TELE 6603: IoT Assignment #4, 100 points total)

Due on 4/18/17

OPTION 1:

Please read the following paper and answer the following questions:

1) In Section 3, SPDY is defined as a transport service that makes "HTTP faster". It says that SPDY relies on a single TCP connection. How is this different than persistent HTTP (set #2, slide 91)? Can you show the message exchange between a client and a server using SPDY when the client requests a web object ? (12.5 points)

2) Also in Section 3, why do you think that the TCP SACK option is removed from IoT-HTTP and IoT-SPDY to improve performance? (12.5 points)

3) What is the average transmission rate (bps) if an infinite number of TFO connections are created and terminated back-to-back between a client and a server when 1000 bytes are sent? Assume a 10 ms RTT. (12.5 points)

4) Based on the information presented in Section 4.3, what are the corresponding transmission rates (bps) for cases presented in Figures 6 through 10 ? (12.5 points)

5) In Section 5, why do the authors say *... the above analysis suggests that it is worth investigating the possibility to further reduce the overhead associated with SPDY* ? (12.5 points)

6) Also in Section 5, why do the authors say *...instead of using the TCP sliding window mechanism, we can use a stop-and-wait protocol...* ? (12.5 points)

7) Given the nature of HTTP, SPDY, CoAP, IoT-HTTP and IoT-SPDY, what are the implications in regards to packet loss and latency ? What would happen if impairments are introduced in a client-server communication based on these protocols ? (12.5 points)

8) Assume you are a performing peer-review of this paper for a journal editor: what are the paper weak points? what is it missing? what would you modify? (12.5 points)

OPTION 2:

Goal: Compare CoAP to HTTP in the context of packet loss and latency to measure throughput. Specifically include:

1. Detailed description of the experimental framework.
2. Presentation of experimental results. Are they expected? Why? Why not?
3. Conclusions including improvements to the framework and potential future work.

Requirements:

- Emulate a CoAP scenario with a server that responds to CoAP GETs with a random ASCII voltage readout (i.e."49.5 V") coming from a simulated analog-digital-converter. Requests must be sent every 20 ms.
- Emulate an HTTP scenario with a server that responds to HTTP GETs with a random ASCII voltage readout (i.e. "49.5 V") coming from a simulated analog-digital-coveter. Requests must be sent every 20 ms. The HTTP response must not include HTML code, it should only contain the readout string.
- Measure throughput when introducing 0%, 1% and 2% uniform unidirectional packet loss from client to server. Consider 0, 10 and 50 ms latency too. No latency or packet loss from server to client. The linux traffic controller can be used to induce these impairments.
- Analyze results. Are they expected? Why? Why not? How do latency and loss affect the throughput?
- Since impairments attempt to emulate wireless impairments typical of WSN, there is no need to rely on wireless L2 and L3 (802.15.4/6LoWPAN). In other words, regular IPv4 over ethernet/wifi is fine so it can be implemented on any linux environment.

There are no restrictions on how this is implemented as long as it models the problem with accuracy. These are some possibilities:

- Use open source stacks like libcoap (i.e. compiling it to work under Linux with POSIX).
- Write a lightweight socket based client and server that emulates CoAP and HTTP.
- Use apache to emulate an HTTP server.
- Use WGET with some scripting to emulate an HTTP client.

etc etc...

Disclaimer: This option requires software development experience. Whoever chooses it must be able to find his/her way around code, open source projects, scripting, SDKs and IDEs.

Experimental Evaluation of the CoAP, HTTP and SPDY Transport Services for Internet of Things

Laila Daniel[1], Markku Kojo[1] and Mikael Latvala[2]

[1] Department of Computer Science, University of Helsinki, Finland
[2] Mosa Consulting, Finland
{ldaniel,kojo}@cs.helsinki.fi,
mikael.latvala@mosa-consulting.com

Abstract. Internet of Things (IoT) seeks to broaden the scope of the Internet by connecting a variety of devices that can communicate with the Internet. Transport services for IoT are crucial in enabling the applications to communicate reliably and in a timely manner while making efficient and fair use of the potentially scarce network resources. The communication with IoT devices is often implemented using HyperText Transfer Protocol (HTTP) or a specifically designed protocol such as Constrained Application Protocol (CoAP) that is a specialized web transfer protocol for constrained nodes and networks. In this paper we discuss various options for modifying or adapting HTTP to offer better transport service for IoT environments. We consider HTTP, SPDY that has been developed to speed up HTTP in general, IoT-HTTP and IoT-SPDY that are adaptations of HTTP and SPDY for IoT, and CoAP as transport services for IoT and experimentally evaluate their performance. The results of our experiments show that CoAP has the lowest object download times and the least number of bytes transferred compared to the other four transport services. IoT-HTTP and IoT-SPDY have around 50% shorter object download times and smaller number of bytes transferred compared to HTTP and SPDY.

1 Introduction

Internet of Things (IoT) refers to an emerging scenario in which a variety of Things (devices, appliances, sensors) equipped with Internet connectivity are embedded in various settings such as automobiles, buildings, homes, forests, etc. and can be used to collect data, communicate and make decisions with or without human intervention [1]. This scenario is interesting and has a wide range of applications in a variety of fields including environment monitoring, surveillance, emergency and rescue, and health care.

To extend the Internet services to IoT devices, a suitable transport service is needed. As the IoT devices have limited resources in terms of computation, communication, radio and battery life, the transport services should be simple, scalable, robust, efficient in making near-optimal use of resources, easy to maintain and deploy and also customisable to the need of the applications.

HyperText Transfer Protocol (HTTP) [12] is the de facto standard for information transfer in the Internet. It operates in a request-reply mode in a client-server environment on top of the Transmission Control Protocol (TCP) [19]. HTTP is often used to

G. Fortino et al. (Eds.): IDCS 2014, LNCS 8729, pp. 111–123, 2014.

implement the communication with IoT devices as it enables the IoT devices to connect to the Internet easily and directly. The main problems in using HTTP in an IoT environment are the lengthy HTTP headers and the need to establish TCP/IP sessions for each request-reply data transfer.

SPDY [5, 9] is a transport service developed by Google to speed up HTTP. It multiplexes several HTTP transactions with priorities over a single TCP connection and employs header compression to reduce data volume of the HTTP headers. Even though SPDY uses header compression, SPDY has the 'verbose' headers of HTTP in addition to the need to establish a TCP connection for data transfer.

Constrained Application Protocol (CoAP) [22] is a transport service designed specially considering the requirement of constrained devices such as sensors and IoT devices. As CoAP has a short binary header, the header overhead in transferring data can be kept at very low level. On the other hand, a CoAP-HTTP proxy is needed to connect a CoAP client to an HTTP server or vice-versa. The IETF RFC on CoAP [22] defines a basic mapping between HTTP and CoAP. In addition, deploying proxies can have scalability issues.

In this paper we examine whether HTTP and SPDY can be adapted to favourably compare with CoAP as they can directly connect the IoT devices to the Internet without the need for proxies. In order to have comparable performance with CoAP in terms of object download time and total bytes transferred to fetch an object, we propose minimizing the HTTP and SPDY headers along with TCP enhancements. We use an extension to TCP known as TCP Fast Open [10, 20] to reduce the object download time. We refer to the enhanced HTTP and SPDY proposed here as IoT-HTTP and IoT-SPDY. The results of our experiments show that by using IoT-HTTP and IoT-SPDY there is at least 50% reduction in object download time and bytes transferred compared to HTTP and SPDY. However, we note that CoAP still has at least 50% lower download time and needs fewer bytes to fetch an object compared to IoT-HTTP and IoT-SPDY respectively.

The organisation of the rest of the paper is as follows. Section 2 describes the related work and Section 3 describes the proposed transport services IoT-HTTP and IoT-SPDY in detail. Section 4 presents an experimental evaluation of different transport services in an IoT environment. Section 5 discusses some additional enhancements to IoT-HTTP and IoT-SPDY that need further evaluation and study. Section 6 concludes the paper.

2 Related Work

Relatively few studies are available in the literature on the comparison with CoAP and HTTP in wireless sensor networks. A comparative study of CoAP and HTTP in terms of mote's (wireless sensor node) energy consumption and response time carried out using simulations and experiments in real sensor networks is given in [11]. The simulation results on energy consumption by motes show that energy consumed by CoAP is about half that of HTTP in processing packets while in transmitting packets CoAP consumes only one fourth. The experiments in real sensor networks show that CoAP's response time is nine times lower than the response time of HTTP. CoAP and HTTP are evaluated in a use case of an intelligent cargo container that transmits information such as temperature, humidity of fruits and meat inside a container during land or sea

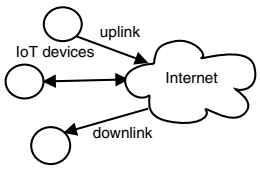


Fig. 1. Typical IoT Topology

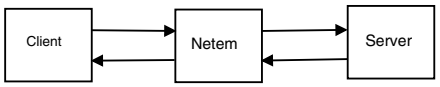


Fig. 2. Experimental Setup

transportation [16]. The results show that CoAP transfers smaller number of bytes and that it features shorter retrieval times compared to HTTP. The authors also compare CoAP/UDP with HTTP/TCP and HTTP/UDP and the results show that HTTP/TCP has longer retrieval times compared to UDP based protocols due to the initial TCP connection establishment. Paper [18] evaluates HTTP/TCP, HTTP/UDP with TinyCoAP, a CoAP implementation in TinyOS [6]. It shows that for transferring small objects, HTTP/UDP is a better choice than TinyCoAP while for transferring large payloads TinyCoAP has the best performance in terms of latency and energy. Analysis of CoAP and HTTP in IoT environments using the total cost of ownership (TCO) model is given in [17]. The paper shows that CoAP is more cost-efficient than HTTP when smart devices communicate frequently with each other. Also CoAP is found to be economically preferable when the charging of the communication is based on the volume of transferred data. The survey paper [14] gives a detailed description on the IETF Standardization in the field of the Internet of Things (IoT) and also compares HTTP and CoAP as the transport services in IoT environments.

In presenting our work, we are not aware of any previous studies that evaluate CoAP, HTTP and SPDY in a comparative manner.

3 Transport Services for IoT

In this section we describe briefly the requirements for an IoT transport service and discuss the five transport services we consider in this paper.

The IoT transport service should provide easy connectivity to the Internet. In this study we are focusing on the IoT topology as shown in Figure 1 where the IoT devices are directly connected to the Internet. The IoT devices may also be connected to a gateway or to a base station that has a connectivity to the Internet. To connect the IoT devices to the Internet, the transport service should be compatible with TCP/IP protocol suite, should be an open standard and proven to be scalable. So in our study we only consider transport services based on HTTP, SPDY or CoAP that are Internet Engineering Task Force (IETF) specified.

The transport services should provide congestion control to regulate the data flow that applications may send to the network and achieve some sort of fairness in sharing the scarce network resources. Even though reliable data transfer is not always a requirement, the transport service should provide reliable data delivery if the application needs it. Congestion control and reliable data delivery may be the functions of the underlying transport protocols. If these functions are not provided by the underlying transport protocol, they may need to be implemented at the upper layers. The congestion control and

the reliability mechanisms needed depend on the mode of data transfer between the IoT devices and the Internet. In push data and request-reply modes of data transfer where a small amount of data from the IoT devices are to be transferred, simple congestion control and reliability mechanisms are needed. On the other hand as data volume increases as in continuous data transfer like imaging data from a habitat, more advanced mechanisms to enforce congestion control and reliability are called for.

As HTTP [12] is the de facto standard of information transfer in the Internet, the main advantage of using HTTP as a transport service is that any device with HTTP can be directly connected to the Internet. Another advantage of using HTTP is that TCP which is the transport protocol for HTTP provides congestion control and reliability. HTTP operates in a request-reply mode in a client-server environment. The client request is called the GET request and the reply for the GET request from the server is known as the ACCEPT message. The header overhead associated with GET and ACCEPT messages is quite large for transferring data in push data and request-reply modes, when only a small amount of payload bytes are to be transferred from the IoT devices to the Internet. As HTTP uses TCP, the connection establishment takes at least one round-trip time(RTT) before the actual data transfer can take place. So for short transfers like in push data and request-reply modes, the connection establishment time may be longer than the time taken for actual data transfer.

SPDY [5, 9] is a transport service developed by Google to make HTTP faster i.e., to reduce the latency of applications that use HTTP. SPDY can be placed at the session layer in the OSI model where the application layer protocol is HTTP. To speed up the HTTP requests and responses, SPDY allows many concurrent HTTP requests in a single SPDY session over a single TCP connection. These concurrent requests are called streams that are bi-directional flows of bytes across a virtual channel in a SPDY session. Using a SYN_STREAM control frame, a stream can be created and a stream ID is associated with each stream. In the SYN_STREAM frame, additional information such as upload bandwidth, download bandwidth, round-trip time, initial window size can be sent through the HEADER block's name/value pairs. The name/value pairs, also known as ID/value pairs in the header block are usually compressed. After the SYN exchange, client and server send a SETTINGS frame that contains the configuration data and this data can be used for future connections from the same IP address and from the same TCP port. SPDY compresses HTTP headers resulting in fewer packets and fewer bytes to be transmitted thereby reducing the bandwidth used by HTTP. SPDY enables the server to initiate the communications with the client and push data to the client whenever possible. SPDY also uses the SSL protocol for security. Google claims that with SPDY around 50% reduction in pageload time can be achieved [5].

CoAP [22] is a transport service for use with constrained devices (devices with low power, small memory and lossy links) and constrained networks. CoAP operates in a request-response mode and its transport layer protocol is UDP. As CoAP has a short binary header of four bytes, the small header overhead in transferring a request-response type of messages suits well for IoT environments. There is no connection establishment phase as CoAP operates over UDP which is a connectionless datagram protocol. CoAP supports optional reliability with exponential backoff for messages. Many issues are associated with using CoAP as a transport service for IoT devices. To directly connect

the IoT devices to the Internet, a CoAP-HTTP proxy is needed and the scalability of proxies is always a concern. The IETF RFC on CoAP [22] defines a basic mapping between CoAP and HTTP. As this RFC is very recent, future changes in CoAP may pose challenges to the mapping between CoAP and HTTP. CoAP supports optional reliability with exponential retransmission timeout backoff to implement a simple congestion control mechanism.

In this paper we propose IoT-HTTP and IoT-SPDY in which we seek for adapting HTTP and SPDY, respectively, to make them better suit as transport services for IoT. In IoT-HTTP and IoT-SPDY, we minimize the headers associated with HTTP GET request and ACCEPT message and also enhance TCP with TCP Fast Open [10, 20], a mechanism that reduces the connection establishment time for successive TCP connections between two end points. In IoT-HTTP and IoT-SPDY, TCP SACK option and TCP timestamps option are disabled. By disabling TCP timestamps option, 12 bytes can be saved with every TCP segment.

For IoT-HTTP, we have implemented a simple web server and client to minimize the HTTP headers associated with GET request and ACCEPT message. When a legacy browser is used as the Web client for HTTP and SPDY, the GET request includes the GET method itself, and the details of the operating system, date and time, accepted data types, encoding schemes, encryption schemes, etc., whereas IoT-HTTP GET request adds only the protocol name as HTTP and the IP address of the server. In IoT-HTTP, the ACCEPT message contains only the protocol name HTTP with version, a short server name and the content type. Thus in IoT-HTTP, the Web client has thinned down the headers significantly. In IoT-SPDY, to minimize the headers, spdy-python [13] is used as the SPDY client and spdyd [23] is used as the SPDY server. In addition to reducing header data volume for IoT-HTTP and IoT-SPDY, we employ TCP Fast Open (TFO) [10, 20] with IoT-HTTP and IoT-SPDY.

TCP Fast Open (TFO) [10, 20] is an enhancement to TCP in which data transfer is allowed during TCP's initial handshake, i.e., during the SYN exchange so as to decrease the delay experienced by short TCP transfers. Usually data exchange is allowed after the SYN exchange, i.e., after one Round-trip time (RTT) and this latency is a significant portion of the latency of a short flow. TFO protocol proposes secure data transfer during SYN exchange thereby reducing the latency for HTTP transfers considerably. The security issues that arise due to the data transfer with the SYN is mitigated by a server-generated Fast Open cookie.

Figure 3 gives the TFO protocol overview. The client sends a request to the server for a Fast Open cookie with a regular SYN packet. The cookie generated by the server authenticates the client's IP address. The server sends the Fast Open cookie to the client in the SYN-ACK. The client caches this cookie and uses it for future TCP connections. Both the cookie request from the client and the issue of the cookie by the server are implemented using TCP options. Once the client gets a cookie from the server, it can use this cookie for opening new TCP connections. For new TCP connections, the client sends the SYN with TFO cookie and data. If the server finds that this is a valid cookie it sends SYN-ACK acknowledging both SYN and data, then it delivers the data to the application. The TFO Internet draft [10] states that the server may also send data with the SYN-ACK before the handshake completes if the TCP congestion control allows.

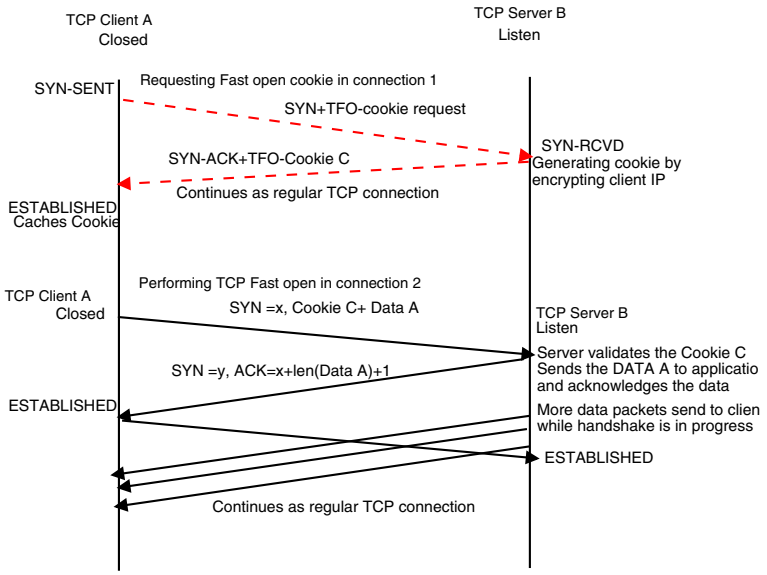


Fig. 3. TCP Fast Open protocol [10, 20]

Typically the server sends the SYN-ACK without any data and starts sending the data in subsequent packets even before the acknowledgement for SYN-ACK from the client arrives at the server. If the cookie is not valid the server drops the data and sends back a SYN-ACK, thus following the usual TCP 3-way handshake (3WHS).

In IoT scenarios where the devices have a small amount of data to be transferred reliably and quite frequently, TFO allows data transfer during the TCP SYN exchange thereby saving up to one RTT compared to standard TCP that requires a 3-way handshake which takes one RTT to complete before the data transfer.

4 Experimental Evaluation of CoAP, HTTP, SPDY, IoT-HTTP and IoT-SPDY

In this section we evaluate the transport services CoAP, HTTP, SPDY, IoT-HTTP and IoT-SPDY that are described in Section 3.

4.1 Experimental Setup

The experiments are carried out over an emulated IoT environment. The initial set of experiments are of request-response type between a client and server that communicate over an emulated link of data rate 20 Kbps, one-way link-delay of 20 ms and Maximum Transmission Unit (MTU) of 128 bytes. The above data rate, delay and MTU size roughly correspond to the data rate, latency and packet size of Zigbee [7] and is

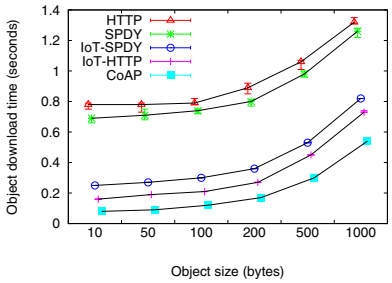


Fig. 4. Object download times for different object sizes and transport services CoAP, HTTP, SPDY, IoT-HTTP and IoT-SPDY

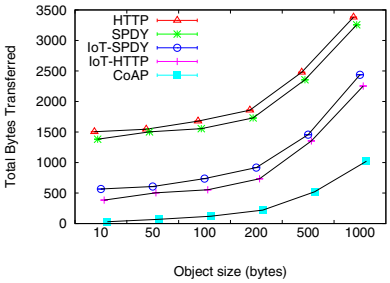


Fig. 5. Total bytes transferred to download objects of different sizes (includes all headers)

emulated using the network emulator, Netem [4]. Netem is a network emulator functionality in Linux that enables to emulate links of different delays, data rates, packet loss and reordering, etc. Netem is controlled by the command line tool 'tc' that allows to show/manipulate traffic control settings. In our experimental set up, shown in Figure 2, the client, server and the network emulator are hosted by x86_64 GNU/Linux machines. The client runs Ubuntu 3.8.0 and the server and emulator run Debian 3.12.2.

In our experiments, Google Chrome and Apache server are used as the client and the server with HTTP and SPDY. As the SPDY server, Apache server enabled with the module mod-spdy [3] is used. As explained in Section 3, we use a simple Web client and Web server, and we enable TCP TFO in the Linux TCP/IP stack. The IoT-SPDY client is spdy-python [13] and the IoT-SPDY server is spdyd [23]. For both IoT-SPDY client and server, TCP is enabled with TFO. Linux kernels (versions from 3.6 onwards) now support TFO. Some small modifications are needed to the client and the server to employ TFO [15]. We use the standard TCP implementation in Linux with the SACK and the timestamp options disabled. The libcoap implementation which is a C implementation of CoAP [2] is used as the CoAP server and the client.

In our test environment, the client fetches from the server a single object of size that varies from 10 bytes to 1000 bytes over an error free channel. The metrics used in the experiments are the object download time and the total bytes transferred in the request-response communication. The object download time is the duration between the time the client initiates the request for the object to the time the last byte of the object arrives at the client. In the cases of HTTP, SPDY, IoT-HTTP and IoT-SPDY, we take the time difference between the SYN request from the client to the arrival of the final byte of the object requested. We calculate these times from the tcpdumps collected at the client while running the experiments.

4.2 Results

Figure 4 shows the object download times for different object sizes when using the different transport services CoAP, HTTP, SPDY, IoT-HTTP, and IoT-SPDY. We observe from Figure 4 that the object download times increase with the size of the object

requested in all the five cases. CoAP has the lowest object download times as the protocol overhead for CoAP is quite small. It can be clearly seen that TFO decreases the object download times for IoT-HTTP and IoT-SPDY. The object download time is decreased also due to the minimized headers for IoT-HTTP and IoT-SPDY. There is at least 50% reduction in object download time for IoT-HTTP and IoT-SPDY compared to HTTP and SPDY. It can be noticed from the Figure 4 that the reduction with smaller object sizes is substantially more. For example, in downloading a 10-byte object, IoT-HTTP achieves around 75 % reduction in object download time compared to HTTP.

Figure 5 shows the total bytes transferred including all headers when objects of different sizes are requested by the client from the server when using the different transport services CoAP, HTTP, SPDY, IoT-HTTP and IoT-SPDY. The amount of total bytes are calculated at the client summing up all IP packets from the first packet sent to the arrival of the last packet carrying the last byte of the object requested. As the protocol overhead is minimum for CoAP it transfers the lowest number of bytes for each object compared to that of HTTP and SPDY. SPDY's header compression accounts for the slight reduction in object download time compared to HTTP. The minimized headers and TFO are responsible for the reduction in total bytes transferred for IoT-HTTP and IoT-SPDY compared to SPDY and HTTP.

4.3 Detailed Analysis of the Results

Next we present a detailed analysis of object download time and bytes transferred for fetching an object of size 10 bytes by describing the message sequence chart (MSC) that shows the different phases of the data transfer. The message sequence chart also shows the packet types and the bytes transferred in this process.

Figure 6 shows the message sequence for HTTP when fetching an object of size 10 bytes from the server. The object download time is 790 ms. The SYN exchange phase takes about 99 ms. The size of SYN segment is 48 bytes that includes the negotiation for maximum segment size (MSS). The GET request of size 588 bytes is sent in four 128 bytes packets plus one 76 bytes packet as the MTU of the link is 128 bytes. The ACCEPT message together with the data is 502 bytes in size and is sent as four packets from the server to the client. A total number of 20 packets including the ACK packets

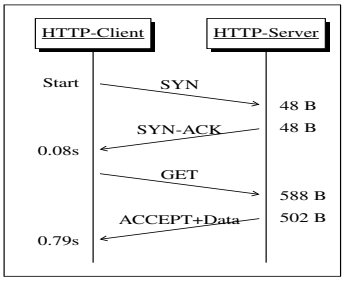


Fig. 6. Message Sequence Chart (MSC) for object download with HTTP

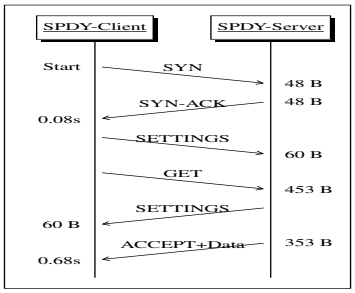


Fig. 7. Message Sequence Chart (MSC) for object download with SPDY

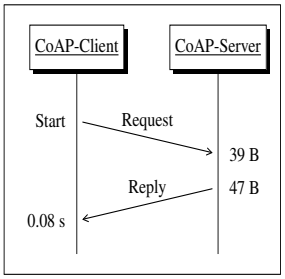


Fig. 8. MSC for object download with CoAP

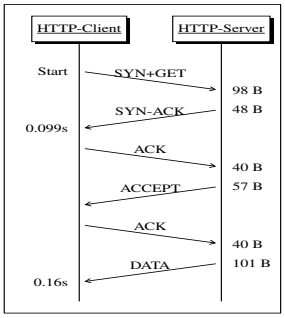


Fig. 9. MSC for object download with IoT-HTTP

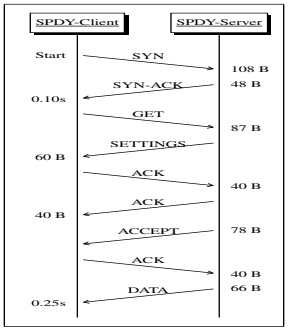


Fig. 10. MSC for object download with IoT-SPDY

are used and 1546 bytes including IP and TCP protocol headers are transferred to fetch an object of size 10 bytes.

Figure 7 shows the message sequence for SPDY when fetching an object of size 10 bytes from the server. It takes 680 ms to fetch an object of size 10 bytes from the server. There are two SETTINGS frames, each 60 bytes in length, transferred both from client to server and from server to client. The GET request is of size 453 bytes and is sent as four packets, while the ACCEPT message together with the data is of 353 bytes and is sent as three packets. The total number of packets transferred in this process is 20 including the ACK packets and a total of 1382 bytes are transferred including IP and TCP protocol headers. SPDY header compression accounts for the reduction of 164 bytes compared to that of HTTP transfer.

Figure 8 shows the message sequence when an object of size 10 bytes is downloaded using CoAP. CoAP takes 80 ms to download the object. CoAP uses only two packets, one request packet sent from the client to the server and one response packet from the server to the client. CoAP transfers 86 bytes including IP and UDP protocol headers.

Figure 9 shows the message sequence for IOT-HTTP to fetch an object of size 10 bytes. The object download time is 160 ms and for this process 384 bytes are transferred in six packets. Compared to the time taken when HTTP or SPDY is used to fetch an object of 10 bytes, this shorter object download time for IoT-HTTP is due to two factors, namely, the use of TFO and the reduced size of the HTTP GET request. With the use of TFO, the GET request is transferred with the SYN segment causing a reduction of one RTT. Google Chrome GET request is of size 588 bytes while in the case of IoT-HTTP, the client sends a single packet of only 98 bytes in size carrying both TCP SYN and HTTP GET request.

Figure 10 shows message sequence for IoT-SPDY taking 250 ms to fetch an object of size 10 bytes. The SYN packet contains SYN, TFO cookie and part of GET request and it has a length of 108 bytes. One SETTINGS frame of 60 bytes in length is transferred from the server to the client. The packet carrying the second part of the GET request is 87 bytes in size and the ACCEPT message with object data is carried in two packets

Table 1. Summary of the analysis of downloading a 10-byte object

Metrics	CoAP	IoT-HTTP	IoT-SPDY	SPDY	HTTP
Object download time	0.08s	0.16s	0.25s	0.68s	0.79s
#Packets	2	6	9	20	20
TotalBytes	86	384	567	1382	1546

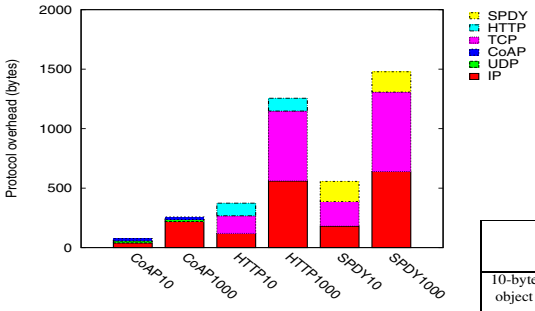
being 78 and 66 bytes in size. The total number of packets transferred in this process is 9 and a total of 567 bytes are transferred including IP and TCP protocol headers.

Table 1 summarises the analysis of downloading a 10-byte object using the five transport services. From the above table, we observe that IoT-HTTP and IoT-SPDY have object download times closer to that of CoAP than HTTP and SPDY. With IoT-HTTP and IoT-SPDY, the object download time reduces from 50% up to 75% compared to that of HTTP and SPDY.

5 Analysis of Protocol Overhead and Discussions on Additional Enhancements

In our experiments reported in Section 4, IoT-HTTP and IoT-SPDY use TFO and minimized HTTP headers. TCP header compression and a reduced set of TCP congestion control may further improve IoT-HTTP and IoT-SPDY in IoT environments. In this section we carry out in detail the analysis of the protocol overhead involved and discuss additional enhancements to improve the performance of IoT-HTTP and IoT-SPDY.

Figure 11 shows another way to compare CoAP, IoT-HTTP and IoT-SPDY based on the protocol overhead associated with them when fetching objects of sizes 10 bytes and



	Transport Services	Protocol overhead					
		IP	UDP	TCP	CoAP	HTTP	SPDY
10-byte object	CoAP	40	16		20		
	HTTP	120		148		106	
	SPDY	180		208			169
1000-byte object	CoAP	220	16		22		
	HTTP	560		588		108	
	SPDY	640		668			170

Fig. 11. Protocol overhead for downloading objects of sizes 10 bytes and 1000 bytes with CoAP, IoT-HTTP and IoT-SPDY

1000 bytes. The x-axis labels CoAP10 and CoAP1000 refer to COAP that downloads 10-byte and 1000-byte objects. Similarly, HTTP10 and HTTP1000 refer to IoT-HTTP and SPDY10 and SPDY1000 refer to IoT-SPDY that download a 10-byte and a 1000-byte objects. In the case of CoAP, out of the total 86 byes transferred to fetch a 10-byte object, IP uses 40 bytes, where UDP and CoAP use 16 bytes and 20 bytes, respectively. The TCP/IP overhead for IoT-HTTP and IoT-SPDY are 268 bytes and 388 bytes, respectively. The HTTP overhead in IoT-HTTP and the SPDY overhead in IoT-SPDY are 106 and 169 bytes, respectively.

When it comes to fetching a 1000-byte object, the object has to be divided into 11 TCP segments for IoT-HTTP and IoT-SPDY as the MTU of the emulated link is 128 bytes. For downloading the 1000-byte object, 28 IP packets are needed for IoT-HTTP including the SYN, SYN-ACK, data and acknowledgements whereas for IoT-SPDY, 32 IP packets are needed. The TCP/IP header overhead for IoT-HTTP and IoT-SPDY are 1148 bytes and 1308 bytes respectively. The HTTP and the SPDY overheads are 108 bytes and 170 bytes for IoT-HTTP and IoT-SPDY respectively. In the case of fetching the 1000-byte object using CoAP, the object is fragmented into 10 fragments and a total of 11 packets are transferred including one CoAP request and 10 CoAP reply data packets. The protocol overhead in this transfer consists of 22 bytes for CoAP, 16 bytes for UDP and 220 bytes for IP. The above analysis shows that HTTP protocol overhead in IoT-HTTP and SPDY protocol overhead in IoT-SPDY are quite large compared to CoAP. It can also be seen that TCP/IP overhead in IoT-HTTP and IoT-SPDY is much larger than UDP/IP overhead in CoAP.

Even though, SPDY has built-in header compression, the above analysis suggests that it is worth investigating the possibility to further reduce the overhead associated with SPDY. The TCP/IP overhead can be reduced significantly by using TCP header compression schemes like RObust Header Compression protocol (ROHC) [21] and thereby reducing the total bytes transferred in fetching an object.

TCP uses a sliding window mechanism that allows to send a number of TCP segments in succession without getting an acknowledgement for each segment. TCP's sliding window mechanism uses 32-bit sequence numbers and 32-bit arithmetic to implement this. In IoT scenarios where data transfer is either push mode or request-reply mode, a small number of TCP segments are sent. So instead of using the TCP sliding window mechanism, we can use TCP as a stop and wait protocol where a new segment is sent only after the sender gets the acknowledgement for the previous segment. This eliminates the processing time and the memory requirements for implementing the sliding window mechanism.

In IoT scenarios where the data transfer is either push mode or request-reply mode, we can go for the simplest congestion control mechanism of retransmission timeout (RTO). The simplified reliability and congestion control mechanisms allow a small footprint for TCP. However in bulk transfer or in continuous data flow mode of data transfer between IoT devices and base station, TCP sliding window mechanisms and congestion control mechanism based on duplicate acknowledgements (dupack) and retransmission timeout [8] can be used.

6 Summary and Future Work

In this paper we evaluated the performance of transport services HTTP, SPDY, CoAP, IoT-HTTP, and IoT-SPDY in an emulated IoT environment. HTTP, SPDY and CoAP are well-known transport services in the Internet whereas IoT-HTTP and IoT-SPDY are adaptations of HTTP and SPDY to make them better suited for IoT environments. The adaptations include minimization of the HTTP/SPDY headers, using TCP Fast Open to lower latency by reducing the TCP connection establishment time, and disabling TCP SACK and timestamps options.

The transport services are compared on the basis of object download time, the total amount of transferred bytes, and the introduced overhead. The experiments are performed in an emulated setup using Netem emulator with Zigbee-like settings and in the context of a simple request-response scenario.

As expected, our experiments show that CoAP has the lowest object download times and the least number of bytes transferred compared to that of HTTP and SPDY due to the header overhead in HTTP and the TCP connection establishment. With our proposed schemes, IoT-HTTP and IoT-SPDY, we observe that IoT-HTTP and IoT-SPDY have around 50% shorter object download times and smaller number of bytes transferred compared to HTTP and SPDY. As SPDY has built-in header compression, we suggest IoT-SPDY as a good candidate for IoT transport service. In addition, we intend to investigate the possibility to further reduce the overhead associated with SPDY and TCP. Possible targets for additional enhancements include improving SPDY's header compression scheme and RObust Header Compression (ROHC) scheme with TCP.

Acknowledgements. This work was supported by TEKES as part of the Internet of Things DIGILE (Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT and digital business). We thank the anonymous reviewers for their constructive comments on our paper.

References

[1] ICT SHOK IoT programme, `http://www.internetofthings.fi/`
[2] libcoap: C-Implementation of CoAP, `http://sourceforge.net/projects/libcoap`
[3] mod_spdy: Apache spdy module, `http://code.google.com/p/mod-spdy`
[4] Netem: Network Emulator, `http://manpages.ubuntu.com/manpages/raring/en/man8/tc-netem.8.html`
[5] SPDY: An Experimental Protocol for a Faster Web, `http://www.chromium.org/spdy/spdy-whitepaper`
[6] TinyOS, `http://www.tinyos.net/`
[7] Zigbee, `http://en.wikipedia.org/wiki/ZigBee`
[8] Allman, M., Paxson, V., Blanton, E.: TCP Congestion Control. Internet RFCs, RFC 5681 (September 2009) ISSN 2070-1721
[9] Belshe, M., Peon, R.: SPDY Protocol. Internet draft "draft-mbelshe-httpbis-spdy-00", Work in progress (February 2012)
[10] Cheng, Y., Chu, J., Radhakrishnan, S., Jain, A.: TCP Fast Open. Internet draft "draft-cheng-tcpm-fastopen-09.txt", Work in progress (June 2014)

[11] Colitti, W., Steenhaut, K., De Caro, N., Buta, B., Dobrota, V.: Evaluation of Constrained Application Protocol for Wireless Sensor Networks. In: Proceedings of the 18th IEEE Workshop on Local and Metropolitan Area Networks (LANMAN), pp. 1–6 (2011)
[12] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: HyperText Transfer Protocol - HTTP/1.1. Internet RFCs, RFC 2616 (June 1999) ISSN 2070-1721
[13] Gupta, A.: spdy-python: A SPDY Library in Python, https://github.com/ashish-gupta-/spdy-python/
[14] Ishaq, I., Carels, D., Teklemariam, G.K., Hoebeke, J., Abeele, F., Poorter, E., Moerman, I., Demeester, P.: IETF Standardization in the Field of the Internet of Things (IoT): A Survey. Journal of Sensor and Actuator Networks 2(2), 235–287 (2013)
[15] Kerrisk, M.: TCP Fast Open: Expediting Web Services, http://lwn.net/Articles/508865/
[16] Kuladinithi, K., Bergmann, O., Pötsch, T., Becker, M., Görg, C.: Implementation of CoAP and its Application in Transport Logistics. In: Proceedings of the workshop on Extending the Internet to Low power and Lossy Networks (IP+SN) (2011)
[17] Levä, T., Mazhelis, O., Suomi, H.: Comparing the cost-efficiency of CoAP and HTTP in Web of Things applications. Decision Support Systems 63, 23–38 (2014)
[18] Ludovici, A., Moreno, P., Calveras, A.: TinyCoAP: A Novel Constrained Application Protocol (CoAP) Implementation for Embedding RESTful Web Services in Wireless Sensor Networks Based on TinyOS. Journal of Sensor and Actuator Networks 2(2), 288–315 (2013)
[19] Postel, J.: Transmission Control Protocol. Internet RFCs, RFC 793 (September 1981) ISSN 2070-1721
[20] Radhakrishnan, S., Cheng, Y., Jerry Chu, H.K., Jain, A., Raghavan, B.: TCP Fast Open. In: Proceedings of the Seventh Conference on Emerging Networking EXperiments and Technologies, CoNEXT 2011, pp. 20:1–20:12 (2011)
[21] Sandlund, K., Pelletier, G., Jonsson, L.-E.: RObust Header Compression (ROHC) Framework. Internet RFCs, RFC 5795 (March 2010) ISSN 2070-1721
[22] Shelby, Z., Hartke, K., Bormann, C.: The Constrained Application Protocol (CoAP). Internet RFCs, RFC 7252 (June 2014) ISSN 2070-1721
[23] Tsujikawa, T.: spdylay:The experimental SPDY protocol version 2, 3 and 3.1 implementation in C, http://tatsuhiro-t.github.io/spdylay/