UbiTester: um *Framework* para Geração de Planos de Teste para Computação Ubíqua

Thiago M. Martins [1], **Raquel A. F. Mini**[1], **Humberto T. Marques-Neto** [1]

[1]Programa de Pós-graduação em Informática

[2]Pontifícia Universidade Católica de Minas Gerais (PUC Minas)
Caixa Postal 30535-901 – Belo Horizonte– MG – Brazil

thiago.moraes@sga.pucminas.br, raquelmini@pucminas.br, humberto@pucminas.br

Resumo. *Este artigo apresenta o framework denominado UbiTester, o qual é capaz de gerar um plano de teste para softwares desenvolvidos para a realização de computação ubíqua. Os planos de teste gerados pelo UbiTester podem abranger todo o escopo do projeto. Além disto, a geração de cenários de testes nesse framework é feita com base em parâmetros definidos em uma interface gráfica construída para facilitar o seu uso. Com o propósito de validar o framework UbiTester, foram criados cenários de testes para serem utilizados em uma aplicação que funciona em um simulador de computação ubíqua, denominado Aura, utilizando-se tanto o plano de teste gerado pelo UbiTester quanto um plano de teste construído de acordo com o que é proposto no RUP (Rational Unified Process). Os resultados indicaram que o uso do UbiTester pode aumentar a qualidade e a produtividade do desenvolvimento de aplicações de computação ubíqua.*

Abstract. *This paper presents a framework called UbiTester, which is capable of generating a test plan for software developed for the realization of ubiquitous computing. The testing plan generated by UbiTester can cover the entire scope of the project. Moreover, the scenario generation of tests with this framework is based on parameters defined in a graphical interface constructed to facilitate its usage. In order to validate the application of UbiTester, we created test scenarios to be used in an application that runs on a simulator of ubiquitous computing, named Aura, using both the test plan generated by UbiTester as a test plan built according to what is proposed in the RUP (Rational Unified Process). The results showed that the usage of UbiTester can increase the quality and the productivity of the development of ubiquitous computing applications.*

1. Introdução

No início da década de 90, o mundo conheceu um novo paradigma computacional criado por Mark Weiser, a computação ubíqua. A utilização de recursos computacionais de forma natural para a realização das mais variadas tarefas do dia-a-dia, a qualquer momento e em qualquer lugar [Weiser 1999]. Atualmente, existem pesquisas destinadas à viabilização de aplicações desse paradigma, indicando que, num futuro não tão distante, essas estarão presentes no nosso cotidiano. Dentre essas aplicações, podem-se destacar aplicações hospitalares [Bardram 2004], casas inteligentes [Kidd 1999], aplicações de varejo [Kindberg and Fox 2002], aplicações de tempo real [Soldatos 2011] e aplicações de entretenimento [Cheok 2004].

A computação ubíqua possui muitas características específicas e que necessitam de tratamentos adequados durante o planejamento e desenvolvimento de *software*. Entre essas características é possível citar que os sistemas da computação ubíqua são geralmente sensíveis ao contexto, possuem restrições de tempo real, são dedicados a tarefas específicas e, na maioria dos casos, a entrada de dados é coletada pelos nós sensores no ambiente de sensoriamento. A entrada de dados coletada pelos nós sensores ocorre através da interação do usuário com o meio físico e que é diferente da entrada de dados de um sistema tradicional que ocorre através do teclado. O reconhecimento da presença de um usuário pelo nó sensor no meio físico pode levar a necessidade da execução de um serviço de ligar a luz e essa entrada de dados na computação ubíqua ocorre de forma transparente para o usuário.

Devido a computação ubíqua ter uma forte interação com o meio físico, os requisitos relacionados à infraestrutura estão mais presentes do que em sistemas tradicionais. Esses requisitos são chamados de requisitos não-funcionais. O requisito não funcional é um requisito que avalia a infraestrutura, o ambiente e a localização dos dispositivos na rede [Spínola 2008, Jakob 2005]. Por outro lado, o requisito funcional é responsável por garantir o funcionamento adequado do sistema, como o cadastro de um cliente ou a emissão de um relatório. Os requisitos não-funcionais não recebem tanta importância nos sistemas tradicionais. Por isso um plano de teste tradicional contempla normalmente, na sua maior parte apenas os requisitos funcionais. Na computação ubíqua, um requisito não funcional pode impactar diretamente um requisito funcional do sistema, ocasionando uma falha na execução de suas atividades. Dessa forma, um plano de teste tradicional não se adequaria completamente na computação ubíqua por não ter como foco os requisitos não-funcionais, dificultando assim o planejamento de um projeto ubíquo.

Um problema enfrentado na computação ubíqua é a falta de artefatos específicos para o planejamento dos testes, pois, os testes nos sistemas ubíquos são realizados através de modelos de processos de testes genéricos. Um exemplo seria a liberação de um serviço para um usuário não autorizado, podendo causar prejuízos para a instituição, como o roubo de um equipamento ou informações dos usuários da rede. A falha em um dispositivo ubíquo pode causar insegurança na execução de suas atividades e nos serviços prestados, pois, o alto número de requisições coletadas pelos nós sensores pode dificultar o controle e gerenciamento dos usuários que podem ter acesso às informações de contexto e como a informação será armazenada. Entretanto, os problemas podem ser solucionados através de modelos de processos de testes que abordam as boas práticas para garantir as funcionalidades dos sistemas.

Entre os modelos de processos de testes utilizados para se obter um funcionamento eficaz dos sistemas ubíquos, destacam-se o RUP (*Rational Unified Process*). O RUP é um modelo de processo que possui, entre outros sub-processos, um de teste [Rup 2012]. O RUP é muito utilizado atualmente por desenvolvedores, analistas, engenheiros de sistemas e é justificado pelo fato de ser um modelo de propósito geral, aplicável à grande maioria dos sistemas e plataformas de implementação. Apesar de o RUP ajudar no planejamento dos testes, o RUP possui um modelo de sub-processo de teste muito genérico, dificultando a identificação de pontos chaves da computação ubíqua e que são necessárias para garantir que os sistemas ubíquos funcionem corretamente. Por exemplo, a computação ubíqua possui um maior número de requisitos não-funcionais que os siste-

mas tradicionais e a identificação desses requisitos no RUP pode se tornar inviável. Segundo [Shuja and Krebs 2007], o RUP incentiva a adaptação de seus processos em qualquer projeto, além de suas atividades e artefatos que seguem um modelo de boas práticas para que atenda as necessidades do projeto. Como o RUP pode ser aplicado em qualquer projeto independente da área, o tempo e custo para o planejamento e elaboração dos artefatos se torna maior que em outros modelos que possuem características específicas para o tipo de projeto. Para [Kruchten 2003], o modelo de sub-processo de teste do RUP não é tão adequado para sistemas específicos por ser um modelo burocrático e por sempre precisar de adaptações, pois os projetos específicos dependem de requisitos que não podem ser generalizados.

O objetivo deste trabalho é desenvolver um *framework*, denominado UbiTester, capaz de gerar um plano de teste para *softwares* desenvolvidos para realização de computação ubíqua. Os planos de teste gerados pelo UbiTester podem abranger todo o escopo do projeto. Além disto, a geração de cenários de testes nesse *framework* é feita com base em parâmetros definidos em uma interface gráfica construída para facilitar o seu uso. Com o propósito de validar o *framework* UbiTester, foram criados cenários de testes para serem utilizados em uma aplicação que funciona em um simulador de computação ubíqua, denominado Aura, utilizando-se tanto o plano de teste gerado pelo UbiTester quanto um plano de teste construído de acordo com o que é proposto no RUP (*Rational Unified Process*).

Os resultados deste artigo indicam que o uso do UbiTester pode aumentar a qualidade e a produtividade do desenvolvimento de aplicações de computação ubíqua. A criação de um plano de teste padrão para a computação ubíqua auxiliará na execução dos testes, buscando uma qualidade maior na troca de requisições na rede pelos nós sensores. Além da adequação do plano de teste, será possível obter um plano de testes claro e eficaz, possibilitando identificar um maior número de erros nos sistemas ubíquos que os modelos utilizados em sistemas tradicionais.

Este artigo está dividido da seguinte forma: Na seção 2, são apresentados os trabalhos relacionados que estão divididos em testes de sistemas em tempo real, testes de sistemas cientes do contexto, testes de sistemas em ambientes inteligentes e testes em sistemas distribuídos. Na seção 3, é apresentado o desenvolvimento do *framework* UbiTester. Na seção 4, é apresentado o cenário para computação ubíqua que foi utilizado para validação dos planos de testes. Na seção 5, é apresentado o resultado do desenvolvimento do *framework* UbiTester e a validação dos planos de testes RUP e UbiTester. Por fim, na seção 6, são apresentadas as conclusões.

2. Trabalhos Relacionados

2.1. Testes de Sistemas em Tempo Real

Os sistemas de tempo real são sistemas que monitoram, respondem ou controlam um determinado ambiente, como por exemplo, um sensor de temperatura que informa ao microcontrolador que determinados níveis de temperatura foram alcançados e através desse microcontrolador é possível ajustar o sistema de refrigeração.

O uso de monitoramento, análise e controle dos eventos do mundo real é apresentado no trabalho de [Pressman 2004]. Para tanto, existem elementos que coletam dados,

obtém e formatam informações de um ambiente externo. O trabalho mostra que o tempo de resposta de um sistema iterativo deve ser restrito para que não gere resultados insatisfatórios e os serviços devem estar preparados para interagir com outros dispositivos e serviços de rede. No entanto, o trabalho não descreve o tempo ideal para a comunicação entre os sistemas, evitando assim uma perda considerável de requisições.

Foi apresentado no trabalho de [Guan and Offutt 2006] que o tempo é um dificultador para os testes de sistemas em tempo real, além do paralelismo das tarefas e a relação do sistema com o seu ambiente de *hardware*. O trabalho descreve formas para garantir uma comunicação adequada entre os dispositivos, sendo necessário aplicar testes de *softwares* com o intuito de identificar problemas de tempo real e que vão ocorrer somente na execução de um determinado evento. No entanto, o trabalho foca apenas no teste de tarefas e não apresenta as demais técnicas de testes da engenharia de *software*.

2.2. Testes de Sistemas Cientes do Contexto

Um sistema ciente do contexto é um sistema capaz de prover informações importantes e relevantes dentro de um ambiente. Esse tipo de sistema é capaz de modificar o seu comportamento e ações baseado no contexto do ambiente ou até mesmo exibir ao usuário informações do contexto por questões de comodidade.

Um estudo de levantamento de requisitos foi realizado no trabalho de [Baldauf and Dustdar 2007]. O trabalho mostra que a identificação de requisitos é essencial para a implementação dos sistemas cientes de contexto, como por exemplo, acessar um sensor localmente ou remotamente, avaliando a quantidade de usuários que irá acessar a rede ou até mesmo os recursos disponíveis pelo dispositivo. No trabalho foram identificados problemas nos protocolos de comunicação, no desempenho da rede e na qualidade dos serviços. A solução foi aplicar técnicas de testes de caixa branca para garantir a estabilidade do sistema e obter o retorno esperado pela aplicação. No entanto, o trabalho não descreve todas as técnicas de testes que foram aplicas ao projeto.

2.3. Testes de Sistemas em Ambientes Inteligentes

Os sistemas para ambientes inteligentes são utilizados para melhorar as atividades no cotidiano das pessoas de forma transparente e imperceptível. Uma infraestrutura para proteção dos dados foi realizado em [Buttyan 2010]. Nesse trabalho, a infraestrutura visa proteger o ambiente de criminosos, problemas na rede, e destaca pontos que tornam as redes de sensores sem fio mais confiáveis no monitoramento da segurança do ambiente. Além disso, é proposto no trabalho a segurança dos dispositivos e a segurança na comunicação dos dados, garantindo que os dados sejam transmitidos de forma segura. No entanto, o trabalho foca na segurança da infraestrutura, porém não apresenta técnicas de testes que possam garantir a segurança na comunicação dos dispositivos.

A implementação de um alarme em projetores com sensores de movimento foi realizado no trabalho de [Mitsubish 2003]. O trabalho mostra que o alarme é acionado através de uma chave de segurança. Para garantir o funcionamento correto do sistema, foram realizados testes de presença com pessoas entrando e saindo do ambiente no qual estava sendo monitorado. No trabalho foi avaliado se o dispositivo conseguia identificar uma ou mais pessoas no perímetro de cobertura. No entanto, o trabalho mostra como foi feito o estudo de caso e não apresenta se foram aplicados outros tipos de testes.

2.4. Testes de Sistemas Distribuídos

Os sistemas distribuídos são sistemas que interligam uma coleção de computadores através de uma rede de computadores e que através de *softwares* permitem o compartilhamento dos dados.

Um trabalho de sistemas distribuídos com alta disponibilidade através de serviços críticos foi apresentado em [Souza and Maldonado 1999]. Nesse trabalho, é indispensável o uso de mecanismos de tolerância a falhas. Estes mecanismos devem ser validados para não se comportar de forma inesperada em situações em que o sistema não pode parar e que atualmente as técnicas de testes são utilizadas para validar funcionalidades e para medir a qualidade dos sistemas. No entanto, o trabalho foca no custo excessivo de aplicar testes e não detalha quais técnicas de testes obteria um custo mais acessível.

A computação ubíqua possui todas as características que os sistemas citados, porém a sua comunicação é realizada de forma inteligente. Conforme o usuário se movimenta pelo ambiente, os sensores trocam informações sobre o seu deslocamento e executa serviços pré-configurados para atender o usuário.

3. Desenvolvimento do *Framework* UbiTester

O UbiTester foi desenvolvido com foco nas características específicas da computação ubíqua, como a integração e comunicação dos dispositivos de forma inteligente, a ausência de entrada de dados pelo teclado e o alto número de requisitos não-funcionais. Através do UbiTester, é possível gerar os cenários de testes que deverão ser testados e, para isso, é necessário que o usuário efetue parametrizações no *framework*, permitindo assim o cadastro de casos de testes, além de ser possível customizar todo o plano de teste de forma rápida e eficaz. Uma das primeiras parametrizações realizadas no *framework* é a escolha do número de nós sensores e nós coordenadores que a rede de sensoriamento possuirá. Os nós sensores são responsáveis por enviar informações coletadas do ambiente para um ou mais nós coordenadores, enquanto que o nó coordenador possui o objetivo de gerenciar todas as requisições dos nós sensores e assim garantir que as atividades sejam realizadas com sucesso. O UbiTester também permite efetuar a seleção das técnicas de testes que serão aplicadas no projeto e que são pontos chaves para garantir a elaboração de um plano de teste eficaz.

Para cada técnica de teste selecionada pelo usuário, será aberta uma aba no *framework* UbiTester contendo as características específicas para o tipo de teste selecionado, como por exemplo, o teste de carga possui o nível de severidade do teste que pode variar entre baixa, normal e crítica. A severidade no nível baixa descreve três cenários de testes, o nível normal seis cenários e o nível crítico doze cenários. Como o teste de carga tem o objetivo de enviar requisições periódicas, foi utilizado tempos aleatórios em uma escala de 0 até 10 segundos. Dessa forma, quanto maior o nível da severidade maior será o número de cenários de testes gerados, pois uma aplicação que possui um monitoramento contínuo deve estar sempre em atividade e normalmente não pode parar em situações de problemas, caso contrário, os danos podem ser irreparáveis. A periodicidade do teste também foi implementada e pode variar entre diário, semanal e mensal. Se o usuário selecionar a opção semanal, os testes serão gerados informando que deverá ser realizado durante toda a semana. Também é possível selecionar o turno que será aplicado os testes como manhã, tarde e noite, além de poder selecionar a quantidade de usuários que estarão

dentro do ambiente de sensoriamento. A quantidade de usuários é importante para fazer simulações em uma rede de computação ubíqua, avaliando se o nó coordenador consegue atender o número de requisições enviadas pelos nós sensores.

A técnica de teste de desempenho e a técnica de teste de estresse possui o mesmo cenário que a técnica de teste de carga, pois esses três tipos de testes necessitam de uma aplicação para automatizar os testes ou rotinas criadas manualmente para gerar um volume grande de requisições. As demais técnicas de testes permitem que o usuário selecione se os cenários de testes pré-estabelecidos na aplicação serão utilizados no projeto. Uma vez avaliada e selecionada todas as técnicas de testes necessárias, será necessário preencher cada atributo da aplicação com os detalhes do escopo do projeto conforme a Figura 1.

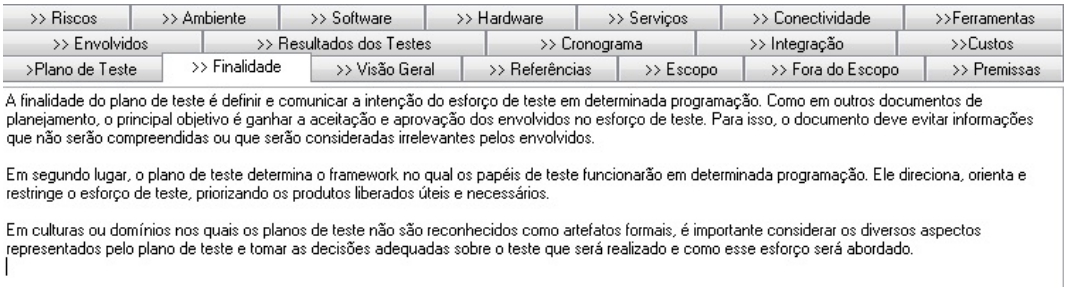

Figura 1. Descrição Detalhada do Atributo Finalidade

4. Cenário para Computação Ubíqua

Foi montado um cenário para computação ubíqua com o intuito de executar o plano de teste gerado pelo *framework* UbiTester e o plano de teste gerado através do modelo de processo de teste do RUP. O objetivo do cenário é simular um ambiente real para computação ubíqua, buscando assim uma validação precisa dos planos de testes através do cenário proposto e o cenário da rede é apresentado na Figura 2.

Para validar os planos de testes, foi instalado em cada computador um simulador para computação ubíqua, denominado Aura [Pedro and Garlan 2002]. Em seguida, os computadores foram posicionados em cada ambiente a ser sensoriado para simular a entrada e saída de um usuário do ambiente. Os nós sensores foram representados com a letra (E) e o nó coordenador com a letra (C). A letra (E) representa o nó sensor cliente que foi posicionado nos ambientes: sala, quarto 01 e quarto 02. A letra (C) representa o nó coordenador que foi posicionado no ambiente da copa. A área total de cobertura do ambiente em metros é de 15x40. As informações trocadas pelos nós sensores ocorrem através da tecnologia Wi-Fi (*wireless fidelity*), que é um dos meios de comunicação utilizado pelo simulador Aura.

5. Resultados

5.1. UbiTester

A interface do *framework* UbiTester é apresentado na Figura 3. Através de uma descrição detalhada presente em cada atributo, é possível se orientar durante a elaboração do

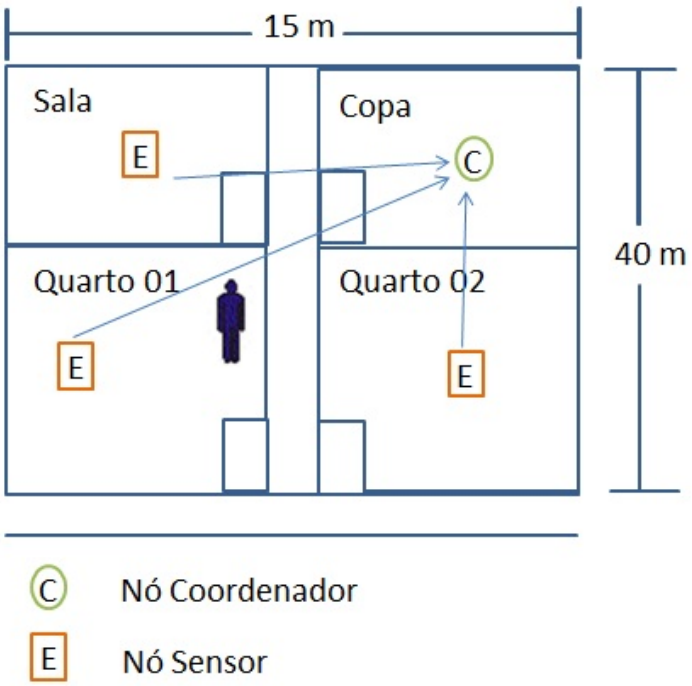

Figura 2. Cenário de Simulação.

plano de teste, e cada técnica de teste selecionada pode ser parametrizado gerando novos cenários de testes. O UbiTester gera os cenários de testes baseado nas definições do usuário, como número de nós sensores, nós coordenadores ou através da técnica de teste selecionada. Através de alguns parâmetros, como definição da criticidade do teste no ambiente de sensoriamento, o UbiTester selecionará os testes mais relevantes à serem testados, dependendo do número de entrada feitas pelo usuário. Os cenários de testes implementados no UbiTester foram avaliados por profissionais da área de testes, buscando gerar cenários pertinentes para a realidade da computação ubíqua.

Foi feito um experimento para criar um plano de teste através dos processos de testes do RUP com 3 grupos de alunos. Cada grupo é composto por 4 alunos de graduação em engenharia de *software* e estão no sexto período do curso de Sistemas de Informação. Os grupos não foram os mesmos para não influenciar nos resultados durante a criação do plano de teste pelo UbiTester. O primeiro experimento foi gerar um plano de teste através do UbiTester por meio do estudo de caso realizado pelos alunos. Para elaborar o plano de teste, foi necessário fazer a seleção e parametrização de cada atributo no UbiTester. Foi avaliado que a curva de aprendizado dos alunos no uso do *framework* UbiTester foi muito rápida, pois, o tempo de aprendizado no uso do *framework* UbiTester e a entrega do plano de teste foi em um tempo menor que o esperado. A elaboração do plano de teste para computação ubíqua terminou em um tempo médio de 60 minutos e é demonstrado um plano de teste parcial gerado pelo UbiTester na Figura 4.

Para a elaboração do plano de teste do RUP, os três primeiros grupos de alunos deveriam escolher e selecionar os atributos do *template* do RUP que atendesse o cenário proposto. Durante o experimento, apenas um grupo conseguiu finalizar o plano

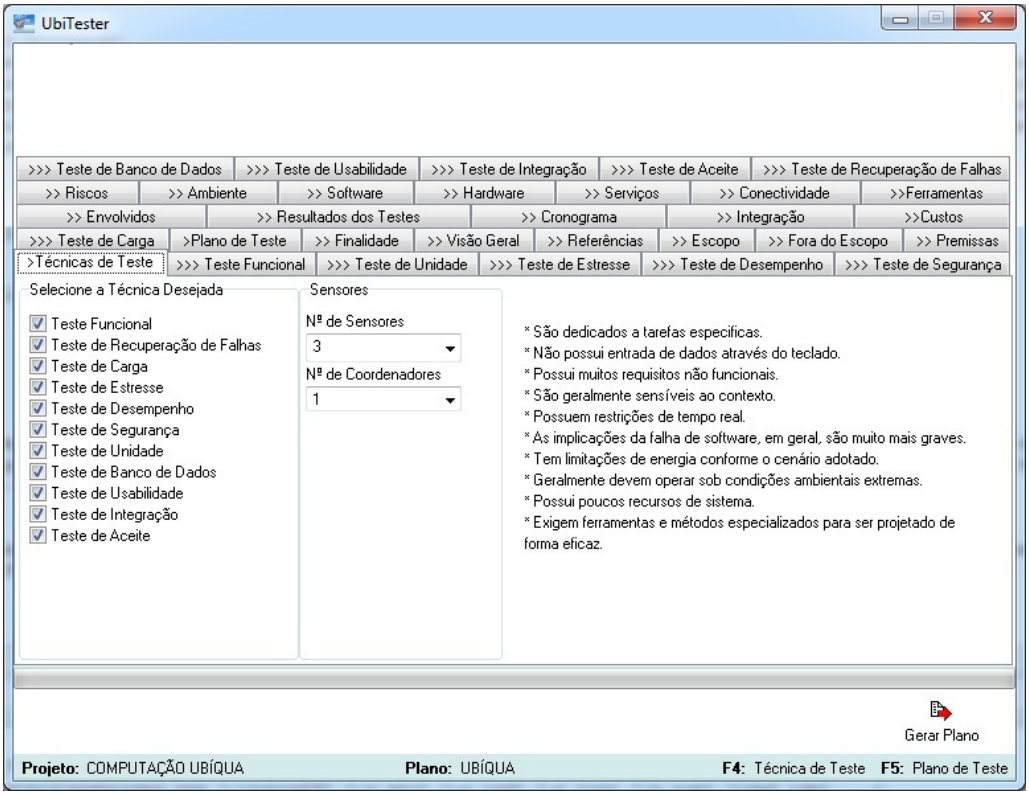

Figura 3. UbiTester

de teste próximo ao tempo proposto e os demais dois grupos não conseguiram finalizar a elaboração do plano de teste em 180 minutos, alcançando aproximadamente 75% do objetivo proposto. Entre as dificuldades encontradas na elaboração dos planos de testes através do RUP, foi destacada a diversidade de requisitos que o RUP disponibiliza e a seleção e identificação de pontos chaves para a computação ubíqua não foram identificadas de forma fácil. O resultado do experimento é apresentado na Figura 5.

Foi identificado que o grupo que conseguiu concluir o plano de teste não utilizou um *template* para o plano de teste do RUP, ou seja, colocaram os textos fora do padrão e estética não se adequando a um artefato de plano de teste utilizado no mercado. Também foi visto que a escrita do plano de teste em várias situações não teve a sua gramática correta, pois a descrição dos testes não foi realizada através de verbos como verificar, elaborar e realizar que é proposto pela literatura durante a elaboração de um artefato de testes. Sendo assim, o plano de teste não obteve um grau de qualidade adequado e dependendo da situação foi avaliado que ficou confuso o entendimento do cenário proposto.

Outro problema encontrado nos planos de testes dos alunos foi à ausência de técnicas de testes que abordam os requisitos não-funcionais, como por exemplo, o teste de recuperação e Falhas não foi identificado no artefato de testes. Como existem diversas técnicas de testes na engenharia de *software*, escolher determinadas técnicas baseado em um contexto necessita de um conhecimento maior e como o plano de teste padrão do RUP é muito amplo e genérico [Kannan et al. 2004], dependendo do cenário proposto

Plano de Testes para Computação Ubíqua

Projeto: UBÍQUA **Plano:** COMPUTAÇÃO UBÍQUA
Sensor(es): 3 **Coordenador(es):** 1

Teste de Desempenho

Verificar o tempo de resposta do(s) 3 nó(s) sensor(es) e o do(s) 1 nó(s) coordenador(es) com 1 usuário(s).
Verificar o tempo de resposta do(s) 3 nó(s) sensor(es) e o do(s) 1 nó(s) coordenador(es) com 10 usuário(s).
Verificar o tempo de resposta do(s) 3 nó(s) sensor(es) e o do(s) 1 nó(s) coordenador(es) com 26 usuário(s).
Verificar o tempo de resposta do(s) 3 nó(s) sensor(es) e o do(s) 1 nó(s) coordenador(es) com 33 usuário(s).
Verificar o tempo de resposta do(s) 3 nó(s) sensor(es) e o do(s) 1 nó(s) coordenador(es) com 46 usuário(s).
Verificar o tempo de resposta do(s) 3 nó(s) sensor(es) e o do(s) 1 nó(s) coordenador(es) com 50 usuário(s).
Verificar o tempo de resposta para processar os dados pendentes após uma falha do(s) 3 nó(s) sensor(es) no período da Noite com 1 usuário(s).
Verificar o tempo de resposta para processar os dados pendentes após uma falha do(s) 3 nó(s) sensor(es) no período da Noite com 10 usuário(s).
Verificar o tempo de resposta para processar os dados pendentes após uma falha do(s) 3 nó(s) sensor(es) no período da Noite com 26 usuário(s).

Figura 4. Plano de Teste Parcial UbiTester.

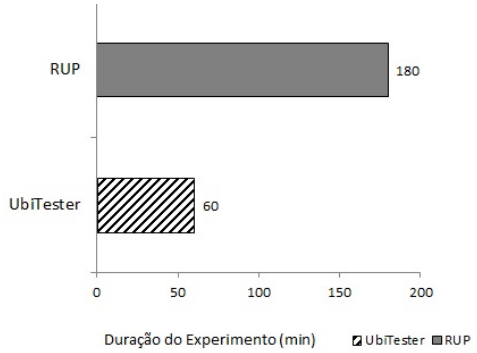

Figura 5. Tempo Médio (em minutos) para a Criação dos Planos de Testes com RUP e UbiTester

pode apresentar dificuldades na escolha das técnicas de testes.

5.2. Validação dos Planos de Testes

A quantidade de cenários de testes é importante para a identificação de problemas em uma aplicação. Dessa forma, foi possível mapear de forma quantitativa os cenários de testes propostos nos planos de testes criados através do RUP e UbiTester. Assim, quanto maior o número de cenários de testes identificados, maiores são as chances de encontrar problemas em uma aplicação. A Tabela 2, apresenta a quantidade de cenários de testes identificados no plano de teste RUP e do UbiTester pelos alunos durante o experimento. No plano de teste do RUP foram identificados 32 cenários de testes e no plano de teste gerado pelo UbiTester foram identificados 102 cenários. O UbiTester possui mais de 150 cenários de testes, porém foram gerados apenas 102 cenários através das definições dos alunos no *framework*. Como a maioria dos cenários de testes foram gerados pelo UbiTester, não houve problemas de cenários repetidos e as demais informações sobre a cobertura do teste foram cadastradas pelos alunos. Nesse caso, os cenários de testes gerados pelo UbiTester são definidos através do número de nós sensores, nós coordenadores e técnicas de testes selecionadas pelo usuário. Em contrapartida, os cenários gerados através do

RUP são baseados no conhecimento do testador. Através dos cenários identificados, pode ser visto um ganho maior no uso do UbiTester e o mesmo possui maiores chances de encontrar problemas em uma aplicação que o plano de teste gerado pelo RUP. Dessa forma, pode ser destacado que o UbiTester possui diversos cenários pré-definidos para ambientes genéricos na computação ubíqua, além de permitir que os cenários de testes específicos sejam cadastrados pelo testador.

Tabela 1. Cenários de Testes.

Técnicas de Testes	**RUP**	**UbiTester**
Teste de Recuperação	0	5
Teste de Estresse	5	16
Teste de Desempenho	8	24
Teste de Carga	9	36
Teste Funcional	3	6
Teste de Segurança	2	3
Teste de Banco de Dados	2	5
Teste de Integração	2	4
Teste de Unidade	0	3
Teste de Usabilidade	0	2
Teste de Aceite	1	2
Total	**32**	**102**

Quanto maior o número de cenários de testes, maiores são as chances de encontrar problemas, pois para cada cenário de teste, poderá ser criados casos de teste, buscando identificar problemas na rede de sensoriamento. Após a criação dos artefatos de testes, é necessário validar se os cenários de testes propostos são eficientes e se podem ser aplicados na computação ubíqua. Para isso, foi necessário validar esses artefatos e em seguida avaliar qual plano de teste possui um melhor custo e benefício. Para validar os artefatos, foram selecionadas as seguintes técnicas de testes: técnica de teste de carga, técnica de teste de estresse, técnica de teste de desempenho e técnica de teste de recuperação.

A primeira simulação utilizou a técnica de teste de carga com o objetivo de verificar o comportamento do nó sensor, ou seja, o envio de requisições em horários e tempos diferentes para medir o funcionamento adequado do dispositivo. Assim, foi necessário enviar as requisições ao nó sensor simulando a entrada de vários usuários no ambiente de sensoriamento. O nó sensor coordenador era responsável por identificar o usuário e gerenciar as requisições recebidas pelos outros nós sensores. Esta simulação ocorreu durante um período de cinco horas com o objetivo de avaliar se um sensor seria capaz de atender vários usuários ao mesmo tempo com um desempenho satisfatório. Foi realizada uma simulação utilizando o envio de requisições nos intervalos de 1, 3 e 6 segundos conforme descrito no plano de teste do UbiTester e os intervalos de 2, 3 e 7 segundos conforme descrito no plano de teste do RUP. Os intervalos gerados pelo UbiTester são baseadas nas definições realizadas pelo usuário na ferramenta, ou seja, quando o número de nós sensores for menor do que dez nós sensores, o testes ocorrem nos tempos de 1, 3 e 6 segundos. Caso o número de nós sensores seja maior do que dez sensores, os testes ocorrem nos tempos de 2, 6 e 11 segundos. Entre as diversas simulações realizadas, foi identificado que esses intervalos conseguem identificar um maior número de requisições perdidas. Em contrapartida, os intervalos do RUP são baseados na experiência do testador sobre a computação ubíqua.

Os intervalos são diferentes, pois, o UbiTester gera os intervalos através de cenários que poderiam alcançar melhores resultados na execução dos testes e no caso do RUP, os intervalos dependem da experiência do testador. A simulação utilizou três nós sensores enviando pacotes a cada 1 segundo para o nó coordenador. Foi identificado que o nó coordenador não consegue atender vários usuários simultaneamente devido ao grande volume de informação. A segunda simulação foi realizada enviando pacotes a cada 3 segundos, e o resultado foram bem melhor que o da primeira simulação ocorrendo uma perda menor de pacotes. A terceira simulação foi realizada enviando pacotes a cada 6 segundos, e o simulador atendeu bem as requisições, não gerando perdas de pacotes na rede. No entanto, o cenário de teste gerado pelo UbiTester conseguiu registrar um maior número de requisições perdidas que a abordagem do RUP, pois, o intervalo de tempo entre o envio das requisições é menor que o RUP, sobrecarregando assim o nó sensor. Os resultados obtidos na simulação são apresentados na Figura 6 e pode ser identificado que o UbiTester consegue identificar uma disponibilidade maior dos nós sensores na rede e um maior número de requisições perdidas.

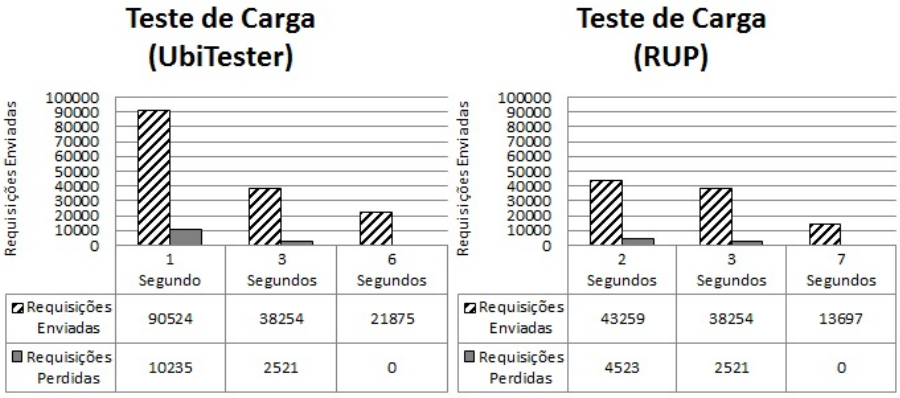

Figura 6. Requisições Enviadas por 5 Horas.

A segunda simulação utilizou a técnica de teste de estresse com o objetivo de verificar qual é o limite máximo de requisições atendidas pelo nó sensor. Nesse teste foi avaliado se os nós sensores conseguem atender uma demanda de requisições durante um período longo de tempo e se o nó coordenador conseguiria atender as requisições dos outros nós sensores simultaneamente. Foi enviada uma requisição a cada 1 segundo, conforme descrito no plano de teste do UbiTester, e enviado uma requisição a cada 2 segundos, conforme descrito no plano de teste do RUP. Para realizar o teste, foi necessário efetuar um *loop* infinito de requisições via *request* nas classes responsáveis pela comunicação com os nós sensores. No entanto, o cenário de teste gerado pelo UbiTester mostra que um nó sensor consegue atender um número maior de requisições que o cenário de teste do RUP. Além disso, foi possível visualizar um maior número de requisições perdidas através do UbiTester. Como o envio de requisições para ambos os planos de testes foi de 12:40 Horas, não foi possível apresentar o limite máximo que o nó sensor consegue atender as requisições através do plano de teste do RUP, pois a periodicidade do envio de suas requisições foi a cada 2 segundos, enquanto que o UbiTester foi a cada 1 segundo. Assim, o plano de teste do RUP não conseguiu estressar o nó sensor durante a simulação. Os resultados obtidos na simulação são apresentados na Figura 7.

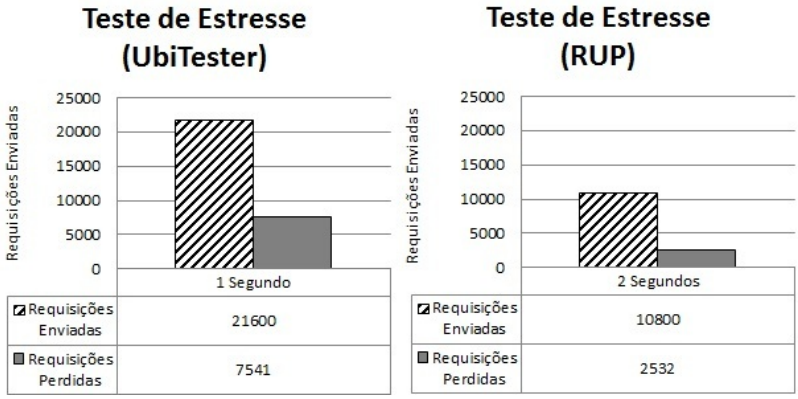

Figura 7. Requisições Enviadas por 12:40 Horas.

A terceira simulação utilizou a técnica de teste de desempenho com o objetivo de verificar o tempo de resposta entre as requisições e o número de usuários suportados pelo ambiente de sensoriamento. Através do teste de desempenho foi avaliado o tempo de resposta dos três nós sensores no período de 6 horas. Além do tempo de resposta dos nós sensores foi necessário verificar se o nó coordenador conseguiria gerenciar as requisições de forma rápida. Nesta simulação foi avaliado a quantidade de requisições que foram enviadas e respondidas em menos de 1 segundo e maiores que 1 segundo pela descrição do plano de teste do UbiTester. Em contrapartida, foram avaliadas as requisições que foram enviadas e respondidas em menos de 5 segundos e maiores que 5 segundos, conforme descrito no plano de teste do RUP. Durante o teste, foi enviada uma carga de requisições antecipadas para um nó sensor em um horário não agendado para o recebimento da informação e também uma carga maior do que a normal antecipada. Para aumentar essa carga de dados, foi necessário criar *loops* de *request* via *browser* para aumentar o número de requisições na rede. Foi avaliado que o desempenho do nó coordenador, nas duas primeiras horas, foi de aproximadamente 0,496 segundos e, nas demais horas, a resposta do nó coordenador variou entre o intervalo de 1 a 4 segundos. No entanto, o cenário de teste gerado pelo UbiTester conseguiu registrar um maior número de requisições perdidas que o cenário de teste do RUP. Como o intervalo de envio das requisições do RUP é bem maior que o intervalo do UbiTester, foi possível identificar um maior número de requisições enviadas pelo RUP, porém não foi possível identificar o intervalo de tempo que a perda de requisições foi maior. Para o cenário de requisições maior que 5 segundos, não houve envio de requisições, pois os nós sensores conseguiram atender todas as requisições em menos de 5 segundos. O resultado da simulação é apresentado na Figura 8.

A quarta simulação apresentada é a técnica de teste de recuperação que teve como objetivo avaliar se o nó coordenador conseguiria recuperar as informações armazenadas após uma falha no dispositivo. Como o simulador Aura guarda todas as informações em um arquivo XML *(eXtensible Markup Language)*, seria necessário apenas parar e iniciar o serviço responsável por guardar as informações no arquivo. Para realizar o teste, foram enviadas requisições através dos três nós sensores durante 6 horas, gerando um volume maior de requisições que o esperado pelo nó coordenador. Após identificar um volume de requisições guardadas no arquivo XML, foi necessário parar o serviço atribuído à

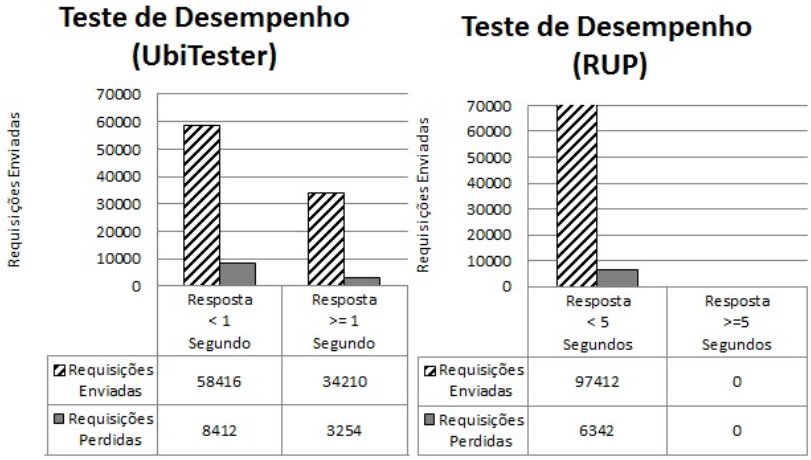

Figura 8. Requisições Enviadas por 6 Horas.

classe Observador de Contexto manualmente, que é responsável pela troca de informações através do Wi-Fi. Este serviço é responsável pela entrada e saída de informações entre os nós sensores. Como os nós sensores continuaram enviando requisições para o nó coordenador que estava inativo foi gerado um volume de requisições que não foram processadas durante 10 minutos. Os nós sensores não fazem a verificação se o nó coordenador está inativo e, por isso, permaneceram enviando as requisições. Após voltar o serviço Wi-Fi manualmente, o nó coordenador não conseguiu reprocessar as informações guardadas no arquivo XML, pois a sessão já tinha sido finalizada. No entanto, o cenário de teste gerado pelo UbiTester conseguiu registrar um alto número de requisições perdidas. Em contrapartida, a técnica de teste de recuperação não foi contemplada no plano de teste do RUP pelos alunos e por isso não foi avaliada. Os resultados obtidos na simulação são apresentados na Figura 9.

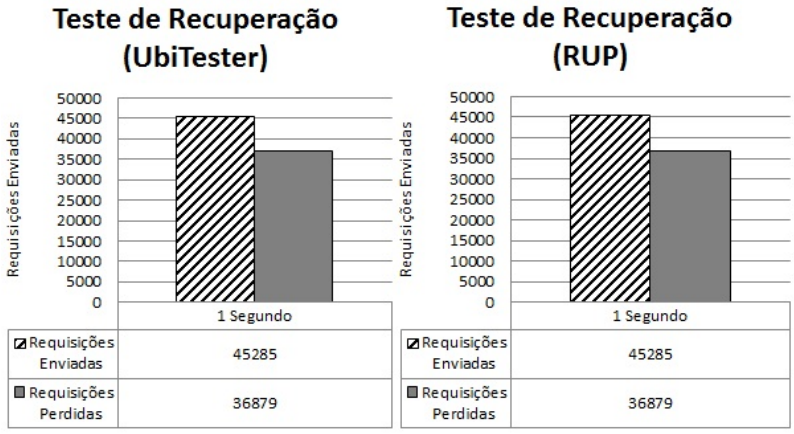

Figura 9. Requisições Enviadas por 6 Horas.

6. Conclusão

O modelo de plano de teste do RUP possui como vantagem a rastreabilidade dos testes comparando com os requisitos do sistema, além de permitir o acompanhamento da cobertura de testes e dos testes já realizados. Os testes podem ser escritos e executados por pessoas diferentes e são iniciados na concepção do projeto o que garante que sejam encontrados defeitos nas fases iniciais do projeto, onde a correção é menos custosa. A grande desvantagem do RUP é que suas orientações são muito abrangentes, visando principalmente modelos de *softwares* tradicionais, dificultando assim a seleção de pontos chaves para sistemas de computação ubíqua.

O *framework* UbiTester busca identificar orientações distintas para sistemas de computação ubíqua, consequentemente a criação e manutenção do plano de teste são realizadas em um menor tempo. A criação de um *framework* possibilitou a representação de alguns artefatos importantes no planejamento de teste de sistemas de computação ubíqua como: plano de teste, caso de teste, *template* padrão de teste, lista de ideias de teste e registro de teste. Dessa forma é possível garantir que as técnicas de testes de requisitos não-funcionais serão avaliadas durante o planejamento dos testes do projeto, além de propor cenários de teste sem dados de entrada, considerando apenas as ações e resultados esperados. Com esse plano de teste espera-se encontrar um maior número de defeitos do que um teste sem documentação ou um plano de testes que não possui o foco sistemas ubíquos.

Percebe-se, notoriamente, por meio da análise das métricas de elaboração e execução dos planos de testes, que o *framework* UbiTester permite um foco maior nos testes para computação ubíqua conforme abordados neste artigo.

7. Agradecimentos

Este trabalho é parcialmente financiado pelo CNPq e FAPEMIG.

Referências

Baldauf, M. and Dustdar, S. (2007). A survey on context-aware systems. In *International Journal of Ad Hoc and Ubiquitous Computing*, volume 2, pages 263–277. ACM, Geneva, Switzerland, EUA.

Bardram, J. (2004). Applications of context-aware computing in hospital work: Examples and design principles. In *In: SAC 04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1574–1579. ACM, New York, NY, USA.

Buttyan, L. (2010). Application of wireless sensor networks in critical infrastructure protection: Challenges and design options [security and privacy in emerging wireless networks]. In *Wireless Communications, IEEE*, volume 17, pages 44–49. IEEE, Washington, DC, USA.

Cheok, A. (2004). Human pacman: A mobile, wide-area entertainment system based on physical, social, and ubiquitous computing. In *Personal and Ubiquitous Computing*, volume 8, pages 71–81. ACM, Springer-Verlag London, UK.

Guan, J. and Offutt, J. (2006). An industrial case study of structural testing applied to safety-critical embedded software. In *ISESE '06 Proceedings of the 2006 ACM/IEEE*

international symposium on Empirical software engineering, pages 272–277. ACM, New York, NY, USA.

Jakob, E. (2005). The java context awareness framework (jcaf) - a service infrastructure and programming framework for context-aware applications. In *PERVASIVE'05 Proceedings of the Third international conference on Pervasive Computing*, pages 98–115. ACM, Berlin, Heidelberg, EUA.

Kannan, R., Sarangi, S., and Sitharama, S. (2004). Sensorcentric energy-constrained reliable query routing for wireless sensor networks. In *Journal of Parallel and Distributed Computing*, volume 64, pages 839–852. ACM, Orlando, FL, USA.

Kidd, C. (1999). The aware home: A living laboratory for ubiquitous computing research. In *Lecture notes in computer science*, pages 191–198. ACM, Springer-Verlag London, UK.

Kindberg, T. and Fox, A. (2002). System software for ubiquitous computing. In *IEEE Pervasive Computing*, volume 1, pages 70–81. ACM, Piscataway, NJ, USA.

Kruchten, P. (2003). The rational unified process: an introduction. pages 54–75. Ciência Moderna, Rio de Janeiro, RJ, BRA, 2 edition.

Mitsubish (2003). Mitsubish xl5900 colorview projector.

Pedro, J. and Garlan, G. (2002). Aura: An architectural framework for user mobility in ubiquitous computing environments. In *WICSA 3 Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance*, volume 224, pages 29–43. ACM, Montreal, Quebec, Canada.

Pressman, R. (2004). Software engineering: A practitioner's approach. pages 73–84. McGraw-Hill, Rio de Janeiro, RJ, BRA, 6 edition.

Rup (2012). Iteration master test plan.

Shuja, A. and Krebs, J. (2007). Ibm rational unified process reference and certification guide. solution design. In *Prentice Hall*, volume 1, pages 105–201. New York, NY, USA.

Soldatos, J.; Pandis, I. K. L. J. (2011). Agent based middleware infrastructure for autonomous context-aware ubiquitous computing services, computer communications. In *Computer Communications*, volume 30, pages 577–591. ACM, Amsterdam, The Netherlands.

Souza, S. and Maldonado, J. (1999). Mutation testing applied to estelle specifications. In *Software Quality Control*, volume 8, pages 285–301. ACM, Hingham, MA , EUA.

Spínola, R., P. F. T. G. (2008). Supporting requirements definition and quality assurance in ubiquitous software project. In *Leveraging Applications of Formal Methods, Verification and Validation Third International Symposium*, volume 17, pages 587–603. Rio de Janeiro, RJ.

Weiser, M. (1999). The computer for the 21st century. In *ACM SIGMOBILE Mobile Computing and Communications Review*, volume 3, pages 3–11. ACM, New York, NY , EUA.