FastReact: In-Network Control and Caching for Industrial Control Networks using Programmable Data Planes

Jonathan Vestin  
Dept. of Maths and Computer Science  
Karlstad University  
65188 Karlstad, Sweden  
Email: jonathan.vestin@kau.se

Andreas Kassler  
Dept. of Maths and Computer Science  
Karlstad University  
65188 Karlstad, Sweden  
Email: andreas.kassler@kau.se

Johan Åkerberg  
ABB Corporate Research  
Västerås, Sweden  
Email: johan.akerberg@se.abb.com

©2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Abstract—Providing network reliability as well as low and predictable latency is important especially for Industrial Automation and Control Networks. However, diagnosing link status from the control plane has high latency and overhead. In addition, the communication with the industrial controller may impose additional network latency. We present FastReact - a system enabling In-Network monitoring, control and caching for Industrial Automation and Control Networks. FastReact outsources simple monitoring and control actions to evolving programmable data planes using the P4 language. As instructed by the Industrial Controller through a Northbound API, the SDN controller composes control actions using Boolean Logic which are then installed in the data plane. The data plane parses and caches sensor values and performs simple calculations on them which are connected to fast control actions that are executed locally. For resiliency, FastReact monitors liveness and response of sensors/actuators and performs a fast local link repair in the data plane if a link failure is detected. Our testbed measurement show that FastReact can reduce the sensor/actuator delay while being resilient against several failure events.

I. INTRODUCTION

The domain of industrial automation and control has gained a lot of attention recently through the introduction of Internet of Things (IoT), including smart cities, home automation, self-driving cars, wearable IT and industry 4.0. These new use cases put great pressure on manufacturers to perform fast innovation, where each customer expects individually customized solutions. Industrial Control Networks (ICNs) must cope with such increased expectations, and allow for cloud-enabled re-configurable and re-programmable networking. However, at the core of automation and control is essentially the reliable and timely delivery and exchange of data. Consequently, the systems need to be designed for high availability and predictable low latency even under extreme circumstances, such as during link or equipment failures. This introduces significant challenges in designing a network which connects sensors, controllers and actuators at low latency in a flexible way.

In todays industrial setup, the integration between network and control is quite simple and the control is highly centralized and customized to the use case. Typically, the sensors generate (periodic) data, which are transferred to an industrial controller or an intermediate proxy node that analyzes the data. Based on the use case and received sensor values, the controller derives control actions and sends them to actuators. Also, the controller can modify network paths, link configurations or proxy locations. Nevertheless, the communication to the industrial controller may involve many hops, each one prone to failure and additional latency. In addition, the network protocols and processing of data packets in the switch are decoupled from the application logic, which consequently requires end-to-end control logic, involving actuators/sensors/controls.

Recently, Software Defined Networking (SDN) has shown to be an interesting architectural solution to make networks more flexible and programmable. In SDN, the control and data plane are decoupled. The control plane is opened up and logically centralized using a so called SDN controller, which enables faster innovation, manufacturer independent configuration and rapid deployment of custom network functions. A Network Operating System (NOS) continuously monitors and reconfigures the network according to customer needs, and provides monitoring statistics to find deployment issues and opportunities for service improvement. Industrial automation has seen an increasing interest in utilizing the advantages which SDN provides, but there still remains concern for unpredictable latency and reduced reliability. However, proactive failure recovery (e.g. fast failover [1]) and redundant controllers (e.g. ONOS [2]) provide solutions for enabling high availability in SDN-based industrial networks. Especially the recent development of programmable data planes using e.g. P4 [3] opens up previously unexplored use cases for industrial automation and control networks, removing many limitations of the southbound protocol OpenFlow [4].

This paper presents FastReact, which advocates the idea to outsource parts of the industrial controller logic to the data plane of new emerging programmable switch architectures. FastReact provides in-network sensor monitoring, control and data caching for industrial automation and control networks. In FastReact, programmable switches parse sensor packets at line rate and cache a history of sensor values in custom data structures in the switch. Highly customizable control logic is run inside the data plane and the switch can trigger local control actions from the data plane. The logic is specified through conditionals composed of boolean logic and comparison functions, which are installed by the SDN controller, as instructed from the Industrial Controller for the given use case. FastReact enables rapid in-network decision-making.
based on observed sensor data, historical sensor data, or
preconfigured lightweight statistical functions, such as moving
averages. Partially relocating the control logic to the data plane
significantly reduces the path between sensors and actuators,
improving latencies while also reducing potential points of
failure. Furthermore, as high availability is critical in ICN,
FastReact supports resiliency through link liveness monitoring,
performing proactive link repair if a link or piece of equipment
fail. We have implemented FastReact in the P4 language, and
evaluated it using the CORE network emulator. We find that
FastReact can effectively reduce the sensor-actuator delay while
also providing the resiliency required in industrial automation.
We also investigate the feasibility of implementing FastReact
on emerging programmable data planes, discussing required
features and memory requirements from the increased flexibility.

The rest of the paper is structured as follows: In Section II,
we present relevant background information. In Section III, we
describe the design of FastReact. In Section IV, we describe
the experimental setup and our results. We conclude the paper
in Section V discussing future work.

II. BACKGROUND

In the area of SDN for industrial automation and control,
there have been several works focusing on traditional SDN,
where a SDN controller manages flow table entries of an
OpenFlow capable switch. For example, [5] analyses the
potential for SDN in the industrial network context and identifies
important research challenges such as security, reliability, real-
time requirements, timing performance and Quality of Service.
[6] evaluated SDN in the field of industrial automation focusing
on the SDN controller. They evaluate the performance penalty
introduced by switch-controller-communication and showed
that SDN is suitable for real-time traffic and has great potential
for the use in the automation industry. [7] proposes a system
architecture for the deployment of software functionality to
manufacturing resources at runtime. It is based on an integration
of formal software model verification, SDN and a close
interaction between a flexible runtime execution environment
and its safety-critical counterpart. [8] finds that SDN is suitable
for complex automation environments but focuses on the real-
time data transmission using a Time Division Multiple Access
(TDMA) mechanism combined with parallel data transmissions
on physically separate links, taking application requirements
into account. [9] proposes a SDN based architecture that
implies software poison and reconfiguration capabilities in an
industrial control system. [10] advocates in-network sensor
data processing for robustness, reduced energy consumption
and latency but modifies the sensor node architecture in order
to support partial SDN functionality.

In terms of resiliency for SDN, the centralized network mon-
itoring allows the SDN controller to repair failed network links,
providing resiliency for the network. Two main approaches are
used for recovery: reactive and proactive link repair. When
using reactive link repair, the data plane device notifies the
SDN controller when one of their links fail. The SDN controller
uses this information to recalculate forwarding paths and install
new rules in the network. While this approach is simple and
memory efficient, recovery can take a significant amount of
time [11]. The other approach is proactive, thus pre-installing
the backup paths in the network devices, allowing for fast local
failure recovery [12].

With the recent introduction of programmable data planes,
fueled by novel switch chip architectures such as Barefoot
Tofino or Cavium, programmable NetFPGAs and SmartNICs,
significant limitations of SDN caused by OpenFlow [4] are
removed because it allows parsing arbitrary protocol headers
and stitch together much more flexible and customized switch
pipelines. The introduction of language and compiler support
such as P4 [3] enables new use cases for SDN, because it enables
the data plane to perform novel functionality. For example,
NetCache [13] leverages the potential of programmable data
planes to perform in-network caching for key/value stores and
NICached [14] enables the programmable NIC to function as
data cache. Failure recovery has been implemented using
P4 OpenState through SPIDER [15]. However, to the best of our
knowledge, the potential of programmable data planes has not
yet been applied yet to the in-network monitoring and control
use case that would be required for industrial automation and
control networks. A flexible and custom specification of control
logic is a key problem that must be solved.

III. FastReact DESIGN

The main idea of FastReact is to outsource parts of the
control logic of an Industrial Controller to the switch data
plane, which will essentially perform in-network processing of
sensor data. FastReact parses sensor packets using emerging
programmable data plane, caches sensor values, computes lightweight operations on sensor values and, based on highly configurable logic, locally triggers actuator messages from the data plane. The microcontroller logic is configured into the FastReact switch by a SDN controller through populating a set of custom tables. Finally, the SDN controller exposes a North-bound API that is used by the Industrial Controller. This API allows specifying the internal processing logic of any FastReact-enabled network switch. The SDN controller then composes processing rules using boolean expressions in normal form, which are transformed into register array entries that the SDN controller installs on the programmable switches.

By acting locally from the data plane, the control loop between sensor and actuator can be significantly shortened, because the decisions are made closer to the sensor and/or actuators avoiding the round trip to the industrial controller (see Fig. 1). Finally, the switch acts as a cache for sensor values and can correlate multiple sensor values to process more complex logical operations. The logic is implemented in P4 using several custom data structures implemented by registers, which allows it to be executed on P4 supported hardware. Thus, sensor packets can be processed at line rate, performing rapid, deterministic decision-making. In addition, this also increases the reliability of the network, as moving the control logic closer to the sensors and actuators reduces the potential points of failure. A high level overview on the architecture of FastReact can be seen from Fig. 2.

A. FastReact - Sensor Packet Parsing and Processing

When a sensor packet enters the FastReact switch, we parse the sensor ID, extract the sensor value and store it in a time series datastore implemented by custom register arrays. We keep a rolling history of the latest \( n \) values per sensor ID, where \( n \) is configurable. When inserting a new sensor value, we also compute a moving average in a separate register array using approximative arithmetic [16]. Both the historical sensor values and moving averages can be fetched by a SDN or industrial controller either through the data plane using a get-like request, or through the control plane via the P4-generated API. In that sense, FastReact acts similar as a key-value cache for sensor readings with the additional functionality to implement lightweight compute operations. In contrast to NetCache [13], which keeps only the hot items in the cache and has a cache replacement strategy, we keep a fixed number of sensor IDs in the switch memory in order to be deterministic. Also, we argue that the Industrial Controller knows how many sensor IDs are deployed, so it can configure the table size during compile time of the switch.

Based on the configured processing logic (see Section III-B), the switch decides, if it should discard the packet, send it to the Industrial Controller or notify an actuator. The connection between sensors and actuators are configured through the match-action table, which is populated by the SDN controller as instructed by the Industrial Controller. If the conditions encoded in the logic tables are true, the actuator specified in the match-action table will be notified of the sensor value. The action will modify the sensor packet, transforming it into an actuator notification message as specified by the SDN controller.

An example of a match-action table can be seen in Table I. The table has two actions, named Route and Failover, respectively. The Route table is used to connect sensors and actuators. In this example, Sensor 1 notifies Actuator 2, while Sensor 2 notifies Actuator 5. Multiple actuators could be notified through separate actions, as P4 does not support variable number of arguments. The Failover table specifies actions the switch should take if a failure is detected. Here, for Sensor 1 the data is forwarded to Actuator 3, while for Sensor 2 the data is sent up to another switch, which, in turn, can pass the packet to a backup actuator.

B. FastReact - Decision Logic

One of the main challenges in FastReact is how to make the decision logic in the switch as generic as possible while being able to dynamically update it through the south-bound API. Most P4 targets expose the ability to read from and write to switch registers, which are array-like data structures stored in the switch memory. Our implementation utilizes these registers to communicate the desired control logic for each switch. In order to encode a complex logical expression in a generic way, we first transform it into its Conjunctive Normal Form (CNF). The CNF is given by:

\[(A \lor B \lor C...) \land (D \lor B...) \land ...
\]

Where \( A, B, C... \) are logical expressions in the form \((s \_v)\). Here \( s \) represents the recorded value of a sensor, \( v \) the value which it should be compared to and the placeholder \( \_ \) represents an operator. A simple example for a logical expression in CNF is the following:

\[(s_1 < 50 \lor s_2 > 25) \land (s_3 = 10)\]

Based on the processing logic provided by the Industrial Controller, the SDN controller derives the CNF of the logical expression and installs it on the switch, in a table format, by using switch registers. This is stored on a per-sensor basis.

The Tables II and Table III shows how this data is stored in the switch. Table II is the conjunctive table, where each row contains a sensor ID signifying which sensor the expression should be applied to. It also contains a set of indices which point to entries in the disjunctive tables (Table III). There are three disjunctive tables, one for sensor IDs, one for operators and one for values, which encode the components of the comparative expression (e.g. \(S1 < 50\)). Each column represents a separate conditional (e.g. \(A \lor B\)). These tables are loaded into the switch, which transforms them into a logical expression.

List 1 shows a shortened version of the P4 code for the decision logic. This code is automatically generated given parameters such as maximum number of disjunctive and conjunctive expressions, supported operators etc. As a packet arrives at the switch, the program loads the indices from the conjunctive table, corresponding to the sensor ID in the packet header. If there is logic configured (\(cidxN !\neq 0\)), it loads

| Table  | Match Key | Action       |
|--------|-----------|--------------|
| Route  | sensor = 1| forward_mod(2 (actuator, ...)) |
| Route  | sensor = 2| forward_mod(5 (actuator, ...)) |
| Failover| sensor = 1| forward_mod(3 (bkup, actuator, ...)) |
| Failover| sensor = 2| send_up(10 (switch)) |

TABLE I: Example of match-action table

\(cidxN \neq 0\)
TABLE II: Example of sensor conjunctive table for the expression 
\((S1 < 50 \lor S2 > 25) \land (S3 = 10)\)

| Index | Cond 1 | Cond 2 | Cond 3 | Cond 4 |
|-------|--------|--------|--------|--------|
| 1     | 1      | 2      |        |        |
| 2     | 3      |        |        |        |

Operator

| Value |
|-------|
| 1     |
| 2     |

TABLE III: Example of sensor disjunctive tables for the expression 
\((S1 < 50 \lor S2 > 25) \land (S3 = 10)\)

```c
conj_table.read(cidx1, hdr.sensorId * CONJ_TABLE_SIZE);
conj_table.read(cidx2, hdr.sensorId * CONJ_TABLE_SIZE+1);

if (cidx1 != 0) {
    disj = false;
    disj_table_op.read(opcode, cidx1 * DISJ_TABLE_SIZE);
    disj_table_id.read(sensor_id, cidx1 * DISJ_TABLE_SIZE);
    disj_table_val.read(value, cidx1 * DISJ_TABLE_SIZE);
    sensor_latest.read(sensor_value, sensor_id);

    if (opcode == 1 && sensor_value == value) disj = true;
    if (opcode == 2 && sensor_value > value) disj = true;
    [...] disj_table_op.read(opcode, cidx1 * DISJ_TABLE_SIZE+1);
    [...] disj_table_id.read(sensor_id, cidx1 * DISJ_TABLE_SIZE+1);
    [...] disj_table_val.read(value, cidx1 * DISJ_TABLE_SIZE+1);
}
if (!disj) should_drop = true;
}
if (cidx2 != 0) {
    [...]}
```

Listing 1: Shortened sensor logic code.

the sensor value, operator, and value for the first conditional. Now, the comparative expression is performed, and if true, the program flags this (\( disj = true \)). After processing all disjunctive conditionals, if the flag is not set, the packet is dropped because one of the conjunctive terms are false. This is done for each condition in the conjunctive table. If the packet isn’t dropped, the switch forwards it according to its forwarding table.

C. FastReact - Failure Recovery

In order to provide failure recovery, FastReact monitors sensor and actuator liveness and reacts locally from the data plane should a sensor or actuator fail. In FastReact, the switch records the timestamp of the last received packet for each switch port in a register array. Assuming that both sensor and actuators send periodic liveness messages, when no messages have been received on a particular port for a certain time, it is considered down. This port timeout interval is configured through a register array, where each entry is the timeout for a particular output port. For each actuator, a backup actuator can be specified in a failover match-action table. When a packet should be forwarded to an actuator which is considered down, the packet is instead forwarded to the backup actuator. This forwarding may be immediate (if it is reachable directly from the switch), or through another switch (as in our experiment, depicted in Figure 4). If there are multiple backup actuators, and the FastReact switch is directly connected to them, it will pick the first live one. If it is not directly connected, it is the responsibility of the FastReact switch connected to the backup actuator to decide where to send the packet next. Again, it is up to the SDN controller to push the proper table entries to specify that behavior in more detail, which is outside the scope of this paper.

D. FastReact - Filtering

To reduce the amount of traffic, FastReact switches can filter the packets sent to the controller. This filtering can be done either using the switch decision logic programmed through the register tables, or we can only forward every \( n \)-th packet. The filter logic is configurable on a per-sensor basis through the match-action table. For each sensor \( ID \), the switch keeps track of how many messages have been received and compares this count to the filtering rate, determining if the incoming packet should be forwarded or discarded. The filtering based on the decision tables is performed just like any other packet, discarding packets which do not match the logic configured through the register tables.

E. FastReact - In Network Caching

FastReact also supports requesting stored sensor values, running averages and simple computation of the time series database through the data path using a \texttt{get} request. If the industrial system contains a large number of sensors, it may be beneficial to have devices request values as they are needed in order to reduce traffic. The switch keeps historical records of the latest received sensor values, along with reception timestamps. These timestamps can be used to determine the age of a certain sensor value. When the switch parses a packet in the ingress and detects a \texttt{get} request, the timestamp of the latest historical value for the requested sensor is compared to the current time. If the difference is smaller than a configurable tolerance, the switch forwards the request to the sensor. This request is replied to with a sensor value updated, which allows the switch to update its time series database, while also passing the message on to the original requester. To instruct the switch what kind of data it should return (e.g. most recent value, moving average), an OpCode is provided in the \texttt{get} request with the sensor \( ID \).

F. FastReact - Feasibility in P4

Our P4 implementation requires support for basic P4 primitives defined by the P4_16 standard [17]. In addition, it requires support for registers reading, register writing, IP and UDP checksum calculation and stateless header parsing. Support for ingress timestamps is also required.

Sensor Dependency Table: The switch memory requirements depend on the logic table sizes, the number of sensors supported and the history size. The size of the conjunctive table is \( S_{count}C_{cols}\lceil \log_2 D_{rows} \rceil \) bits. Here \( S_{count} \) is the number of sensors, \( C_{cols} \) is the maximum number of conjunctive conditionals and \( D_{rows} \) is the maximum number of rows in the conjunctive table. The size of the disjunctive table is
G. FastReact - Implementation

In order to evaluate the feasibility of our design, we implemented a prototype in P4 using the v1model architecture. As can be seen from Fig. 3, packets are parsed, determining if they are sensor messages. get requests (not shown for simplicity) or regular network traffic. If a sensor packet is detected, the sensor value is extracted and recorded in the time series registers. Then, the decision logic is applied, which determines if the packet should be dropped, sent to another switch or an actuator should be notified. If an actuator is notified, the forward(port) action simply forwards the packet normally and the forward_mod(port, ip, mac) action sends a notification to an actuator.

Then, the ingress timestamp of the packet is compared to the latest timestamp of the output port. If the difference is greater than the configured tolerance, the failover table is applied. The failover table can only result in the send_up(port) action. The send_up(port) action will change the packet type to indicate that it is using a backup route and sends it to the specified port. This indication is later used by other switches to help determine the appropriate backup destination. Finally, checksums are recalculated and the packet is moved to the egress pipeline.

Filtering is performed after the decision logic, where a filtered packet counter is incremented, and compared to the filter rate, determining if the packet should be filtered (dropped) or forwarded. Cache timestamps are updated when the sensor value is recorded, while get requests are processed in the beginning of the pipeline. These get requests are handled separately from the sensor logic. When a get request enters the switch, the timestamp of the latest historical sensor entry is compared to the ingress packet timestamp. If the difference is smaller than a configurable tolerance, a new sensor value is requested from the sensor. Otherwise, the operational code is extracted and the operation on the time series registers is calculated and returned from the cache.

IV. Evaluation

In this section, we evaluate the effectiveness of FastReact by implementing it in P4 and performing several experiments using the CORE network emulator. CORE uses Linux Network Namespaces to create virtual hosts and links. These are run using the normal Linux networking stack, which allows integration of the P4 behavioral model. We use the modified CORE version which uses netem instead of tbf for rate management, which allows to tune the queue length.

A. Experimental Setup

We set up a testbed (see Fig. 4) for validating the basic functionality and performance characteristics. The industrial controller is connected to the switch C1, which in turn is connected to the switch B1. B1 is connected both to switches A1 and A2. Both switches have one sensor and one actuator connected, named Actuator 1 and Sensor 1 for A1, and Actuator

---

1The P4 code will be available at https://github.com/jonavest/P4-FastReact
2Available at https://github.com/p4lang/behavioral-model
3Available at https://github.com/tohojo/core
2 and Sensor 2 for A2 respectively. Each link is configured with a latency of 1 ms, and a throughput of 1 Gbps. Therefore, the delay between Sensor 1 and Actuator 1 is 2 ms, while the delay between Sensor 1 and Actuator 2 is 4 ms. For the path Sensor → Controller → Actuator, the total delay is 8 ms.

All experiments are run on a Dell Optiplex 9020 (4-core i7 4790 3.6GHz, 8 GB RAM), running version 4.1.39-rt47 of the Linux kernel. The Linux kernel has been patched using the PREEMPT_RT patch, running with the PREEMPT_RT_FULL model, in order to reduce latency and jitter caused by process scheduling, while improving timestamp accuracy.

B. Results

1) Baseline: First, we performed a reference experiment using traditional switching, which is not aware of the sensor format and can not react to changes in the sensor readings. All sensor messages are sent to the industrial controller which then sends a reply to the appropriate actuator. In this experiment, Sensor 1 sends continuous sensor messages to the industrial controller over 40 s with an interval of 10 ms. The industrial controller will receive these messages and passes an appropriate action to Actuator 1. Fig. 5 shows the results of the experiment. In the figure, the x-axis represents the time when Sensor 1 sent the packet, and the y-axis represents the latency between Sensor 1 and Actuator 1. We compare the delay with our approach FastReact, where the P4 enabled switch parses sensor packets in the data plane, checks the control actions and directly sends commands from the switch to the actuator. As we can see from our experimental data, the sensor-actuator delay averages to 8.201 ms for the case where the Industrial Controller decides the action compared to an average delay of 2.249 ms for FastReact. The significant reduction in latency is because the sensor messages only pass through the switch, and we avoid the costly communication to reach the industrial controller.

2) Failover: In order to evaluate the failure recovery behavior of FastReact, we performed a second set of experiments. First, we evaluated how resilient the network is to a single link failure. In this experiment, Sensor 1 sends messages at a 10 ms interval and the port timeout is set to 30 ms. If the switch does not receive any packets for 30 ms on a port, that port is considered to be down, and the message will be forwarded up the network, to switch B1. B1 sends the message down to switch A2 which sends it down to Actuator 2. Fig. 6 shows sensor-actuator delays over time, and the lines show the link states of Actuator 1 and Actuator 2 respectively. As we can see, when the link between Actuator 1 and A1 or the node Actuator 1 fails, FastReact immediately reacts by sending the sensor values through B1 and A2 to Actuator 2, which causes an increase in the latency from an average of 2.244 ms to 4.624 ms. This increase is due to the longer path between the sensor and the actuator. It is worth pointing out that if both links between Actuator 1 and A1 and between Actuator 2 and A2 fail, no data is received, because no actuator is reachable. The number of packets lost during the local repair in this scenario is between 2 and 3.

One important aspect of how fast the switch can recover from a link failure is determined by the packet sending interval configured in the actuator and the port timeout configured in the switch. The shorter the sending interval, and the shorter port timeout, the faster faults can be detected. However, using a short sending interval increases the amount of traffic on the link, and using a short port timeout increases the probability for false positives (due to unexpected packet loss or jitter). Fig. 7 shows the sensor-actuator delay with varying actuator sending intervals and port timeouts. In this experiment, packets are sent between Sensor 1 and Actuator 1, as the link between Actuator 1 and A1 is taken down. The measurement points in the figure are the difference between the last received packet for Actuator 1 and the first received packet for Actuator 2. From the figure, we can see that the recovery time is highly impacted by both sending interval and port timeout. For short port timeouts of 10 ms in combination with a short sending interval of 5 ms, the mean sensor-actuator delay (during failure) is 14.45 ms. The missing measurement points are in the cases when the sending rate is slower than the port timeout, causing both ports to be in a down state, and no packets are being transmitted.

3) FastReact Logic Processing: FastReact is capable to correlate multiple sensor values in order to determine proper actions as given by the control logic in the data plane. This
This behaviour is of course fully configurable depending on the application.

In the next experiment, we have two sensors, *Sensor 1A* and *Sensor 1B*, which report integer values. The *FastReact* logic is configured so that when the reported value exceeds 50 for both sensors we want to notify the actuator, i.e. \((A \geq 50) \land (B \geq 50)\).

Because we record the historical values for each sensor in the data plane in *FastReact*, we compare the received sensor value and correlate it with the previously received value of the other sensor. Fig. 9 shows two dashed lines representing the delay between sensor and actuator. The sensor values increase by 1 unit every second. The sensors are configured to start at 20 and 30 respectively. As we can see from the figure, the switch is able to keep the state of both sensors in memory, and wait for both sensors to reach the configured value before it triggers the fast reaction and notifies the actuator.

In order to demonstrate the moving average functionality, we constructed an additional scenario where the sensor values occasionally spike. If we were just looking at the last sensor value, we would notify the actuator intermittently, causing maybe an unnecessary activation of the actuator leading to state oscillation. The spikes occur with an interval of 7 s and changes the sensor value to 55 for 50 ms. As we can see from Fig. 10, two spikes occur at \(t = 7\) and \(t = 14\). The first does not trigger an alert to the actuator, because the moving average evens out the matched value. However, the later spike (at \(t = 14\)) triggers an immediate fast reaction and sends a message directly to the actuator, because the moving average will increase above 50.

Fig. 11 shows the effect of filtering sensor messages at the switch. In this experiment, messages are always sent from the sensor to the industrial controller, and then forwarded to the actuator. No fast reaction is employed, however, packet filtering is enabled with various sampling rates. The sampling rate can be seen in the x-axis of the figure while the y-axis is the network load (in packets) and the time between messages received at the controller. The sensor-actuator delay show the

---

**Fig. 7:** Sensor-Actuator delays in case of a failure with varying actuator sending intervals and port timeouts.

**Fig. 8:** Sensor-Actuator delays with binary sensor values.

**Fig. 9:** Experiment using two sensors with stateful switching.

**Fig. 10:** Experiment using single sensor and actuator. Sensor values suffering from spikes. Using moving average values instead of latest. Points are drawn on the figure lines during the spikes for readability.

**Fig. 11:** Experiment filtering packets at various sampling rates.
Fig. 12: Time between request and response for get requests sent from the industrial controller to a sensor node.

increase in delay due to filtering messages. Decreasing the sampling rate reduces the load on the network, while increasing the potential sensor-actuator delay.

Finally, Fig. 12 shows the industrial controller continuously sending get requests to the sensor node. As the cache timeout is set to 5s, the first request is transmitted through A1 to the sensor, while in the 4 subsequent requests the value has been cached by FastReact. As such the cached sensor value is immediately returned by the switch. The y-axis shows the time between the controller request was sent until the response was received. The “Not Cached” dots shows the request time when there is no cache entry, while the “Cached” dots shows the request time when the sensor value is in the switch cache. As we can see from the figure, the average delay between request and response is 8.73 ms if the value has not been cached, and 6.38 ms if the value has been cached by the switch.

V. CONCLUSION

In this paper, we have introduced FastReact, an in-network monitoring, control and caching system for industrial automation. Based on highly configurable logic expressed in boolean algebra, the SDN controller tailors the control logic of the switch data plane as instructed by the industrial controller. FastReact switches perform lightweight computational operations and local control actions, while being resilient to link or sensor/actuator node failures. We have presented a high-level design of the system, and discussed the feasibility of implementing it on emerging programmable switch platforms. We discuss scalability issues in relation to implementation and platform constraints as given by the P4 language. Finally, we have implemented and evaluated basic functionality and performance characteristics, showing that FastReact can significantly reduce the sensor/actuator delay, while still providing high resiliency to network failures.

As for future work, we intend to evaluate the performance impact of FastReact on the data plane performance using NetFPGA and programmable network cards. In addition we intend to focus on SDN controller interaction with Industrial controller for a more flexible design of the north-bound API.

ACKNOWLEDGEMENT

Parts of this work has been funded by the Knowledge Foundation of Sweden through the project READY.