Análise de Desempenho de Brokers MQTT em Sistema de Baixo Custo

Andrei B. B. Torres, Atslands R. Rocha, José Neuman de Souza

[1]Grupo de Redes de Computadores, Engenharia de Software e Sistemas (GREat)
Universidade Federal do Ceará (UFC) – Fortaleza – CE – Brazil

andreibosco@great.ufc.br, atslands@ufc.br, neuman@ieee.org

Abstract. *This paper presents a performance analysis (CPU usage, memory consumption and message throughput) of MQTT brokers in a low-cost hardware, the Raspberry Pi 2 Model B. The objectives of the analysis are to ascertain which MQTT broker implementation is best suited to the limitations of the hardware and to verify if the Raspberry Pi 2 is actually able to function as a gateway in a sensor and actuator network for the Internet of Things (IoT). The results showed that the Raspberry Pi 2 can handle large numbers of connections and that the implementation in Erlang (eMQTT) obtained the best results in data throughput, while the implementation in C (Mosquitto) obtained the lowest CPU load and memory consumption.*

Resumo. *Este artigo apresenta uma análise de desempenho (uso de CPU, consumo de memória e envio de mensagens) de* brokers *MQTT em um hardware de baixo custo, o Raspberry Pi 2 Modelo B. Os objetivos da análise são averiguar qual implementação de* broker *MQTT é a mais adequada às limitações do hardware e se o Raspberry Pi 2 é realmente capaz de funcionar como* gateway *de uma rede de sensores e atuadores para a Internet das Coisas (IoT). Os resultados mostraram que o Raspberry Pi 2 consegue lidar com grandes números de conexões, sendo a implementação em Erlang (eMQTT) a que obteve melhor resultado em vazão de dados, enquanto a implementação em C (Mosquitto) apresentou a menor carga de processamento e consumo de memória.*

1. Introdução

A Internet das Coisas (IoT - *Internet of Things*) é um conceito de tornar a Internet e a comunicação entre objetos ou 'coisas' pervasiva, onde eles são capazes de interagir e de cooperar entre si para alcançar um objetivo [Singh et al. 2014]. Por meio de sensores e atuadores embutidos em tais "coisas inteligentes" seremos capazes de receber dados e de controlá-los remotamente. Algumas áreas com grande potencial para aplicação da IoT são: domótica, saúde, cidades inteligentes, escritórios, comércio, uso militar, dentre outros. A IoT possui um potencial comercial entre 3,5 a 11,1 trilhões de dólares, podendo alcançar entre 25 a 50 bilhões de dispositivos até 2025 [Manyika et al. 2015].

Para tornar a IoT viável e permitir a conexão de centenas de milhares de coisas, é necessário que tais coisas sejam de baixo custo, o que implica baixa capacidade de processamento, armazenamento e comunicação. Deve-se considerar que, ao contrário de *smartphones*, em que o usuário possui em média uma unidade, as coisas inteligentes irão permear o ambiente. Dependendo da limitação de recursos, em alguns cenários torna-se

necessário um dispositivo que funcione como *gateway* de acesso à Internet, repassando os dados das coisas para o usuário final ou até mesmo para outras coisas. As plataformas de hardware de baixo custo são fortes candidatas para preencher este espaço de *gateway* na IoT, e o Raspberry Pi[1] é um dos mais famosos dentre elas. O projeto Raspberry Pi é um projeto educacional britânico cujo foco era criar um hardware simples e acessível para crianças, além de motivar o aprendizado de eletrônica e computação. Outras plataformas de baixo custo são: Beaglebone Black[2], Banana Pi[3], MinnowBoard MAX[4], dentre outras. O Raspberry Pi 2 foi escolhido para este trabalho, dentre as plataformas citadas, devido a facilidade de compra do equipamento no Brasil, menor preço em comparação às plataformas similares[5] e a existência de uma ampla gama de acessórios e componentes.

Os protocolos para a comunicação entre os componentes da IoT devem lidar com fatores como baixa largura de banda, alta latência e instabilidade da comunicação. Alguns protocolos foram criados exatamente para lidar com tais fatores: CoAP (*Constrained Application Protocol* [Shelby et al. 2014]), MQTT (*Message Queuing Telemetry Transport* [MQTT Version 3.1.1 2014]), WAMP (*Web Application Messaging Protocol* [WAMP Draft 2 2015]), dentre outros. Neste artigo, o protocolo MQTT foi adotado para a realização dos experimentos, devido à ampla gama de *brokers* implementados em diferentes linguagens e à crescente adoção no mercado. A Seção 2 aborda o protocolo MQTT mais detalhadamente e o motivo de ter sido escolhido para este experimento.

Os objetivos deste artigo são averiguar a viabilidade de uso do Raspberry Pi 2 modelo B como *gateway* de uma rede de sensores e atuadores para IoT, tendo que lidar com um grande número de clientes e de mensagens sendo enviadas em modo QoS 0[6], além de determinar qual *broker* MQTT é a implementação mais adequada a este hardware.

O restante deste trabalho está dividido da seguinte forma: a Seção 2 apresenta o protocolo MQTT e os *brokers* estudados. A Seção 3 apresenta trabalhos relacionados à análise de desempenho de protocolos e de dispositivos voltados à comunicação machine-to-machine (M2M) e/ou à IoT. A Seção 4 aborda a análise de desempenho em si, apresentando os métodos, os materiais e a metodologia utilizada. A Seção 5 apresenta e analisa os resultados encontrados, e a Seção 6 apresenta uma análise geral com base nos resultados discutidos e possíveis trabalhos futuros.

2. Protocolos de Comunicação para IoT

A IoT ainda está em rápida evolução e está em aberto qual protocolo de comunicação se tornará o padrão do mercado. Dentre os diversos protocolos existentes (MQTT, CoAP, XMPP, SNMP, WAMP) que podem ser usados para IoT, este artigo focou no protocolo MQTT, o qual tem sido largamente adotado por diversas empresas, seja com foco no uso na IoT ou apenas como um protocolo de comunicação. Um exemplo fora do contexto de IoT é a adoção do MQTT pelo Facebook como protocolo de comunicação de

[1] `https://www.raspberrypi.org/` - Acesso em 06/04/2016.

[2] `http://beagleboard.org/black` - Acesso em 06/04/2016.

[3] `http://www.bananapi.org/` - Acesso em 06/04/2016.

[4] `http://www.minnowboard.org/meet-minnowboard-max/` - Acesso em 06/04/2016.

[5] De acordo com pesquisa de preço realizada online pelos autores em 15/05/2016.

[6] MQTT possui 3 níveis de garantia de entrega de mensagem (*QoS - Quality of Service*), que vai de 0 (nenhuma garantia de entrega) a 2 (garantia de entrega sem duplicidade)

seu sistema de *instant messaging* [Zhang 2011], e dentro do contexto de IoT, a Amazon adotou MQTT, HTTP e Websockets como protocolos padrões em sua plataforma AWS IoT [Amazon Web Services 2016]. Segundo [Skerrett 2015], diretor de marketing da fundação Eclipse, "me parece que o MQTT se tornou o padrão a ser suportado por qualquer provedor sério de soluções para IoT"[7]. Portanto, a análise de implementações do protocolo MQTT é bastante relevante no cenário atual.

2.1. MQTT

Criado em 1999 pela IBM, MQTT (*MQ Telemetry Transport*), é um protocolo aberto de mensagens projetado para comunicação M2M, na qual deve lidar com alta latência, instabilidade na comunicação e baixa largura de banda. O protocolo MQTT foi padronizado pelo OASIS em 2013 e atualmente está na versão 3.1.1 [MQTT Version 3.1.1 2014], sendo livre de *royalties* desde 2010.

O protocolo MQTT adota o protocolo TCP e o padrão de mensagens *publisher/-subscriber* (publicador/assinante), onde todos os dados são enviados para um intermediário, chamado *broker*, que se encarrega de enviar as mensagens aos destinatários corretos. Esta estrutura permite desacoplar o produtor do cliente, assim, apenas o endereço do *broker* precisa ser conhecido, possibilitando a comunicação de um para um (*one-to-one* - ver Figura 1), um para muitos (*one-to-many* - ver Figura 1) ou muitos para muitos (*many-to-many* - ver Figura 1). Já no quesito de segurança, até a versão 3.1.1, MQTT não implementa qualquer tipo de criptografia (SSL pode ser utilizado independentemente), existindo apenas um método de autenticação com nome de usuário e senha.

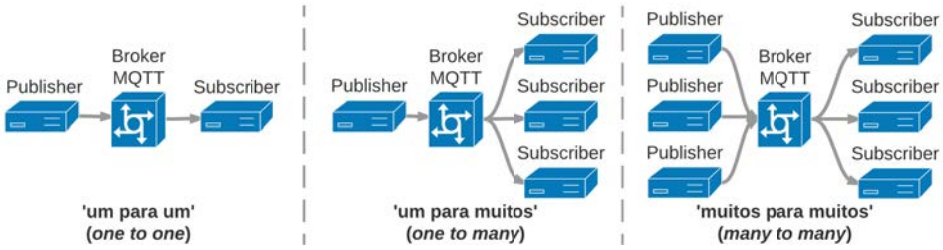

Figura 1. Tipos de distribuição de mensagem suportados pelo protocolo MQTT. Fonte: os autores.

3. Trabalhos relacionados

Existem outros trabalhos que focam na análise de desempenho, seja de protocolos e serviços ou de hardware e dispositivos, porém nenhum dos artigos encontrados através de um processo de revisão sistemática foca especificamente no uso de hardware de baixo custo com MQTT. A metodologia da revisão sistemática está detalhada na Tabela 1.

O trabalho mais similar ao aqui apresentado foi realizado por [Scalagent 2015], que realiza uma análise comparativa entre diversos *brokers* MQTT, porém com um escopo diferente. O cenário é montado como uma comunicação de muitos-para-muitos (chegando a um máximo de 100 *subscribers* e 100.000 *publishers*), com foco na escalabilidade dos *brokers*, latência no tempo de envio das mensagens e a vazão de entrega, enquanto o presente artigo objetiva principalmente o desempenho do hardware em lidar

[7]Tradução livre dos autores.

Tabela 1. Metodologia da Revisão Sistemática

String de busca	((”mqtt”OR ”raspberry pi”) AND (”performance”OR ”analysis”OR ”simulation”)) AND IoT
Fonte de dados	ACM Digital Library, IEEE Digital Library, Science@Direct, Scopus
Critérios	Inclusão: análise de desempenho Exclusão: survey, não lidar diretamente com MQTT ou Raspberry
Período	2012 - atual
Resultado	113 artigos, 4 relevantes, nenhum com foco em hardware de baixo custo e MQTT

com um elevado número de *subscribers* (máximo de 10.000). O hardware utilizado foi um Intel Core 2 Duo 3.00 GHz com 4 GB de memória, muito superior ao Raspberry Pi 2 utilizado neste artigo, mas também muito mais caro.

3.1. Benchmark de hardware

O artigo de [Kruger and Hancke 2014] realiza uma análise comparativa de diversos dispositivos *off-the-shelf* (“disponíveis na prateleira” - soluções completas, autocontidas): Raspberry Pi 1, BeagleBone, BeagleBone Black. Os objetivos do artigo foram determinar se o uso de soluções de armazenamento de maior desempenho causam alguma influência notável e avaliar os dispositivos como *gateways* CoAP, onde o foco da análise foi o tempo de latência das requisições. Os resultados apontaram que utilizar armazenamento de maior desempenho e mais caro não causa impacto significativo, e o fator mais importante na escolha da solução é o processador.

O mesmo autor possui um artigo mais recente, [Kruger et al. 2015], onde estuda o uso do Raspberry Pi como *gateway* de uma rede de sensores sem fio, utilizando, novamente, dispositivos *off-the-shelf* para monitoramento do consumo de água. O autor conclui que a utilização de dispositivos *off-the-shelf* é viável, reduzindo consideravelmente o tempo do ciclo de desenvolvimento de produtos.

3.2. Benchmark de software

Outros artigos relevantes, mas com foco no software, são os apresentados por [Thangavel et al. 2014], [Kovatsch et al. 2014] e [Collina et al. 2014]. O primeiro realiza uma análise comparativa entre MQTT e CoAP através de um *middleware* desenvolvido pelos autores, utilizando atraso e consumo de banda como métricas. Os resultados experimentais apontam que o desempenho depende das condições da rede, em que mensagens MQTT obtiveram menor atraso em uma rede com pouca perda de pacotes, e maior atraso em rede com elevada perda de pacotes. Também foi identificado que CoAP gera menos tráfego adicional do que o MQTT para garantir a entrega de mensagens.

O segundo artigo apresenta um *benchmark* do *framework* Californium, com foco na criação de um servidor CoAP na nuvem que seja escalável e capaz de lidar com um número massivo de clientes (acima de 100k). O *framework* CoAP Californium obtém uma vazão de dados de 33 a 64 vezes superior à do HTTP.

E o terceiro artigo realiza uma análise de desempenho quantitativa dos protocolos MQTT e CoAP, e analisa o comportamento dos protocolos em diversas situações de rede

(perda de pacote, atraso, dentre outros fatores). No estudo, o MQTT apresentou a melhor vazão de dados, enquanto o CoAP apresentou a menor latência em caso de baixo tráfego e baixa probabilidade de perda de pacote.

4. Experimentos

Nesta Seção são descritos os materiais utilizados e o design experimental dos testes. O objetivo dos experimentos foi avaliar o funcionamento e desempenho do Raspberry Pi 2 e dos *brokers* MQTT em um cenário de alta carga, simulando seu uso como um *gateway* acessível pela internet, respondendo a milhares de requisições de leituras realizadas simultaneamente. Um exemplo de cenário prático é o de um sensor público, como um sensor de temperatura ou de poluição, acessível via web ou por um aplicativo móvel.

4.1. Ambiente

4.1.1. Hardware

Um Raspberry Pi 2 modelo B (especificações na Tabela 2) e uma máquina *desktop* foram utilizados nos experimentos. No *desktop* foi executado o aplicativo Apache JMeter que realizou a carga simulada de *subscribers* (esse processo é explicado na Subseção 4.2).

Tabela 2. Especificações do Raspberry Pi 2 modelo B

Processador	ARM Cortex A7 900MHz Quad-core	**Disco**	Micro SDHC 16GB UHS Class 1
Memória	1 GB	**S.O.**	Raspbian Wheezy 2015-05
Ethernet	10/100 Mbps	**Alimentação**	Micro USB 5V/1.8A

4.1.2. Software: *Brokers* MQTT

Uma ampla gama de servidores MQTT estão disponíveis, tanto em código aberto quanto em software proprietário[8]. Para a execução dos experimentos relatados neste artigo, foram selecionados apenas *brokers* de código aberto e de diversas linguagens de programação (C, Java, Javascript, Erlang) para aferir qual obteria um melhor desempenho (ver Tabela 3).

Tabela 3. Brokers MQTT analisados

Broker	Versão	Linguagem	Implementação	Extras
Mosquitto[9]	1.4.8	C	MQTT 3.1.1	–
Apache ActiveMQ Apollo[10]	1.7.1	Java	MQTT 3.1	Suporte a STOMP, AMQP, MQTT, OpenWire
Mosca[11]	1.1.2	Javascript (node.js)	MQTT 3.1.1	Sem suporte a QoS 2
eMQTT[12]	0.17.0-beta	Erlang	MQTT 3.1.1	Foco em clusterização
Ponte[13]	0.0.16	Javascript (node.js)	Não informado	Realiza ponte entre HTTP (REST), MQTT e CoAP.

[8]Listagem mantida pela comunidade disponível em `https://github.com/mqtt/mqtt.github.io/wiki/servers` - Acesso em 06/04/2016.

[9]http://mosquitto.org/ - Acesso em 13/05/2016.

4.2. Design Experimental

Para testar o desempenho do hardware foi realizada uma carga de trabalho progressiva, onde o número de clientes (*subscribers*) é o fator de controle, de acordo com a seguinte sequência de passos (ver Figura 2):

1. Iniciar captura de dados;
2. Aguardar 60 segundos;
3. Iniciar 200 conexões de *subscribers* a cada 30 segundos;
4. ao atingir a carga máxima de 10 mil *subscribers*, mantê-la durante 180 segundos;
5. Encerrar 25 conexões a cada 1 segundo;
6. Aguardar 5 minutos após encerrar a última conexão;
7. Encerrar captura de dados.

O controle do processo de conexões, conforme apresentado na Figura 2, foi realizado por meio do software Apache JMeter[14], em conjunto com os plugins *Stepping Thread Group*[15], que permite definir uma carga progressiva em degraus, e *mqtt-jmeter*[16], que habilita o JMeter a utilizar o protocolo MQTT.

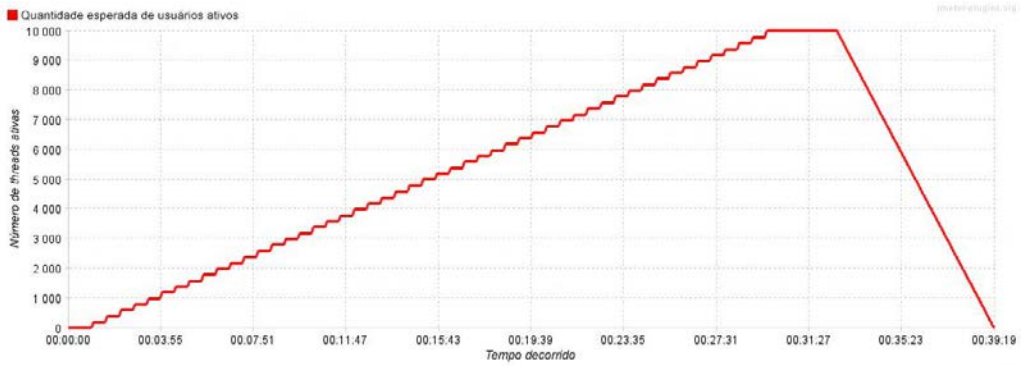

Figura 2. Gráfico da carga de trabalho progressiva adotada

A coleta de dados de desempenho foi realizada por meio do software *collectl*, que permite a coleta de diversas métricas, onde as seguintes foram elencadas:

- CPU: porcentagem de uso do processo (e seus filhos);
- Memória: quantidade de uso do processo (e seus filhos).

Para a coleta de dados de rede, o software *tshark*[17] foi utilizado, gerando um arquivo que pode ser analisado posteriormente utilizando o software *wireshark*[18]. Já o papel de *publisher* foi realizado por um *script* bash, executado no próprio Raspberry Pi 2, enviando a mensagem 'hello' em modo QoS 0 (envio sem confirmação de recebimento), sem retenção, a cada 1 segundo. Toda a estrutura de comunicação entre os componentes do experimento está explicitada na Figura 3.

[10]https://activemq.apache.org/apollo/ - Acesso em 13/05/2016.
[11]http://www.mosca.io/ - Acesso em 13/05/2016.
[12]http://emqtt.io/ - Acesso em 13/05/2016.
[13]https://github.com/eclipse/ponte - Acesso em 13/05/2016.
[14]http://jmeter.apache.org/ - Acesso em 06/04/2016.
[15]http://jmeter-plugins.org/wiki/SteppingThreadGroup/ - Acesso em 06/04/2016.
[16]https://github.com/tuanhiep/mqtt-jmeter - Acesso em 06/04/2016.
[17]https://www.wireshark.org/docs/man-pages/tshark.html - Acesso em 06/04/2016.
[18]https://www.wireshark.org/ - Acesso em 06/04/2016.

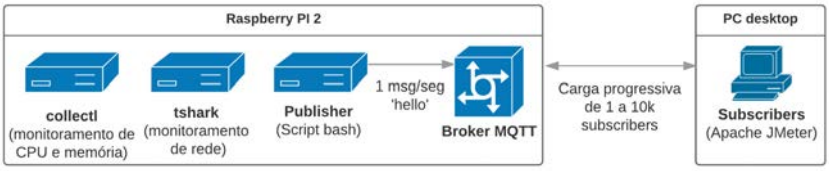

Figura 3. Diagrama de comunicação e monitoramento

Todos os experimentos foram executados 10 vezes após a reinicialização do sistema (*warm boot*), com comandos executados via SSH e na seguinte sequência:

1. Iniciar o *broker* MQTT;
2. Iniciar *script* de publicação de mensagens (*publisher*).
3. Iniciar coleta de dados de rede (*tshark*);
4. Iniciar coleta de dados de desempenho (*collectl*).

Dentre os *brokers* elencados para o experimento, alguns necessitaram de ajustes nas configurações para a realização dos experimentos[19]:

- *Brokers* Mosca e Ponte: aumentar o limite de arquivos abertos;
- *Broker* eMQTT: aumentar o limite máximo de processos;
- *Broker* Apollo: aumentar limite máximo de conexões e ajuste de uso de memória.

A Tabela 4 apresenta resumidamente o design experimental dos testes, incluindo as métricas, fatores e carga de trabalho.

Tabela 4. Critérios para análise de desempenho

Critérios	**Descrição**
Sujeitos	Raspberry Pi 2 e *brokers* MQTT
Delineamento	Análise comparativa de *brokers* utilizando uma carga progressiva de clientes
Métricas	Uso de CPU, Consumo de memória, Mensagens enviadas
Parâmetros	CPU e memória
Fatores	Número de conexões (*subscribers*)
Técnica de avaliação	Medição
Iterações	10 iterações por *broker*
Carga de trabalho	Carga progressiva em escada com 200 novas conexões a cada 30s. Ao atingir a carga máxima de 10 mil conexões ela é mantida por 180s, e então é iniciado o processo de desconexão de 25 conexões por segundo.
Análise dos dados	Interpretação dos resultados das médias de uso de CPU, memórias e total de mensagens enviadas.

5. Resultados

Com exceção do Apollo, todos os *brokers* conseguiram completar com sucesso as 10 iterações de experimento. Dentre os 5 *brokers* testados, a expectativa era de que aqueles

[19]Código disponível online em `https://gist.github.com/andreibosco/b9e70757ed6315d4ea57`

que foram programados em *Javascript via node.js* (uma linguagem interpretada de alto nível) teriam pior desempenho (maior carga de processamento e uso de memória) do que aqueles programados em C, Java ou Erlang (linguagens compiladas). Porém esta hipótese foi refutada, em parte, pelos experimentos.

A Figura 4 apresenta um resumo dos resultados obtidos através de uma média das 10 iterações. Nela é possível notar a similaridade de desempenho dos *brokers* Mosca, Ponte e Mosquitto, e que o Apollo apresenta um resultado excêntrico. Nas próximas subseções cada tópico dos resultados será abordado de maneira mais aprofundada.

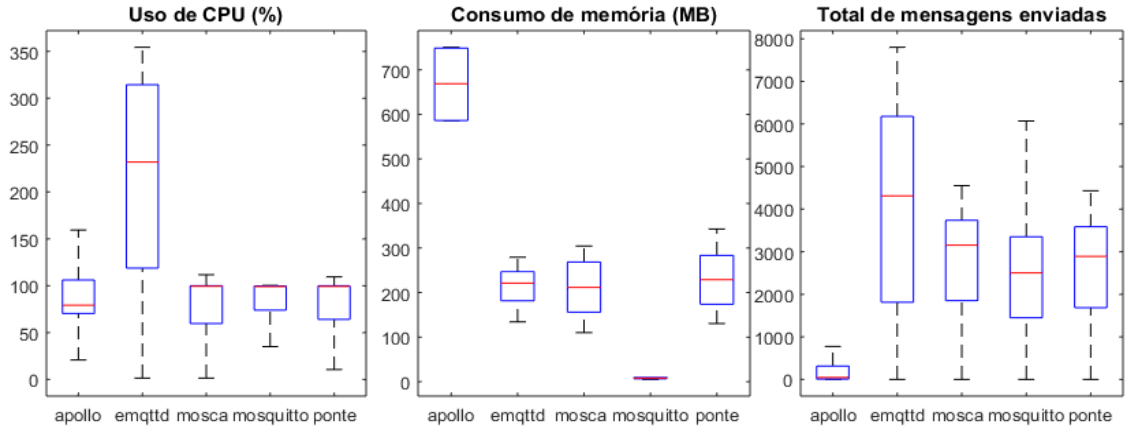

Figura 4. Resumo dos resultados obtidos

5.1. CPU

Conforme apresentado na Figura 5, os *brokers* Mosquitto (C), Mosca (node.js) e Ponte (node.js) apresentaram a menor carga de processamento. Ponte é um *broker* de múltiplos protocolos de comunicação (CoAP, MQTT e HTTP REST), e sua implementação de MQTT é baseado no Mosca, portanto, a similaridade de desempenho já era esperada. O fato mais marcante é o Mosca e Ponte terem obtido desempenho equivalente ao Mosquitto, que é utilizado como *broker* MQTT de referência.

Representando a linguagem Erlang temos o *broker* eMQTT, cujo foco é em clusterização e escalabilidade[20]. Talvez devido a esse foco, sua arquitetura deve ter sido criada para utilização em máquinas mais robustas do que o Raspberry Pi 2. O *broker* eMQTT conseguiu concluir os experimentos com sucesso, porém obteve a maior carga de processamento.

Já para o *broker* programado em Java, os resultados foram problemáticos. O Apollo não chegou a concluir todos os experimentos, mesmo após diversas tentativas de ajustes nas configurações do *broker*. Na Subseção 5.2, fica claro o momento e o motivo de travamento do *broker*. No *log* de comunicação entre o *broker* e o JMeter (representando os *subscribers*), foi reportada uma série de erros *Uncaught exception: java.lang.NullPointerException*. Portanto, apesar de algumas iterações de experimento terem concluído com sucesso, seu funcionamento foi errático, e na Subseção 5.3 fica claro o problema no envio de mensagens.

[20]A linguagem Erlang foca em multiparalelismo e aplicações distribuídas. Mais informações: `http://www.erlang.org/download/armstrong_thesis_2003.pdf` - Acesso em 06/04/2016.

5.2. Memória

A Figura 6 apresenta o resultado do consumo de memória, e as implementações em node.js novamente obtiveram desempenho similar, tendo o *broker* Ponte uma leve desvantagem, talvez pelo projeto não ser desenvolvido tão ativamente[21]. O *broker* Mosca obteve o segundo menor consumo de memória, chegando a um pico de ~275MB.

No caso do Apollo, ele obteve o maior consumo de memória, chegando a travar em algumas iterações reportando memória insuficiente para execução (*There is insufficient memory for the Java Runtime Environment to continue*). Por padrão, o Apollo inicia com um parâmetro que indica 1GB de memória como o máximo alocável para a máquina virtual Java (JVM - *Java Virtual Machine)*, que é exatamente a quantidade existente no Raspberry Pi 2, sendo assim um valor inadequado para a plataforma. Tentou-se repetir os experimentos com um valor menor (700MB), porém o comportamento instável persistiu.

Contrastando com sua carga de processamento, o eMQTT obteve o terceiro menor consumo de memória, apresentando um desempenho próximo ao apresentado pelos *brokers* Ponte e Mosca. Um fato curioso é o aumento do consumo de memória no término do período da carga máxima e início do processo de desconexão dos *subscribers*.

Já o resultado obtido pelo Mosquitto foi surpreendente, chegando a um pico de apenas 9.5MB. Era esperado que a implementação em C obtivesse melhor resultado, porém não era esperado um valor tão díspar ao segundo colocado, eMQTT. Interessante notar na Figura 7 como o consumo de memória do Mosquitto representa claramente a carga progressiva representada na Figura 2.

5.3. Rede

Na Figura 8 são apresentados os resultados de vazão das mensagens (número de pacotes entregues por minuto). Os *brokers* Mosca e Ponte obtiveram não apenas bom desempenho na carga de uso do processador, baixo consumo de memória, mas também bons índices de entrega de mensagens. Mosca e Ponte estão tecnicamente empatados, novamente lembrando que ambos possuem uma base de código similar.

Apesar do Mosquitto ter apresentado um excelente resultado no quesito consumo de memória, ele aparenta ter alcançado um limite de conexões por volta de 15 minutos, onde houve um claro declínio no envio de mensagens. Em comparação, ele obteve apenas 68% do índice de entrega de mensagens do Mosca, o melhor colocado.

O eMQTT apresentou a maior carga de processamento dentre os *brokers* analisados ao alcançar o primeiro lugar em quantidade de mensagens entregues, conseguindo quase o dobro da vazão em comparação ao Mosca e Ponte. Em último lugar temos o Apollo. A quantidade ínfima de mensagens entregues, em comparação com os outros *brokers*, confirma o comportamento errático apontado pelos dados do desempenho de memória e processamento. Devido à geração do gráfico através de mediana e suavização das curvas, o Apollo sequer chega a ser representado.

Quase todos os *brokers*, com exceção do eMQTT, aparentam ter alcançado um limite após aproximadamente 15 minutos de experimento, onde a taxa de envio de mensagens estagnou, chegando até a decair no caso do Mosquitto. Esse tempo equivale a cerca de 4 mil conexões, e é necessário investigar o que causou esse limite.

[21] Projeto Ponte teve 8 *commits* desde o início de 2016, enquanto o projeto Mosca teve 38.

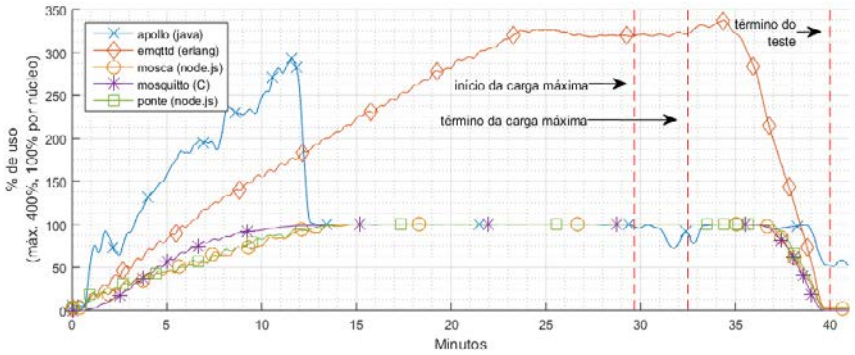

Figura 5. Carga de uso de processador

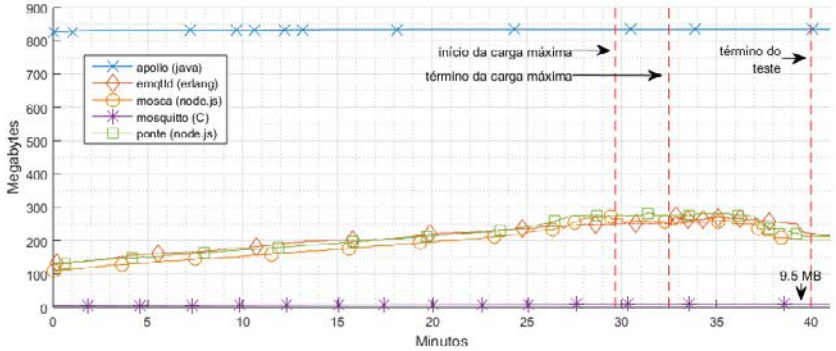

Figura 6. Consumo de memória

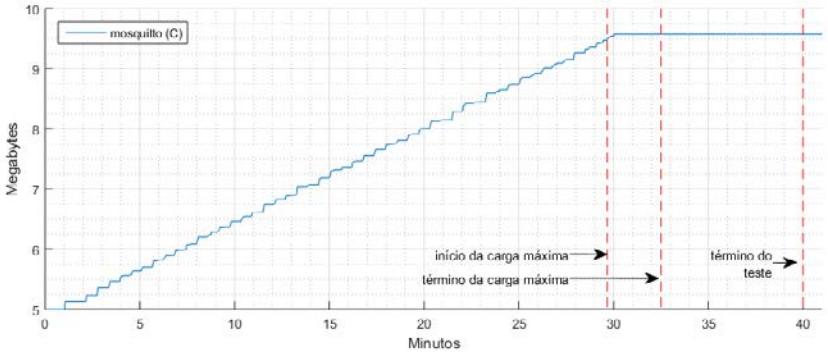

Figura 7. Consumo de memória do *broker* Mosquitto

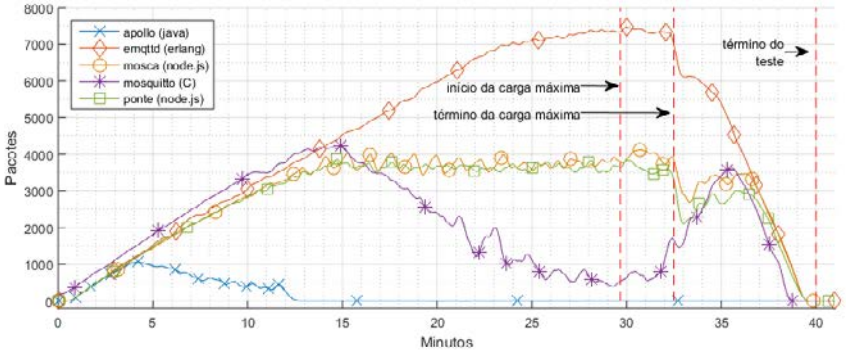

Figura 8. Pacotes por minuto

6. Conclusão

Foi realizada uma avalização do uso de CPU e consumo de memória de *brokers* MQTT em um hardware de baixo custo, Raspberry Pi 2 Modelo B. Os objetivos foram averiguar a viabilidade de uso de tal hardware como *gateway* para IoT e de determinar qual *broker* MQTT possui a implementação mais adequada a este hardware. Essa avaliação foi realizada por meio de um experimento de comunicação "um para muitos", com 1 *publisher* se comunicando com 10 mil *subscribers* através de um *broker* instalado no Raspberry Pi 2.

A hipótese inicial de que a implementação em C, Mosquitto, teria o melhor desempenho (menor uso de CPU e menor consumo de memória) foi confirmada pelos resultados obtidos nos experimentos, apresentando consumo de memória em uma ordem de grandeza abaixo do segundo melhor colocado, a implementação em node.js, Mosca. No entanto, melhor desempenho não implicou em melhor vazão de dados, com o Mosquitto atingindo um limite em aproximadamente 4 mil conexões e passando a se comportar erraticamente. Os *brokers* Mosca e Ponte também apresentaram um limite de aproximadamente 4 mil conexões, porém mantiveram uma vazão de dados constante. O *broker* Apollo (Java) não chegou a concluir todos os experimentos por falha de falta de memória, sendo o pior em todos os quesitos analisados. Já a implementação em Erlang, o eMQTT, demonstrou um *trade-off*, com o maior uso de CPU e o terceiro menor consumo de memória (muito próximo ao apresentado pelo Mosca), porém com a maior vazão de mensagens.

Quanto ao hardware, o Raspberry Pi 2 provou ser um hardware capaz de lidar com altos números de requisições, sendo a baixa quantidade de memória o ponto frágil do hardware. Quanto ao software, o *broker* Mosquitto é o mais recomendado caso o número de conexões seja inferior a 4 mil e a necessidade de baixo consumo de memória e baixa carga de processamento sejam primordiais. Porém, caso a necessidade seja alta vazão e grande número de conexões, e onde carga de processamento, e, consequentemente, consumo energético, não seja um problema, o eMQTT é a melhor opção.

Como trabalhos futuros, pretende-se realizar experimentos de comunicação do tipo "muitos para muitos", analisar os motivos que causaram a barreira de quatro mil conexões encontrada nos experimentos, e a comparação do MQTT com outros protocolos, como CoAP, HTTP/2 e WAMP.

Referências

Amazon Web Services (2016). AWS IoT. https://aws.amazon.com/pt/iot/. Acesso em 17/03/2016.

Collina, M., Bartolucci, M., Vanelli-Coralli, A., and Corazza, G. E. (2014). Internet of Things application layer protocol analysis over error and delay prone links. *2014 7th Advanced Satellite Multimedia Systems Conference and the 13th Signal Processing for Space Communications Workshop (ASMS/SPSC)*, pages 398–404.

Kovatsch, M., Lanter, M., and Shelby, Z. (2014). Californium: Scalable cloud services for the Internet of Things with CoAP. In *2014 International Conference on the Internet of Things (IOT)*, pages 1–6. IEEE.

Kruger, C. P., Abu-Mahfouz, A. M., and Hancke, G. P. (2015). Rapid prototyping of a wireless sensor network gateway for the internet of things using off-the-shelf compo-

nents. In *2015 IEEE International Conference on Industrial Technology (ICIT)*, pages 1926–1931. IEEE.

Kruger, C. P. and Hancke, G. P. (2014). Benchmarking Internet of things devices. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*, pages 611–616. IEEE.

Manyika, J., Chui, M., Bisson, P., Woetzel, J., Dobbs, R., Bughin, J., and Aharon, D. (2015). The Internet of Things: Mapping the value beyond the hype. *McKinsey Global Institute*, page 144.

MQTT Version 3.1.1 (2014). Edited by Andrew Banks and Rahul Gupta. OASIS Standard. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. Latest version: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html. Acesso em 06/04/2016.

Scalagent (2015). Benchmark of MQTT servers. Technical Report January.

Shelby, Z., Hartke, K., and Bormann, C. (2014). The constrained application protocol (coap). RFC 7252, RFC Editor. http://www.rfc-editor.org/rfc/rfc7252.txt. Acesso em 06/04/2016.

Singh, D., Tripathi, G., and Jara, A. J. (2014). A survey of Internet-of-Things: Future vision, architecture, challenges and services. *2014 IEEE World Forum on Internet of Things, WF-IoT 2014*, pages 287–292.

Skerrett, I. (2015). Case Study MQTT: Why Open Source and Open Standards Drive Adoption. https://ianskerrett.wordpress.com/2015/03/04/case-study-mqtt-why-open-source-and-open-standards-drives-adoption/. Acessado em 16/11/2015.

Thangavel, D., Ma, X., Valera, A., Tan, H.-X., and Tan, C. K.-Y. (2014). Performance evaluation of MQTT and CoAP via a common middleware. In *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pages 1–6. IEEE.

WAMP Draft 2 (2015). The Web Application Messaging Protocol. Technical report. https://tools.ietf.org/html/draft-oberstet-hybi-tavendo-wamp-02. Acesso em 06/04/2016.

Zhang, L. (2011). Building Facebook Messenger. https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920. Acesso em 16/11/2015.