SPARCS: Stream-Processing Architecture applied in Real-time Cyber-physical Security

Reinhard Gentz, Sean Peisert, Joshua Boverhof, Daniel Gunter

***Abstract*—In this paper, we showcase a complete, end-to-end, fault tolerant, bandwidth and latency optimized architecture for real time utilization of data from multiple sources that allows the collection, transport, storage, processing, and display of both raw data and analytics. This architecture can be applied for a wide variety of applications ranging from automation/control to monitoring and security. We propose a practical, hierarchical design that allows easy addition and reconfiguration of software and hardware components, while utilizing local processing of data at sensor or field site ("fog computing") level to reduce latency and upstream bandwidth requirements. The system supports multiple fail-safe mechanisms to guarantee the delivery of sensor data. We describe the application of this architecture to cyber-physical security (CPS) by supporting security monitoring of an electric distribution grid, through the collection and analysis of distribution-grid level phasor measurement unit (PMU) data, as well as Supervisory Control And Data Acquisition (SCADA) communication in the control area network.**

I. Introduction

Real-time data holds potentially high value for business and control decisions but it also comes with a perishable expiration date. If the value of this data is not realized in a certain window of time, its value is lost and the opportunity for corrective action lost. Such data is acquired continuously and quickly, therefore, we call it streaming data. Data streaming requires prompt attention as sensor readings change rapidly. A blip in log file, sudden price change, or detected system attacks may hold immense value but only if it alerted in time.

Real-time streaming and analysis of multi-source sensor data is an especially challenging task, and efficient architecture designs are complex and difficult to deploy. For our requirements we need a system that has a small bandwidth footprint on the sensor networks, low latency, scale-ability, while allowing (central) human oversight and providing fail-safe operation when systems fail. From the systems perspective, a design challenge is dealing with the *big data* problem of managing the collection of measurements in the field.

In this paper we showcase the architecture that we have developed for our cyber-physical security (CPS) system for the electrical grid. It supports the combined analysis of multiple sensors and sensor types, results are computed as soon as enough sensors are aggregated i.e., *"fog computing"* (or *edge computing*) [1] and *cloud computing*. Due to the fog sensor network's low bandwidth and our computational center's limited resources, our facilities cannot produce real time events given the constraints unless we reduce network usage and computational load. We accomplished this by offloading initial processing and analysis to small inexpensive mini computers that perform sensor readings. The fog networks limited bandwidth is of particular concern in metered or shared wireless connections. For example for the in sensors commonly used wireless standard IEEE 802.15.4 (i.e., used in the Zigbee standard [2]) only has a data-rate of 250kbyte per second that have to be shared by all wireless clients in range of each other (see [3] and references therein). Thus a reduction and prioritization of traffic can be very important.

We also address problems of resiliency by incorporating this data and its analysis in the monitoring systems. For example in our case, to identify hardware failures or potential cyber attacks on the electrical grid. The CPS system we have developed can show early signs of cyber-physical attacks by utilizing the combined data analysis on measurements from both sensors in the physical domain (in our case distribution phasor measurement units (PMUs)) and data from the cyber domain (sniffed supervisory control and data acquisition (SCADA) network traffic), to identify malicious network intrusions and compromised distribution controllers in near-real time.

A. Related Works

Most notable prior work is the implementation of Apache Kafka [4]. The Kaftka ingest and messaging systems follows the same design principles of distributed clusters and publisher subscriber messaging system. Major differences to the proposed work is that data driven compression, and prioritization throughout the pipeline is not supported. Further Amazon Kinesis [5] offers an all in one cloud solution for real time streaming. Most notably it also works only in the cloud. This means that all data has to travel to a central location, i.e., the cloud, and gets processed. Because all data has to be sent to the cloud local processing for latency, reliability and prioritization is not available.

Additional work such as [6] and [7] and references herein are modular and scalable, but again do not support analytical processes at multiple locations and thus cannot prioritize the data accordingly when propagating it through the network, nor can operate locally for fail safe reasons. Other researchers implemented aggregation algorithms that can perform computations with aggregation nodes [8] these systems however miss out on the ability of data driven prioritization as they ignore the information value in the transported data for prioritization purposes. Others [9] ignore the need for authentication/security and are vulnerable by simple man in the middle attacks or unauthenticated/untrusted third party nodes.

B. Contributions

The contributions of this work are as follows:

R. Gentz (rgentz@lbl.gov), S. Peisert, J. Boverhof, D. Gunter are with Lawrence Berkeley National Laboratory.

- A unified, stream-processing architecture
- Local or fog processing of data where possible for improved latency and reliability
- prioritization/compression of the upstream data-flow, and reducing the need of upstream networking and computation resources.
- Increased resilience to equipment or networking failures
- Modular and exchangeable components
- Simple API for processing and sharing historical and live data.
- Easy and secure deployment

II. Overview of the Stream-Processing Architecture for Real-time Cyber-physical Security (SPARCS)

An overview of our architecture is seen in Fig. 1. The

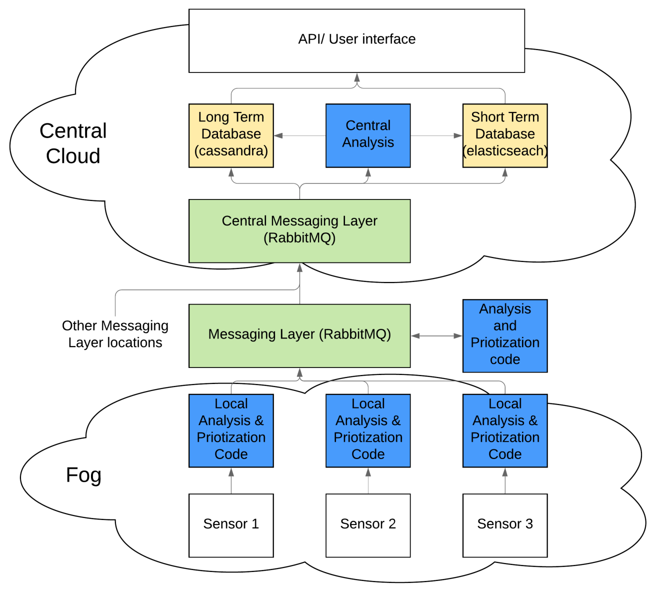
API/ User interface
Central Cloud
Long Term Database (cassandra)
Central Analysis
Short Term Database (elasticseach)
Central Messaging Layer (RabbitMQ)
Other Messaging Layer locations
Messaging Layer (RabbitMQ)
Analysis and Priotization code
Fog
Local Analysis & Priotization Code
Local Analysis & Priotization Code
Local Analysis & Priotization Code
Sensor 1
Sensor 2
Sensor 3

Fig. 1. Stream-Processing Architecture Overview

main novelty of our architecture lies in the application of fog computing/on-site computation where possible. This approach has two major benefits. First the local portion of the data analysis allows for minimal latency to react on local events, even in situations where wide area communications are not available (and could have protected against the attacks described in, for instance [10]). This also includes means of triggering an alert that is propagated to both a local and central alerting system. Second, the paradigm of fog computing also allows prioritization and annotation of upstream data flows when anomalies are detected by local scripts. This is shown in Fig 2, where local analysis performs triage on sensor data and computed results, routing it to a queue based on urgency and other factors. This ensures that important analysis results and data can be handled with priority by the rest of the infrastructure.

In order for the prioritization to work for the remaining system we need to use a high performance messaging system

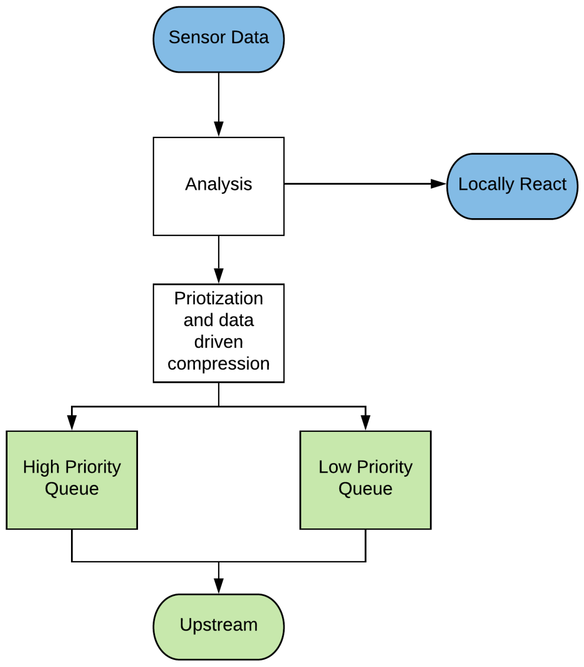

Fig. 2. Prioritization of traffic based on local data analysis

that can handle massive data ingest and allows for queue prioritization. We decided to use the well supported RabbitMQ messaging system [11] for data transport between the components for the following reasons. Data Prioritization is the first and the most important reason for choosing RabbitMQ as messaging system is ability to prioritize individual queues. Secondly, the RabbitMQ messaging system is designed for scalability and can be deployed across a computing cluster. A single queue can be spanned over multiple hosts for scaling and performance advantages. Third, we can deploy RabbitMQ as interconnected decentralized clusters, allowing a layered deployment as seen in Figure 1 Lastly, data fan-out allows each sensor to publish its data once, RabbitMQ handles data replication to multiple recipients. The distribution of the data to all the analysis functions that need this data is handled by the RabbitMQ Cluster and can handle alerting and reacting to alerts (see Fig 3). In addition to data This mechanism

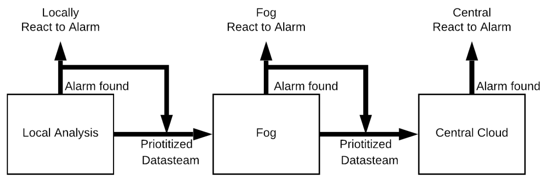

Fig. 3. Alerts can be triggered and reacted to by any stage in the process, and will be propagated upstream.

has the positive side effect of reducing the network load on each sensor and shifting network load to the well-connected RabbitMQ Cluster, reducing the network load on potentially wireless and or metered connections common at the network edge/fog.

In our own system we have deployed one central RabbitMQ cluster and multiple lower level RabbitMQ clusters in a tree like structure. Further we have deployed data-ingesting analytical processes co-located at each RabbitMQ cluster. This ensures that we can execute all analysis scripts with the minimal latency as close to the data source as possible, only going upstream if non-local data is required. This local processing allows us to prioritize the data at each level of aggregation and importantly discard (or heavily compress) uneventfully, uninteresting data, as determined by the analysis algorithm. This discarding of uneventful data significantly reduces the load on upstream computing resources as well as storage requirements for archiving.

A. *Central location*

When data is received by the central RabbitMQ cluster it is processed by the final set of analysis algorithms. In addition to real time streaming data, these algorithms can also utilize the Historian archive. The historian consists both of short and long term databases, is deployed at the central location and allows one to (re)evaluate historical measurements. The Historian is especially useful for newly-developed algorithms or configuration updates that can be tested and evaluated on a substantial corpus of real data. For machine learning applications, this historical data can be used to train the necessary machine learning models. The short-term database is intended to enable rapid searches and complete queries, and stores the (compact) analysis reports from the various subsystems and recent raw data for easy visualization. If the database system is running low on storage space, the oldest raw data is automatically removed, as it is duplicated in the long-term database.

B. *Long-term database*

For the the long-term database we combine multiple sensor samples together in one small package, utilizing using Google's Protocol Buffers [12] format. The protocol buffer encoding results in much reduced data size as compared to JSON and other formats. This reduces the access IO required for our analyses, which typically requires many consecutive samples. Other analyses with more random data access requirements might optimally use fewer samples per package. Before deciding on using Protocol buffers for the packaging step we compared the performance of protocol buffers with MsgPack [13] and JSON in terms of size and computation time on the BeagleBoneBlack (BBB) minicomputer using both python and C++ with the results shown in Table. I, averaged over 1000 runs. We did these test on a low performance minicomputer to confirm that even low power fog devices can perform the (de)serialization at real-time speeds, while still being able to do the analysis task required. We can see in Table. I, that the Protobuf is the best in terms of transportation size, which comes largely from the fact that Protobuf is *not* self-describing and requires the use of a static description file. MsgPack and JSON on the other hand are a lot bigger in

TABLE I
SIZE OF TRANSPORTATION CONTAINERS FOR ONE SECOND WORTH OF PMU DATA (6304 BYTES RAW DATA LENGTH) AND COMPUTATION TIME ON A BEAGLEBONEBLACK MINICOMPUTER.

	Protobuf	MsgPack	JSON
size (Bytes)	7703	13408	15487
serialization Python(ms)	47.0197	0.6806	6.4709
deserialization Python(ms)	48.6306	0.3453	2.0404
serialization C++(ms)	0.79528	0.47313	0.18463
deserialization C++(ms)	0.66304	0.29642	0.29469

size, but are self-describing and only differ from each other in the encoding of the data stream. The speed we found is largely impacted by the language, in which the individual packaging solution is written. The Protobuf solution in C++ is heavily optimized for this purpose, while the Protobuf python solution uses wrappers that reduce the speed dramatically as seen from Table. I. Msgpack in python on the other hand is very fast. JSON is by default integrated into python and is the largest container in terms of size. Since there is no default implementation for JSON in C++ we use [14], which has the fastest speed among the tested packaging solutions. In general, all of the solutions are acceptable for our application in terms of the speed, and we ended up using the Protobuf solution in C++ due to the data size advantage over the other solutions. Thus in our real time application of PMU-data a CPU load of 0.8% is used to perform this task on our BBB.

Lastly we store this data package in a commercial database [15] and compressed, using the built-in "DEFLATE" compression algorithm [16]; Combining packaging and deflate compression algorithm has resulted for us in a data size reduction of 9% compared to raw data. Note that this includes descriptive metadata added to describe the raw data such as the unique identifier of the data source (e.g., sensor or network security monitor) and timestamps, that are not included in the raw data byte-count.

C. *Short-term database*

In the short term Historian database we do not package sensor samples together, as we want to be able to quickly do on-demand computations on the data. In particular we are utilizing this for displaying data to the user in the requested time resolution. We are using the Elasticsearch search service for this purpose as it allows for server-side aggregation of data-points for the users display and caching of results.

In addition to the aforementioned databases the central RabbitMQ cluster will support any third party systems and software that understand JSON formatted data. We have verified this with streams to commonly used tools for operation tasks (such as the 'OSISoft PI Server') and security tools (such as 'Splunk SIEM').

D. *Deployment*

All of our server code is deployed in docker containers for ease of maintenance and scalability. This approach is fairly standard in the industry these days and not further discussed here. For the first analysis stage of sensor data, we support next to docker the option of utilizing local computing with a bastion host. Cheap minicomputers such as the Raspberry Pi or BeagleBoneBlack have sufficient computing power to handle a single real time stream. The benefit for this deployment is not only the option of prioritization and data driven compression as discussed before but also allows isolating the sensor from the rest of the network. This reduces the attack vector on each sensor dramatically as with this deployment only the deployed bastion host has to be kept secure, not the sensor itself. This fact is important as sensor vendors are often slow in updating their software to the latest security standards, if updated at all. Therefore the deployment of cheap bastion host can extend the lifetime of a sensor by protecting against security flaws of many sensor vendors at the cost of the attack vector against a single regularly updated open-source system.

Each component in our system is communicating encrypted and authenticated with other system members. Because we deploy identical containers/pre-configured BBB at multiple locations, we are using a distributed key management system [17], that like our architecture is also resilient to node and link failures, allowing automated key distribution and revocation.

E. Fail-Over and Resilience

In the presented architecture we have several methods to mitigate failures in devices and networking connections. In particular we have deployed clusters for the central messaging and historian solution, that offer redundancy. Furthermore we have deployed a buffer system at each stage that allows temporal storage of data, to mitigate any outages of upstream systems without loosing data. Important to note is that all analysis processes that are downstream of a failure will continue to work even if all upstream cluster members fail. This allows each system to locally react on events observed in the sensor data.

For fog nodes we did not deploy clusters for economic reasons and only use a buffer. We did this because our deployed analysis algorithms are tolerant to missing single measurements, thus we do not require a fail-over cluster; In particular because the sensors who are delivering data are also not 100% reliable thus analysis algorithms have to tolerate single sensor outages.

F. User Interface and API

To access the real time data, one can simply subscribe to a sensors or analytic processes data stream at any messaging layer and the requested data is delivered. By utilizing descriptive tags for each sensor measurement even subsection of data can be requested, such as only a particular tag or a down-sampled data-stream. This method ensures that each process is getting exactly the data it requires.

Central locations in addition to real-time data, can access the historian databases. This can be useful for testing new analysis processes quickly based on historic data; or training machine learning models. We did not deploy any historian at any other layers as this would significantly increase storage requirements as duplicates would be needed.

Furthermore we are utilizing this database access for a user interface to view the data-streams as a time series with the server side computations and caching as described in section II-C.

III. Conclusion

In this project we have designed, developed and deployed a real-time streaming architecture that has the unique feature of prioritization and data-driven compression, utilizing single or multiple data sources/sensors.

References

[1] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog Computing and its Role in the Internet of Things," in *Proc. Workshop on Mobile Cloud Computing (MCC)*. ACM, 2012, pp. 13–16.
[2] Z. Alliance, "Zigbee specification," in *ZB Alliance - ZigBee Document 053474r06*, 2004.
[3] R. Gentz, A. Scaglione, L. Ferrari, and Y. . P. Hong, "Pulsess: A pulse-coupled synchronization and scheduling protocol for clustered wireless sensor networks," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 1222–1234, Dec 2016.
[4] ApacheKafka, https://kafka.apache.org/, 2019.
[5] AmazonKinesis, https://aws.amazon.com/kinesis/, 2019.
[6] T. Li, K. Keahey, K. Wang, D. Zhao, and I. Raicu, "A dynamically scalable cloud data infrastructure for sensor networks," in *Proceedings of the 6th Workshop on Scientific Cloud Computing*. ACM, 2015, pp. 25–28.
[7] R. Rajagopalan and P. K. Varshney, "Data aggregation techniques in sensor networks: A survey," 2006.
[8] H. Chan, A. Perrig, and D. Song, "Secure hierarchical in-network aggregation in sensor networks," in *Proceedings of the 13th ACM conference on Computer and communications security*. ACM, 2006, pp. 278–287.
[9] K. Kalpakis, K. Dasgupta, and P. Namjoshi, "Efficient algorithms for maximum lifetime data gathering and aggregation in wireless sensor networks," *Computer Networks*, vol. 42, no. 6, pp. 697 – 716, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128603002123
[10] R. Smith, "Assault on california power station raises alarm on potential for terrorism," *Wall Street Journal*, vol. 5, 2014.
[11] RabbitMQ, https://www.rabbitmq.com/, 2017.
[12] K. Varda, "Protocol buffers: Googles data interchange format," *Google Open Source*, vol. 72, 2008.
[13] MsgPack, https://msgpack.org/, 2017.
[14] "Niles lohmann's json implementation for c++11," https://github.com/nlohmann/json, 2016.
[15] Cassandra, https://cassandra.apache.org/, 2019.
[16] P. Deutsch, "Deflate compressed data format specification version 1.3," Tech. Rep., 1996.
[17] DisruptionTolerantKeyManagement, https://github.com/pnnl/ADTKM/, 2018.
[18] S. Peisert, R. Gentz, J. Boverhof, C. McParland, S. Engle, A. Elbashandy, and D. Gunter, "LBNL Open Power Data," https://powerdata.lbl.gov, 2017.