Paradigma Orientado a Notificações Aplicado à Programação de Microcontroladores

Lucas V. S. de Mamann
CPGEI - UTFPR
Curitiba - PR, Brasil
lucasmamann@alunos.utfpr.edu.br

Jean Marcelo Simão
CPGEI - UTFPR
Curitiba - PR, Brasil
jeansimao@utfpr.edu.br

Myriam Regattieri Delgado
CPGEI - UTFPR
Curitiba - PR, Brasil
myriamdelg@utfpr.edu.br

Daniel F. Pigatto
PPGCA - UTFPR
Curitiba - PR, Brasil
pigatto@utfpr.edu.br

***Resumo*—Este artigo visa à análise de desempenho de aplicações desenvolvidas em materializações do Paradigma Orientado a Notificações (PON), no contexto de microcontroladores. A aplicação envolve sensores e atuadores para Internet das Coisas comunicando-se em rede. Os resultados apresentados mostram que materializações do PON possuem diversas vantagens quanto ao usual Paradigma Imperativo (PI), sendo mais eficiente do que PI neste cenário. Este estudo expande os horizontes da aplicação do PON para um novo conjunto de plataformas, particularmente para aquelas com grande limitação de memória e processamento.**

I. INTRODUÇÃO

Os equipamentos com processamento embarcado e conectividade à Internet das Coisas (ou IoT do inglês *Internet of Things*) possuem vasta aplicabilidade. Eles podem ser utilizados tanto em ambientes residenciais para prover conforto, segurança ou assistência médica [1], quanto em ambientes industriais e corporativos para possibilitar o monitoramento e controle do ambiente e de processos [2]. A acessibilidade à IoT porém, muitas vezes é alcançada utilizando-se *hardwares* com microprocessadores enxutos, devido ao seu baixo custo monetário e devido ao pouco poder de processamento requerido pelos equipamentos que estes controlam [1].

O Paradigma Orientado a Notificações (PON) surge como uma alternativa aos demais paradigmas de programação, como o Paradigma Imperativo (PI), mais usado na indústria. O PON busca melhorias como o aumento de desacoplamento e redução de redundâncias entre entes computacionais [3], melhorando a *performance* de processamento e viabilizando aplicações distribuídas [4]. Dado que aplicações IoT necessitam de um código mais eficiente, é primordial buscar integrar o PON a microcontroladores voltados a essas aplicações.

Este artigo tem por base dois trabalhos anteriores. O primeiro, de Banaszewski [3], utiliza PON para controlar o estado de aparelhos de ar condicionado, com base em sensores de temperatura. O segundo trabalho, de Oliveira [4], desenvolve uma aplicação IoT distribuída em rede para o auxílio médico domiciliar, porém sem a limitação de recursos imposta no presente trabalho. Neste artigo, busca-se desenvolver um sistema genérico de sensores e atuadores distribuídos em rede, controlados via PON em um *hardware* microprocessado enxuto, comparando-o a uma solução PI.

Agradece-se à UTFPR (CPGEI e PPGCA) e aos pesquisadores do grupo do PON. Particularmente, J. M. Simão agradece à Fundação Araucária - bolsa PQ - Edital 15/2017.

II. PARADIGMA ORIENTADO A NOTIFICAÇÕES

O PON surgiu de uma solução de controle discreto para processos industriais, proposta por Simão [5], a qual evoluiu, por meio de diversos trabalhos, culminando neste novo paradigma de programação [6]. Este novo paradigma oferece alternativas para lidar com problemas do Paradigma Imperativo (PI) e do Paradigma Declarativo (PD) no tocante à computação lógico-causal, como redundâncias estruturais (repetição indevida de código) e temporais (repetição indevida de processamento), bem como o acoplamento entre partes do código [6].

O principal diferencial do PON, quando comparado ao PD ou PI, é a utilização de um mecanismo de notificação por meio de entidades minimalistas, colaborativas e adaptadas ao uso deste mecanismo. As notificações são geradas e recebidas pelas entidades apenas nos momentos apropriados, em que de fato há mudanças de estado que demandem avaliação lógico-causal, fazendo com que o sistema desenvolvido em PON seja mais conciso quanto ao uso de recursos computacionais [6].

A estrutura do PON contém duas partes: (a) uma lógico-causal, em que regras são avaliadas, sendo ou se mantendo aprovadas ou desaprovadas; e (b) outra facto-execucional, em que os elementos executam ações (processamentos usuais) que alteram o estado do sistema, mantendo registro destes estados [6]. A parte lógico-causal, via entidades *Rules* (Regras), e a parte facto-execucional, via *Fact Base Elements* (*FBE* - Elementos da Base de Fatos), são representadas na Figura 1.

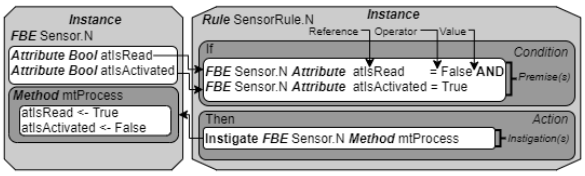

Figura 1: Exemplo de *Rule* do PON (Neves, 2021 [6]).

A parte lógico-causal do sistema é implementada por *Rules*, sendo estas entidades que definem as condicionantes analisáveis a partir de notificações advindas dos *FBEs*. Conforme a Figura 2, as *Rules* possuem *Conditions* a serem satisfeitas que, por sua vez, são relacionadas a *Premises* a serem avaliadas. Caso as *Premises* de uma *Rule* sejam todas aprovadas, a *Rule* executa sua respectiva *Action* e esta suas *Instigations*, que instigam execuções nos *FBEs*. Por outro lado, os *FBEs*

compõem a parte facto-execucional, realizando ações quando instigados. Mais precisamente, os *FBEs* possuem *Attributes*, cujos estados são relacionados e avaliados pelas *Premises*, bem como possuem *Methods*, que são instigados pelas *Rules*, completando o quadro de entidades do PON [3][6].

Destaca-se a maneira inovadora de execução colaborativa das entidade do PON por meio de notificações, constituindo uma cadeia de notificações (Figura 2). Cada *Attribute* de uma instância de um *FBE* que mudar de estado, notifica apenas as *Premises* pertinentes, que refazem seus cálculos lógicos. Cada *Premise* que mudar de estado lógico notifica apenas as *Conditions* pertinentes, as quais refazem seus cálculos lógicos pelos estados notificados contabilizados. Se a *Condition* é aprovada, ela pode aprovar sua respectiva *Rule*. Esta, quando aprovada, ativa sua *Action*, que notifica suas *Instigations*, as quais instigam os *Methods*. Estes últimos geralmente alteram os estados dos *Attributes*, reativando o fluxo de notificações. As conexões entre as entidades para fins de notificação ocorrem em tempo de construção dos sistemas [7] [8].

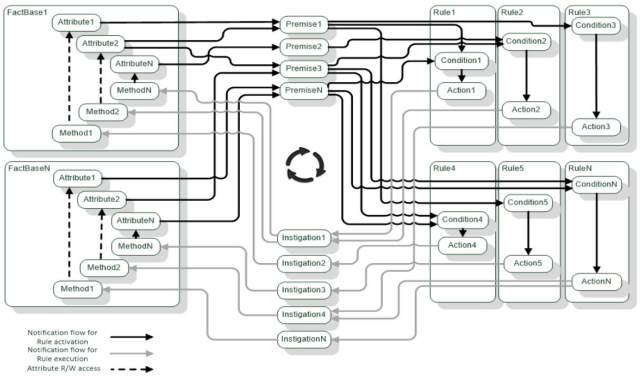

Figura 2: Representação das entidades da cadeia de notificação do PON (Ronszcka, 2019 [8]).

O aparato para programação em PON está disponível em diversas formas. Para este trabalho serão utilizadas a Tecnologia LingPON 2.0, que implementa um sistema de compilação e linguagem próprios do PON, bem como a versão 4.0 do *Framework* PON, que permite a criação de código em C++, porém utilizando os conceitos do PON. A Tecnologia LingPON 2.0, é considerada o estado da arte em PON, enquanto o *Framework* PON C++ 4.0 estabelece o estado da técnica dele.

III. PROPOSTA: MICROPON EM IOT

O sistema em estudo no presente trabalho é descrito pelo diagrama apresentado na Figura 3, no qual há *N* sensores, *M* atuadores e uma central controladora, todos conectados a uma rede local que possui acesso à Internet. Ao detectar alguma mudança de estado, podem enviar sinais à central controladora na forma de requisições a uma interface API, por meio de uma conexão qualquer (RF, WiFi ou Ethernet). Por sua vez, os elementos atuadores são passivos, no sentido que têm a necessidade de receber um comando para atuar, ainda assim possuem capacidade de alterar o ambiente de algum modo. Tais comandos são recebidos através de sinais

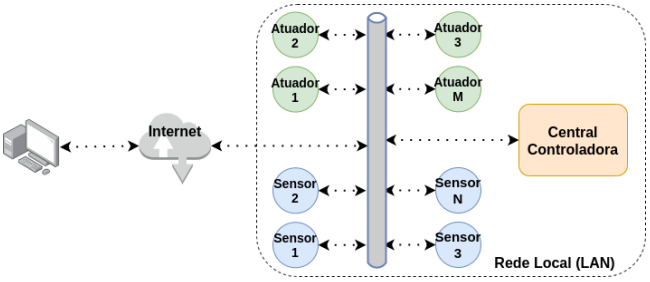

Figura 3: Representação do sistema em estudo

predeterminados, em seu endereço na rede e em uma porta dada. Assim como os sensores, os atuadores podem ser de variados tipos e possuir diferentes tipos de conexão à rede.

A central controladora gera a interface API que permite consulta e modificação dos estados atuais dos elementos, além de poder enviar comandos aos atuadores. Também, a central implementa a lógica de interação entre os diversos elementos do sistema e possui memória sobre os estados atuais.

No caso do sistema implementado em PON (MicroPon IoT) os elementos sensores podem ser entendidos como *FBEs* com *Attributes*, que notificam suas mudanças de estado não por chamadas de funções e sim por meio de troca de mensagens em uma rede, assim como estabelecido por Simão [5] para o caso de uma aplicação industrial. Os elementos atuadores, por outro lado, assumem o papel dos *Methods* de *FBEs* em PON, sendo também instigados por mensagens em rede. Os demais elementos constituintes do PON, porém, continuam sendo implementados em um código único, na central controladora, sempre à luz de seus próprios princípios.

Os **sensores** têm uma lógica simples, necessitando apenas notificar uma mudança de estado à central controladora. Dada a simplicidade da natureza dos sensores, optou-se por não implementá-los utilizando PON, pois isto não acrescentaria informações relevantes às análises realizadas para este artigo. Os sensores, portanto, foram simulados somente em computador, utilizando-se C++. Os sensores aqui implementados, simulam a extração de informações binárias do ambiente, possuindo apenas dois estados: ativo e inativo. Estes estados são alterados manualmente pelo usuário, por meio de comandos disponibilizados pelo programa simulador dos sensores. Inicialmente, todos os sensores estão no estado inativo.

Quando há uma mudança no estado de um dos sensores, o mesmo envia uma requisição à API da central controladora, para que o valor seja atualizado na central. A requisição é feita utilizando-se a biblioteca CURL, que envia um pacote TCP a um endereço, porta e URL dados no código da central de controle. Esta comunicação se dá por meio de uma conexão WiFi, entre o computador e o roteador, e ethernet, entre o roteador e a central.

Os **atuadores** são executados como *threads* concorrentes. Cada um cria um *socket* em uma porta arbitrária do computador, através do qual a central de controle pode escrever um bit de comando, causando o respectivo atuador a mudar para o estado indicado pelo bit. A implementação usou a

biblioteca padrão de *sockets* da linguagem C, de modo que esta comunicação também ocorre por meio de pacotes TCP enviados a um endereço e porta específicos. Os atuadores possuem as mesmas formas de conexão que os sensores.

A **central de controle** foi implementada de diferentes formas, além de ter sido testada tanto em uma plataforma microprocessada quanto simulada em computador. Todas as implementações, porém, podem ser divididas em duas partes: (i) uma desenvolvida em C++ que provê as funcionalidades básicas de acesso ao *hardware* da rede, implementação do protocolo de rede e da API; (ii) outra que realiza a função de controle em si, desenvolvida tanto em PON (via *Framework* PON C++ 4.0 e Tecnologia LingPON 2.0) quanto PI (via linguagem de programação C++).

Para uma comparação justa entre as implementações, tanto a estrutura do código quanto as funcionalidades foram replicadas de maneira tão similar quanto a linguagem ou plataforma permite. No entanto, em se tratando da conexão à rede, há uma diferença significativa: para a simulação em computador, foi utilizada a biblioteca padrão de *sockets* da linguagem C, tanto para o recebimento quanto para o envio de mensagens; para a implementação em microcontrolador, porém, foi feita uma adaptação da biblioteca AVR-ENC28J60 [9], pois além de implementar o protocolo para comunicação TCP/IP, o microcontrolador necessitava da implementação do acesso ao hardware de rede.

Por fim, a lógica de decisão foi implementada em C++ PI, LingPON Namespaces AVR, LingPON Namespaces e PON *Framework* 4.0, sendo a segunda utilizada apenas no microcontrolador e as duas últimas apenas simuladas em computador. Esta parte do código é a que de fato provê *Rules*, relacionando os estados dos sensores e atuadores, enquanto *FBEs*, com seus *Attributes* e *Methods*.

A aqui chamada LingPON Namespaces, diz respeito à Linguagem de Programação do PON (LingPON), associada a um dos compiladores do sistema de compilação da Tecnologia LingPON 2.0, considerado com melhor *performance* [7] [8]. Por sua vez, a LingPON Namespaces AVR diz respeito à LingPON com um compilador adaptado para o contexto de AVR, elaborado no âmbito deste presente trabalho.

A **lógica de controle** foi incorporada ao código da central controladora. Esta lógica tem a responsabilidade de associar os estados dos sensores aos estados desejáveis para os atuadores. Para simular algumas possíveis situações de aplicações reais e possibilitar o estudo de casos de interesse do sistema, foram criados quatro conjuntos de regras, cada qual variando o número de sensores envolvidos na ativação dos atuadores. Nas implementações em PON, estes conjuntos foram transcritos utilizando-se de *Rules*, enquanto que para implementações em C++, foram utilizadas estruturas usuais de se-então (*if-else statements*) equivalentes.

O primeiro conjunto de *Rules* (conjunto I) relaciona cada atuador diretamente a um sensor. Isto é, quando um sensor com ID *J* é ativado, o atuador com mesmo ID *J* também é ativado, enquanto que o atuador é desativado quando seu respectivo sensor é desativado.

Os três demais conjuntos de *Rules*, relacionam *K* sensores com os mesmos *K* atuadores, sendo *K* igual a 5 (conjunto II), 10 (conjunto III) e 20 (conjunto IV). Nestes casos, se os *K* sensores estiverem ativos ao mesmo tempo, seus respectivos *K* atuadores recebem o sinal de ativação (lógica E para ativação), enquanto que se um ou mais dos *K* sensores for desativado, todos os *K* atuadores respectivos são desativados (lógica OU para desativação).

IV. MATERIAIS E MÉTODOS

Para a simulação dos sensores e atuadores, utilizou-se o sistema operacional Fedora 32, 64-bit, com *kernel* versão 5.6.6-300, executando em um processador AMD A8-4555M Quad core, 1.6 GHz. Para reduzir as chances de preempção, utilizaram-se apenas os terminais virtuais do sistema (TTYs), de modo que a interface gráfica e serviços não essenciais não estivessem rodando em paralelo com os testes. Para a simulação da central de controle, utilizou-se um servidor com processador AMD FX-4300 Quad Core, 3.8 GHz, rodando o sistema operacional Ubuntu 20.04.2, 64-bits, com *kernel* 5.4.0-65-generic. Este servidor também estava executando apenas em modo terminal, sendo seu acesso realizado por meio de uma conexão SSH. A implementação física da central foi realizada em um microprocessador ATMEGA328P-PU, AVR 8-bit, 16 MHz. Em conjunto com este *hardware*, foi utilizado um módulo ENC28J60 para prover acesso à *ethernet*. Diferentes métricas foram utilizadas para a extração de resultados.

Tempo de requisição sem ativação de *Rules* ou equivalente (se-então) - diferença de tempo entre uma requisição, com método *GET*, à API da central controladora e o recebimento de sua resposta. Este tipo de requisição não causa a verificação de nenhuma *Rule* ou equivalente, servindo apenas como medida do tempo de processamento das funcionalidades básicas de rede e do tempo de tráfego da mensagem na rede.

Tempo de requisição com ativação de *Rules* ou equivalente - diferença de tempo entre uma requisição, com método *POST*, à API da central controladora e o recebimento de sua resposta. Este tipo de requisição pode causar ativação de *Rules* e a consequente ativação de atuadores, processamentos adicionais que também estão incluídos no tempo de resposta. Desta forma, o caso médio desta métrica, representa o tempo de processamento do mecanismo de inferência das *Rules*.

Tráfego de dados na rede - quantidade de bytes trocados entre os elementos do sistema, enviados pela rede através de mensagens. Esta métrica se torna relevante, pois um sistema distribuído em rede é tão mais rápido quanto menos dados forem transmitidos ou quanto mais rápida for a transmissão.

Os experimentos foram executados com os sensores e atuadores executando em um computador em combinação com: a central executando a lógica em C++, no microcontrolador; a central executando a lógica em PON Namespaces AVR, no microcontrolador; a central executando a lógica em C++, no servidor; a central executando a lógica em PON Namespaces, no servidor; e a central executando a lógica em PON *Framework* 4.0, no servidor. Para todos os experimentos, foram simulados 20 sensores e 20 atuadores. Este número reduzido

foi escolhido devido à limitação de memória do microcontrolador, como será melhor discutido na seção de resultados. Para manter os diferentes casos comparáveis, o mesmo número de sensores foi adotado, inclusive para sistemas com maior capacidade de memória.

1) Requisição sem ativação de Rules ou equivalente: O primeiro experimento consistiu em requisitar os estados de todos os sensores, um por vez, em sequência, e aguardar a resposta da central. O ciclo de requisições foi repetido 20 vezes, gerando um total de 400 requisições. O tempo total foi mensurado e sua média foi calculada, gerando o tempo médio de requisição sem ativação de *Rules*.

2) Requisição com ativação de Rules ou equivalente: Para o segundo experimento, incialmente a central controladora foi carregada com o código contendo o conjunto I de *Rules* (descrito na seção III). Isto dito, 5% dos sensores, ou seja, 1 sensor, foi ativado e na sequência desativado. Esta parte do experimento foi repetida por 10 vezes, gerando 20 requisições. O tempo total foi mensurado e sua média foi calculada, gerando o tempo médio de requisição com ativação de *Rules*. O mesmo procedimento foi repetido com 50% dos sensores (10 sensores) sendo ativados e na sequência desativados, gerando 200 requisições. Por fim, os mesmos passos foram realizados com 100% dos sensores (20 sensores) sendo ativados e então desativados, gerando 400 requisições.

Este experimento, composto de suas três etapas (ativação de 5, 50 e 100% dos sensores), foi então repetido para a central contendo os conjuntos de *Rules* II, III e IV, extraindo-se sempre as mesmas métricas.

3) Tráfego de dados: Como último experimento, a central foi carregada com o código contendo o conjunto de *Rules* IV, devido ao seu potencial de ser um caso crítico quanto ao tempo de processamento e volume de dados. Foram ativados 100% dos sensores (20 sensores), sendo os mesmos desativados na sequência, este processo foi repetido por 10 vezes. Foram monitorados os pacotes de dados trocados entre os sensores, atuadores e a central, obtendo-se o total de bytes movimentados na rede por estes elementos. A média de bytes por requisição foi calculada, obtendo-se o volume médio de bytes gerados por requisição.

V. RESULTADOS

Do Código: A Tabela I apresenta uma análise quantitativa do número de linhas necessárias para implementar a lógica em cada linguagem utilizada, o tamanho do código executável compilado e da memória RAM utilizada na execução deles.

Dentre as variantes, nota-se que o código em C++ é o mais simples de ser escrito, tanto em número de linhas quanto em facilidade de implementação dos códigos decisionais. A simplicidade deste, no entanto, é em relação à estrutura de código necessário e não em relação às *Rules*, dado que estas são semelhantes em todos os casos, como abordado na seção III. Esta estrutura notavelmente reduz os gastos de memória, tanto em disco quanto RAM, tornando-a atrativa principalmente para a plataforma microprocessada, que possui grande restrição de memória.

Tabela I: Análise quantitativa dos códigos

	Microcontrolador		**Computador**		
	C++	**PON Namespaces AVR**	**C++**	**PON Namespaces**	**PON Framewrok 4.0**
Número de linhas utilizadas para lógica	Entre 5 e 90	Entre 700 e 1000	Entre 5 e 90	Entre 700 e 1000	Entre 150 e 200
Tamanho do executável gerado	Entre 20 e 24 KB	48 KB	28 KB	Entre 108 e 120 KB	Entre 368 e 376 KB
Uso de memória RAM	0.577 KB	Entre 1.05 e 1.089 KB	5884 KB	Entre 5912 e 5920 KB	Entre 6188 e 6328 KB

No outro extremo, encontra-se o código que utiliza o *Framework* PON C++ 4.0, implementação estável do PON - o estado da técnica. Tal código em *Framework* PON C++ 4.0 é o mais custoso em termos de memória. Isto claramente se deve à estrutura adicional necessária para implementar o PON em si usando estruturas de dados sobre o C++. A codificação em *Framework* PON C++ 4.0 se assemelha ao C++, em facilidade e em número de linhas. Porém, há impedimentos para que o código seja portado para a plataforma microprocessada, uma vez que o *Framework* 4.0 utiliza diversas bibliotecas e funcionalidades não suportadas pelo compilador AVR. E mais, o executável gerado precisaria ser otimizado, de modo a reduzir seu tamanho em cerca de 10 vezes.

O código em PON Namespaces e sua versão adaptada para o compilar AVR, por sua vez, são um meio termo em questão de uso de memória. Isto possibilita a sua aplicação na plataforma microprocessada, apesar de ainda ser um limitante da complexidade e número de *Rules*. Entretanto, o menor número de funcionalidades desta implementação do PON reduz a facilidade e escalabilidade de sua implementação. Como se observa nos dados da Tabela I, são necessárias de 5 a 10 vezes mais linhas para se programar a mesma lógica, principalmente devido à redundância da linguagem.

A maior diferença entre o Namespaces e as demais implementações, se dá pela impossibilidade de se utilizar vetores e *loops* para a criação dos objetos e das *Rules*. O Namespaces possui a implementação de vetores e *loops*, porém estas se mostram limitadas para o desenvolvimento dos experimentos. Os vetores, por exemplo, não podem ser utilizados em *Instigations*, forçando o programador a declarar todas as variáveis ali utilizadas. Também, os índices dos vetores não suportam expressões numéricas, limitando as *Formation Rules* (FR - i.e., modelo de *Rules* que as cria) a serem mais simples. Por fim, os *loops* podem ser utilizados somente em FRs, não atendendo ao caso de atribuição de valores aos diversos índices de um vetor, por exemplo. Por estas razões, a lógica em Namespaces se tornou a menos facilitada quanto ao seu desenvolvimento.

Dos Experimentos: Ao observar os gráficos de 4 a 8, é importante notar a relação entre as colunas e linhas de barras nos gráficos. As colunas (com valores de 5 a 100%) indicam quantos sensores foram ativos durante a medição, enquanto as linhas (com valores de 1 a 20) indicam o número de sensores que compõem cada regra. Desta forma, para a linha 1, sempre

há regras sendo ativadas (para 5, 50 e 100%), pois cada regra depende apenas de um sensor. Porém, para a linha 20, por exemplo, só há regras sendo ativadas para a coluna de 100%, pois a única regra presente depende dos 20 sensores.

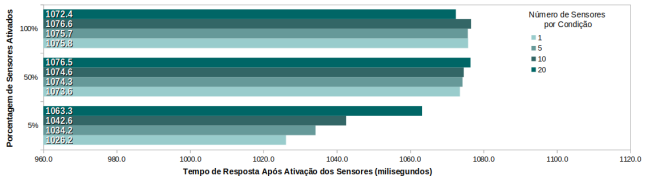

Figura 4: Tempo médio de requisição para ativação de regras com código em C++ em plataforma microprocessada

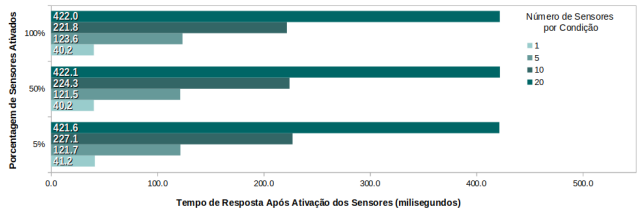

Figura 5: Tempo médio de requisição para ativação de regras com código em C++ simulado em computador

Para as implementações das regras em C++, Figuras 4 e 5, notam-se comportamentos um tanto diferentes, porém ambos tendem a crescer com a complexidade da regra e/ou número de sensores ativados. Isto indica que um maior número de requisições causa um maior volume de processamento, assim como estruturas se-então mais complexas geram o mesmo efeito principalmente para o ambiente microprocessado. Também, nota-se que o número de estruturas se-então ativadas pouco influencia no tempo de processamento, pois os valores para uma mesma linha do gráfico são muito próximos.

Estes fatos demonstram que o código em C++ repete desnecessariamente o processamento para objetos que não sofreram mudança de estado. Claramente, para uma aplicação real nesta linguagem, o código seria otimizado para evitar tais redundâncias, o que demandaria professional treinado, com gasto de tempo e esforço para tal. Assim, neste trabalho, escolheu-se manter o mesmo nível de dificuldade entre as implementações, para tornar mais justas as comparações.

Figura 6: Tempo médio de requisição para ativação de regras com código PON Nspc AVR em plataforma microprocessada

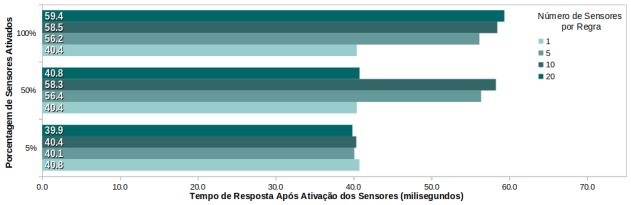

Figura 7: Tempo médio de requisição para ativação de regras com código em PON Namespaces simulado em computador

Considerando agora os resultados das implementações em PON Namespaces, apresentados nas Figuras 6 e 7, é possível notar um comportamento bastante diferenciado do caso anterior. Para a plataforma microprocessada, fica evidente que com o aumento da complexidade das *Rules*, reduz-se o tempo médio de processamento de uma requisição, o que pode parecer contraintuitivo. No entanto, isto demonstra que as notificações do PON reduzem drasticamente o processamento redundante, que ocorre no caso do C++, de forma que o processador é ocupado somente quando realmente há *Rules* para serem aprovadas.

No caso do PON Namespaces simulado, os tempos de processamento são mais próximos entre si, devido à maior capacidade de processamento da plataforma. Ainda assim, nota-se que o processador é mais exigido somente para os casos em que de fato há aprovação de *Rules*; resultado semelhante ao da implementação com PON *Framework* 4.0, apresentado na Figura 8, não sendo possível distinguir ambas as implementações.

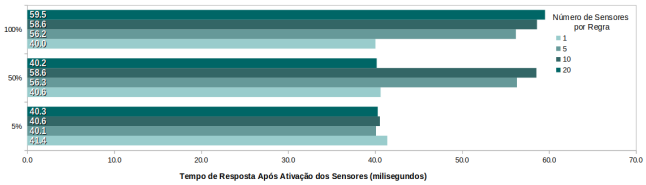

Figura 8: Tempo médio de requisição para ativação de regras com código em PON *Frw* 4.0 simulado em computador

Nota-se nos resultados uma discrepância, já esperada, entre a plataforma microprocessada e a simulação em computador. Devido ao maior processamento, o computador consegue responder mais rapidamente às requisições, até mesmo nos piores casos em C++, além de apresentar tempos semelhantes para todas as implementações, cujos melhores casos são aproximadamente 40 milissegundos. Entretanto, a implementação em PON para o microprocessador surge como uma boa alternativa, pois esta foi capaz de responder cerca de duas vezes mais rápido às requisições do que as simulações, em seus melhores casos. Isto mostra que o PON, em sua implementação via LingPON Namespaces, tem um grande potencial, até mesmo para *hardwares* com pouco poder de processamento, chegando a se igualar e a superar o código em programação imperativa. Claramente, ainda são necessárias melhorias quanto ao Namespaces, conforme já citado, para que possa ser considerado

estado da técnica em PON. No entanto, tais melhorias são um caminho natural no grupo de pesquisa.

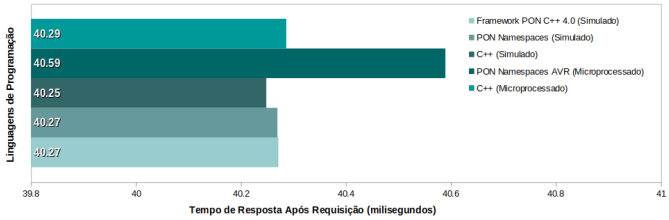

Figura 9: Tempo médio de requisições sem ativação de regras

A Figura 9 apresenta os resultados relativos ao segundo experimento. Enquanto os resultados anteriores analisavam o tempo de processamento de possíveis ativações de *Rules*, causadas por mudanças de estado, o presente resultado trata apenas do tempo necessário para se retornar os estados atuais. O objetivo deste teste é verificar se há diferenças consideráveis quanto ao processamento básico das diferentes implementações. Os resultados obtidos, porém, mostram que as diferenças são mínimas e as flutuações estão dentro do esperado. Isto mostra que, para esta aplicação em específico, todas as implementações são igualmente satisfatórias no que se trata do envio de mensagens em rede.

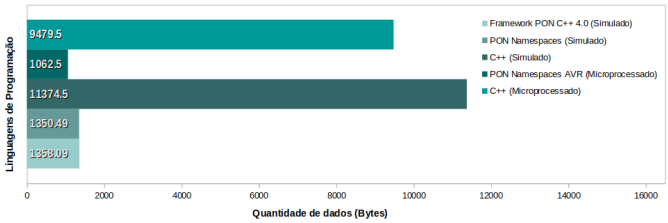

Figura 10: Resultados para tráfego de dados por requisição

Finalmente, os resultados respectivos ao tráfego de dados na rede, causados pela comunicação entre sensores, atuadores e central, estão mostrados na Figura 10. A primeira observação que deve ser feita é que este resultado está diretamente relacionado com a redundância do código em C++, já mencionada anteriormente. Esta redundância faz com que a central notifique seus atuadores, mesmo quando não há mudança de estado, congestionando desnecessariamente a rede. Isto causa um aumento de 70 a 80 vezes no tráfego de dados. Outra observação cabível, porém de menor importância, é a ligeira vantagem dos códigos carregados no microprocessador, quanto ao volume de dados gerado. Isto ocorre, pois a versão do protocolo TCP/IP ali implementado foi refeita manualmente e simplificada, devido às limitações de memória desta plataforma, gerando assim mensagens menores em número de bits.

Em aplicações que dependem fortemente da comunicação em rede, esta se tornar rapidamente congestionada em um caso real, se a mudança de estado de um único sensor causar a movimentação de um volume tão grande de dados. Os dados da Figura 10 destacam a importância de se gerar notificações apenas quando realmente há mudanças de estado.

VI. CONCLUSÕES

Este trabalho utilizou o PON em um *hardware* microprocessado. Os resultados mostraram que é possível desenvolver códigos que possuem desempenho igual ou melhor que a programação imperativa, além serem mais eficientes, dependendo da aplicação. O estudo serviu para demonstrar que o PON não está restrito a *hardwares multicore* ou com grande poder de processamento. Mostrou-se que o PON pode ser aplicado a plataformas bastante limitadas, estendendo sua aplicação também a qualquer tipo de dispositivo de IoT, alinhando seu uso com a idealização inicial, de um paradigma pensado para um sistema distribuído em rede.

A implementação do PON no cenário deste trabalho, também serviu para explicitar as vantagens da estrutura providenciada por este paradigma. Com esforço equiparável à implementação de uma lógica simples em programação imperativa, o PON reduz as redundâncias do código e pode ser integrado a sistemas distribuídos sem grande esforço, uma vez que esteja tecnologicamente estável. Contudo, neste âmbito, fica a ressalva de que o LingPON Namespaces carece de certas melhorias, bem como sua melhor adaptação para o compilador AVR. O estágio inicial destes, dificulta sua programação e integração, porém de forma alguma as impossibilita, conforme demonstrado neste trabalho.

Como trabalhos futuros, sugere-se estudar melhorias no LingPON Namespaces AVR e em seu compilador, para aplicações com microcontroladores. Isto pode propiciar um código mais otimizado, em tamanho, além da geração de códigos já integrados ao código básico necessário para um microcontrolador. Também serão buscadas aplicações mais complexas e completas, no sentido de haver mais elementos microprocessados, conectados em rede.

REFERÊNCIAS

[1] F. John Dian, R. Vahidnia, and A. Rahmati, "Wearables and the internet of things (iot), applications, opportunities, and challenges: A survey," *IEEE Access*, vol. 8, pp. 69200–69211, 2020.

[2] S. Aheleroff, X. Xu, Y. Lu, M. Aristizabal, J. Pablo Velásquez, B. Joa, and Y. Valencia, "Iot-enabled smart appliances under industry 4.0: A case study," *Advanced Engineering Informatics*, vol. 43, p. 101043, 2020.

[3] R. F. Banaszewski, "Paradigma orientado a notificações: avanços e comparações," 2009. Dissertação de Mestrado, CPGEI/UTFPR, Curitiba - PR.

[4] R. N. Oliveira, "Assistência à autonomia domiciliar empregando paradigma orientado a notificações," 2019. Dissertação de Mestrado, CPGEI/UTFPR, Curitiba - PR.

[5] J. M. Simão, "A contribution to the development of a hms simulation tool and proposition of a meta-model for holonic control," 2005. Doctoral Thesis, CPGEI/UTFPR (Brazil) - UHP (France), Curitiba - PR.

[6] F. S. Neves, J. M. Simão, and R. R. Linhares, "Application of generic programming for the development of a c++ framework for the notification oriented paradigm," *11th International Conference on Information Society and Technology*, pp. 56–61, 2021.

[7] L. K. Oshiro, A. F. Ronszcka, J. A. Fabro, and J. M. Simão, "Linguagem e compilador para o paradigma orientado a notificações: uma solução performante orientada a regras," in *Anais da XII Escola Regional de Alto Desempenho de São Paulo*, pp. 61–64, SBC, 2021.

[8] A. F. Ronszcka, "Método para a criação de linguagens de programação e compiladores para o paradigma orientado a notificações em plataformas distintas," . Tese de Doutorado - CPGEI, UTFPR, 2019.

[9] B. Prayudha, "Biblioteca avr-enc28j60," 2014. Disponível em: https://github.com/bprayudha/avr-enc28j60. Acesso em: 01 Ago. 2021.