible that two devices requiring low latency are on different VMs. As a result, the deviation% increases. Here, we see that the latency deviation is independent of the bandwidth of the network link.

We observe that the deviation in bandwidth is a function of both latency and bandwidth. In fact, it is also a function of the *TCP window size*, which by default is set to $262,144$ bytes in the containers. The *Bandwidth Delay Product (BDP)* is defined as the product of the bandwidth and latency. For efficient use of the network link, the TCP window size should be greater than this BDP, i.e., $Window \geq Bandwidth \times Latency$. In other words, given a fixed latency and TCP window size, the $Peak\ Bandwidth = \frac{Window}{Latency}$.

Fig. 4b shows the bandwidth deviation% on the Y axis for different latencies, as the bandwidth increases on the X axis. It also shows the maximum possible bandwidth for a given latency (based on the window size) along the bottom X axis. We observe that for low latencies of $1 - 25$ ms, the bandwidth deviation% is low and falls between $-5.1 - 18\%$ for all bandwidths from $5 - 100$ Mbps. This is because with the default window size, even a latency of 25 ms supports a bandwidth of 83 Mbps, and lower latencies support an even higher peak bandwidth. The positive deviation% is also high for low bandwidth values and lower for high bandwidth values – even small changes in absolute bandwidth causes a larger change in the relative deviation% when the bandwidth is low.

We also see that as the latency increases, the negative deviation% increases as the bandwidth increases. In particular, as we cross the peak bandwidth value on the X axis, the deviation% becomes more negative. E.g., at 75 ms, the peak bandwidth supported is only 28 Mbps, and we see the bandwidth deviation% for this latency worsen from -3.6% to -11.9% when the bandwidth configuration increases from 25 Mbps to 75 Mbps. This is as expected, and indicates that the users of the container need to tune the TCP window size in the container to enforce bandwidths more accurately.

5 Related Work

The growing interest in IoT and edge/fog computing has given rise to several *simulation environments*. *iFogSim* [11] extends the prior work on CloudSim [5] to simulate the behavior of applications over fog devices, sensors and actuators

that are connected by a network topology. Users define the compute, network and energy profiles of fog devices, and the properties and distributions of tuples from sensors. DAG-based applications with tasks consuming compute capacity and bandwidth can be defined by the user, and its execution over the fog network is simulated using an extensible resource manager. The goal is to evaluate different scheduling strategies synthetically. We similarly let devices, network and sensors to be defined, but actually instantiate the first two – only the sensor stream is simulated. This allows users to evaluate real applications and schedulers.

Edgecloudsim [18] offers similar capabilities, but also introduces mobility models for the edge into the mix. They simulate network characteristics like transmission delay for LAN and WAN, and also task failures due to mobility for a single use-case. *IOTSim*, despite its name, simulates the execution of Map Reduce and stream processing tasks on top of a cloud data center, and uses CloudSim as the base simulation engine. While IoT motivates the synthetic application workloads for their big data platform simulation, they do not actually simulate an IoT deployment.

In the commercial space, city-scale simulators for IoT deployments in smart cities are available [14]. These mimic the behavior of not just devices, sensors, actuators and the network, but also application services like MQTT broker and CoAP services that may be hosted. These offer a comprehensive simulation environment for city-planners to perform what-if analysis on the models. We go a step further and allow realistic devices and networks to be virtualized on elastic cloud VMs, and applications themselves to be executed, without physically deploying the field devices. Simulators are popular in other domains as well, such as cloud, network and SDN simulators [5,12,13].

There have been container-based solutions that are closer to our approach, and allow large-scale customized environments to be launched and applications to be run on them. Ceesay, et al. [6], deploy container-based environments for Big Data platforms and workloads to test different benchmarks, ease deployment and reduce reporting costs. Others have also used such container-based approaches to inject faults into the containers, and evaluate the behavior of platforms and applications running on them [7].

Other have proposed IoT data stream and application workloads for evaluating big data platforms, particularly stream processing ones. Here, the sensor data is simulated at large-scales while maintaining realistic distributions [1,10]. These can be used in place of the synthetic sensor streams that we provide. Our prior work has proposed stream and stream processing application workloads for IoT domains [17]. These can potentially use VIoLET for evaluating execution on edge and fog, besides just cloud resources.

Google's Kubernetes [4] is a multi-node orchestration platform for container life-cycle management. It schedules containers across nodes to balance the load, but is not aware of network topologies that are overlaid on the containers. VIoLET uses a simple graph-partitioning approach for placement of containers on VMs to balance the CPU capacity, as measure by CoreMark, and ensure that the required device bandwidths stay within bandwidth available between the hosts.

6 Conclusions and Future Work

In this paper, we have proposed the design requirements for a Virtual IoT Environment, and presented VIoLET to meet these needs. VIoLET allows users to declaratively create virtual edge, fog and cloud devices as containers that are connected through user-defined network topologies, and can run real IoT platforms and applications. This offers first-hand knowledge of the performance, scalability and metrics for the user's applications or scheduling algorithms, similar to a real IoT deployment, and at large-scales. It is as simple to deploy and run as a simulation environment, balancing ease and flexibility, with realism and reproducibility on-demand. It is also affordable, costing just US$26/hour to simulate over 400 devices on Amazon AWS Cloud. VIoLET serves as an essential tool for IoT researchers to validate their outcomes, and for IoT managers to virtually test various software stacks and network deployment models.

There are several extensions possible to this initial version of VIoLET. One of our limitations is that only devices for which container environments can be launched by Docker are feasible. While any device container that runs a standard Linux kernel using `cgroups` (or even a Windows device [7]) can be run, this limits the use of edge micro-controllers like Arduino, or wireless IoT motes that run real-time OS. Also, leveraging Docker's support for GPUs in future will help users make use of accelerators present in devices like NVidia TX1 [8]. There is also the opportunity to pack containers more efficiently to reduce the cloud costs [2], including over-packing when devices will not be pushed to their full utilization.

Our network configurations focus on the visibility of public and private networks, and the bandwidth and latency of the links. However, it does not yet handle more fine-grained transport characteristics such as collision and packet loss that are present in wireless networks. Introducing variability in bandwidth, latency, link failures, and even CPU dynamism is part of future work. More rigorous evaluation using city-scale models and IoT applications are also planned using large private clusters to evaluate VIoLET's weak scaling.

7 Acknowledgments

This work is supported by research grants from VMWare, MHRD and Cargill, and by cloud credits from Amazon AWS and Microsoft Azure. We also thank other DREAM:Lab members, Aakash Khochare and Abhilash Sharma, for design discussions and assistance with experiments. We also thank the reviewers of Euro-Par for their detailed comments that has helped us improve the quality of this paper.

[7] Docker for Windows, https://docs.docker.com/docker-for-windows/
[8] GPU-enabled Docker Containers, https://github.com/NVIDIA/nvidia-docker

References

1. Arlitt, M., Marwah, M., Bellala, G., Shah, A., Healey, J., Vandiver, B.: Iotabench: an internet of things analytics benchmark. In: International Conference on Performance Engineering (ICPE) (2015)
2. Awada, U., Barker, A.: Improving resource efficiency of container-instance clusters on clouds. In: Cluster, Cloud and Grid Computing (CCGRID) (2017)
3. Bonomi, F., Milito, R., Zhu, J., Addepalli, S.: Fog computing and its role in the internet of things. In: ACM Workshop on Mobile Cloud Computing (MCC) (2012)
4. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, omega, and kubernetes. ACM Queue 14(1) (2016)
5. Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A.F., Buyya, R.: Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Software: Practice and Experience (SPE) 41(1), 23–50 (2011)
6. Ceesay, S., Barker, D., Varghese, D., et al.: Plug and play bench: Simplifying big data benchmarking using containers. In: IEEE International Conference on Big Data (BigData) (2017)
7. Dabrowa, J.: Distributed system fault injection testing with docker. In: JDD (2016)
8. Dastjerdi, A.V., Gupta, H., Calheiros, R.N., Ghosh, S.K., Buyya, R.: Internet of Things: Principles and Paradigms, chap. Fog Computing: Principles, Architectures, and Applications. Morgan Kaufmann (2016)
9. Ghosh, R., Simmhan, Y.: Distributed scheduling of event analytics across edge and cloud. ACM Transactions on Cyber Physical Systems (TCPS) (2018), to Appear
10. Gu, L., Zhou, M., Zhang, Z., Shan, M.C., Zhou, A., Winslett, M.: Chronos: An elastic parallel framework for stream benchmark generation and simulation. In: IEEE International Conference on Data Engineering (ICDE) (2015)
11. Gupta, H., Vahid Dastjerdi, A., Ghosh, S.K., Buyya, R.: iFogSim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. Software: Practice and Experience 47(9), 1275–1296 (2017)
12. Henderson, T.R., Roy, S., Floyd, S., Riley, G.F.: Ns-3 project goals. In: Workshop on Ns-2: The IP Network Simulator (2006)
13. Lantz, B., Heller, B., McKeown, N.: A network in a laptop: Rapid prototyping for software-defined networks. In: Workshop on Hot Topics in Networks (2010)
14. Leland, J.: Deploy scalable smart city architectures confidently with network simulation. Tech. rep., insight tech (2017)
15. Ravindra, P., Khochare, A., Reddy, S.P., Sharma, S., Varshney, P., Simmhan, Y.: Echo: An adaptive orchestration platform for hybrid dataflows across cloud and edge. In: International Conference on Service-Oriented Computing (ICSOC) (2017)
16. Satyanarayanan, M., et al.: Edge analytics in the internet of things. IEEE Pervasive Computing 14(2), 24–31 (2015)
17. Shukla, A., Chaturvedi, S., Simmhan, Y.: RIoTBench: A Real-time IoT Benchmark for Distributed Stream Processing Platforms. Concurrency and Computation: Practice and Experience 29(21) (2017)
18. Sonmez, C., Ozgovde, A., Ersoy, C.: Edgecloudsim: An environment for performance evaluation of edge computing systems. In: Fog and Mobile Edge Computing (FMEC) (2017)
19. Varshney, P., Simmhan, Y.: Demystifying fog computing: Characterizing architectures, applications and abstractions. In: IEEE International Conference on Fog and Edge Computing (ICFEC) (2017)