

**ESCOLA SUPERIOR ABERTA DO BRASIL – ESAB
CURSO DE PÓS-GRADUAÇÃO LATO SENSU EM
ENGENHARIA DE SISTEMAS**

MESAILDE SOUZA DE OLIVEIRA MATIAS

**TESTES DE INTEGRAÇÃO CONTÍNUA EM UM DISPOSITIVO
EMBARCADO BASEADO EM LINUX**

**VILA VELHA – ES
2014**

MESAILDE SOUZA DE OLIVEIRA MATIAS

**TESTES DE INTEGRAÇÃO CONTÍNUA EM UM DISPOSITIVO
EMBARCADO BASEADO EM LINUX**

Monografia apresentada ao Curso de Pós-Graduação em Engenharia de Sistemas da Escola Superior Aberta do Brasil, como requisito para obtenção do título de Especialista em Engenharia de Sistemas, sob orientação do Professor Ramon Rosa Maia Vieira Junior.

**VILA VELHA – ES
2014**

MESAILDE SOUZA DE OLIVEIRA MATIAS

**TESTES DE INTEGRAÇÃO CONTÍNUA EM UM DISPOSITIVO
EMBARCADO BASEADO EM LINUX**

Monografia aprovada em de de

Banca Examinadora

**VILA VELHA – ES
2014**

RESUMO

Muitos sistemas embarcados disponíveis no mercado e presentes no cotidiano da população são hoje baseados no sistema operacional Linux. O software embarcado nesses sistemas precisa ser cuidadosamente testado, pois a expectativa do usuário é que o dispositivo funcione sem causar a interrupção de suas atividades. Entretanto, existem poucos estudos de caso detalhados na literatura sobre integração contínua nesses sistemas. Relatos são ainda mais raros quando o software embarcado é desenvolvido por um projeto colaborativo. Este trabalho visa reduzir essa lacuna, descrevendo uma solução para execução de testes automatizados de integração contínua em um dispositivo com processador ARM e sistema operacional Linux, customizada para um projeto de software livre já existente. Mesmo nesse estudo de caso tão específico, foi possível encontrar testes que só falharam quando executados no próprio dispositivo embarcado, ou seja, existiam problemas no código que só foram descobertos com o processo de testes proposto.

Palavras-chave: Integração contínua. Testes automatizados. Sistemas embarcados. Depuração.

LISTA DE TABELAS

Tabela 1 - Requisitos de alinhamento de instruções de carga/armazenamento. Fonte: Adaptado de ARM (2014, p. 108).....	37
Tabela 2 - Número de testes bem sucedidos e mal sucedidos quando executados diretamente no Kindle, antes e depois das contribuições enviadas ao projeto.....	40
Tabela 3 - Configurações do computador utilizado nos testes de desempenho.....	41
Tabela 4 - Resumo dos tempos de execução dos testes em um Kindle e em um computador.....	42

LISTA DE FIGURAS

Figura 1 – Dispositivo embarcado Kindle PaperWhite, da empresa Amazon. Na foto, o aparelho aparece executando o software KOReader para a leitura de um livro em formato PDF.....	12
Figura 2 – Trecho de um livro eletrônico preprocessado pelo software de refluxo de texto k2pdfopt (WILLUS, 2011). O valor “250 μ F” (destacado em vermelho) fazia parte de uma figura de diagrama eletrônico, mas é incorretamente reconhecido como parte do texto e inserido no meio de uma frase. Além disso, o tamanho da fonte do texto muda bruscamente após esse ponto.....	14
Figura 3 – Benchmark do LuaJIT versus a implementação padrão do Lua em arquitetura ARM. Fonte: Pall (2014).....	16
Figura 4 – Exemplo de uma pull request na plataforma GitHub. A página principal é um espaço para discussões. Clicando na aba Files changed, pode-se visualizar as alterações incluídas na pull request.....	17
Figura 5 – Tela de configuração de webhooks do projeto no GitHub, onde deve ser inserido o endereço do servidor HTTP que monitora as alterações no repositório.....	20
Figura 6 – Listagem dos registros de compilação e testes automatizados disponibilizados pelo servidor de integração contínua para cada alteração (commit) efetuada no repositório.	21
Figura 7 – Exemplo de requisição HTTP enviada pelo GitHub notificando a ocorrência de um evento do tipo push.....	22
Figura 8 – Início de um registro de compilação e testes automatizados sendo acessado por meio do navegador.....	23
Figura 9 – Configuração da interface de rede USBNetwork na ponta do computador usando o NetworkManager.....	27
Figura 10 – Saída do depurador gdb ao analisar o erro de instrução ilegal que ocorria na revisão 217f56590b5eb5fc08a286d636a8a237ab68e930 da biblioteca lua-serialize quando executada em arquitetura ARM. Esse erro não ocorria em ambientes x86.....	32
Figura 11 – Leitura do conteúdo de uma posição de memória com o depurador gdb.....	33
Figura 12 – Trecho do código da biblioteca lua-serialize em linguagem de máquina analisado com o auxílio da ferramenta IDA Pro.....	34
Figura 13 – O gdb é capaz de listar as bibliotecas dinâmicas em uso por um executável, juntamente com o endereço onde estão carregadas.....	35

Figura 14 – Arquivo de código fonte serialize.c em torno da linha 441.....	36
Figura 15 – Estrutura de serialização de objetos onde um campo é representado por um byte indicando seu tipo seguido de seu valor. Caso o ponteiro para o valor do campo seja desreferenciado diretamente, é grande o risco de acesso não alinhado à memória.....	38
Figura 16 – Trecho da saída do Valgrind ao monitorar-se a execução dos testes automatizados no dispositivo Kindle, mostrando a ocorrência de um acesso inválido à memória que não ocorria em ambiente simulado.....	39
Figura 17 – Arquivo de código fonte serialize.c em torno da linha 471.....	40
Figura 18 – Histograma de tempos de execução para os testes de backend no Kindle. O tempo médio foi de 92,96 segundos, cerca de 4,5 vezes mais lento que no computador. Entretanto, os valores de tempo ficaram menos espalhados que nos testes em computador, gerando um desvio padrão menor e indicando um maior determinismo do Kindle na execução destes testes.....	42
Figura 19 – Histograma de tempos de execução para os testes de frontend no Kindle, com cache frio. O tempo médio foi de 20,46 segundos, cerca de 6,8 mais lento que no computador.....	43
Figura 20 – Histograma de tempos de execução para os testes de frontend no Kindle, com cache quente. O tempo médio foi de 5,77 segundos, cerca de 5,5 mais lento que no computador. A redução de desempenho no Kindle (com relação ao computador) é, portanto, menor em testes que envolvem aproveitamento do cache, provavelmente devido à eficiência razoável das leituras realizadas a partir da memória Flash.....	43
Figura 21 – Histograma de tempos de execução para os testes de backend no computador hospedeiro. O tempo médio foi de 20,65 segundos. Dentre os 50 tempos, 4 foram muito menores que os outros, valendo em torno de 18,5 segundos. Esse comportamento provavelmente ocorre devido aos testes de rede que fazem parte de suíte de backend..	44
Figura 22 – Histograma de tempos de execução para os testes de frontend no computador hospedeiro, com cache frio. O tempo médio foi de 3,00 segundos. Não são observadas características especialmente peculiares na distribuição de tempos.....	45
Figura 23 – Histograma de tempos de execução para os testes de frontend no computador hospedeiro, com cache quente. O tempo médio foi de 1,04 segundos. Não são observadas características especialmente peculiares na distribuição de tempos.....	45

SUMÁRIO

1 INTRODUÇÃO.....	7
2 SISTEMA ESTUDADO.....	11
2.1 Dispositivo Kindle.....	11
2.2 Projeto KOREader.....	15
Recursos e implementação do software.....	15
Organização e práticas da equipe.....	17
3 DESENVOLVIMENTO DAS TÉCNICAS.....	19
3.1 Monitorando alterações no repositório.....	19
3.2 Compilando automaticamente o KOREader.....	23
3.3 Mapeando o KOREader via rede para o dispositivo.....	25
3.4 Adaptando o Valgrind e o Busted para o Kindle.....	28
3.5 Executando remotamente o Busted.....	29
4 APLICAÇÃO DAS TÉCNICAS.....	30
4.1 Suítes de teste disponíveis.....	30
4.2 Identificação de testes limitados pelo hardware.....	30
4.3 Depuração de erros de instrução ilegal.....	31
4.4 Identificação de acessos inválidos à memória.....	39
4.5 Solução dos problemas.....	40
5 AVALIAÇÃO DE DESEMPENHO.....	41
6 CONCLUSÕES.....	47
REFERÊNCIAS.....	48
ANEXOS.....	51
Anexo 1 – Código fonte.....	51
ciserver.lua.....	51
ssl-genkeys.sh.....	54
config.env.....	55
common.sh.....	55
dispatch.sh.....	55
update-koreader.sh.....	55
make-koreader.sh.....	56
run-tests.sh.....	57
quirks.sh.....	58
make-infra.sh.....	59

Anexo 2 – Tempos de execução.....	62
Tempos no Kindle.....	62
Tempos no computador.....	63

1 INTRODUÇÃO

Muitos equipamentos eletrônicos de uso cotidiano são controlados internamente por sistemas computacionais cuidadosamente projetados para uma aplicação específica – os denominados sistemas embarcados. Tratam-se de celulares, televisores, roteadores, máquinas de lavar roupa, automóveis, dentre outros produtos hoje praticamente indispensáveis para a vida moderna.

Os sistemas embarcados precisam ser testados de forma extensiva para que não tragam problemas ao usuário (BROEKMAN e NOTENBOOM, 2003). No caso de aviões, veículos, aparelhos de suporte à saúde, ou ainda aplicações industriais de nicho, tal necessidade é óbvia, devido aos riscos à segurança das pessoas que podem ser acarretados por uma falha nesses sistemas. Entretanto, a estabilidade é importante mesmo em sistemas não-críticos, pois não há nada mais inconveniente para o usuário final que um aparelho celular travando no meio de uma ligação, ou um televisor que não muda de canal até ser retirado da tomada.

Uma prática de desenvolvimento de software que pode contribuir fortemente com a estabilidade de um sistema é a integração contínua. De acordo com Fowler e Foemell (2006), essa prática consiste em fazer com que os desenvolvedores de uma equipe integrem frequentemente seus trabalhos a uma árvore de código principal, sempre de forma que cada integração seja verificada por um processo de compilação e testes automatizados.

No entanto, é necessário superar uma série de desafios para realizar testes automatizados em um sistema embarcado (GRENNING, 2007). Por exemplo, a plataforma de hardware pode não estar completamente pronta para uso, ou seja, estar sendo desenvolvida simultaneamente ao software embarcado. Mesmo que o projeto de hardware esteja finalizado, este pode ser de difícil produção ou obtenção, apresentando-se como um recurso escasso. Outro problema é que as plataformas embarcadas costumam dispor de uma quantidade limitada de memória e processamento, bastante inferior à dos computadores utilizados pelos desenvolvedores, inviabilizando ou tornando muito lenta a execução de alguns tipos de teste.

Por esses e outros motivos, é comum que os desenvolvedores optem por executar a maioria dos testes em um ambiente simulado em seus próprios computadores, fora das plataformas embarcadas reais. Caso o software embarcado seja desenvolvido na forma de módulos ou objetos desacoplados, é possível utilizar a técnica de *mocking* para simular a interação com o hardware (KARLESKY et al., 2007).

Contudo, Grenning (2007) argumenta que devido a diferenças entre arquiteturas de processadores, compiladores e bibliotecas de sistema, não é possível confiar que o comportamento dos testes seja o mesmo no computador do desenvolvedor e na plataforma embarcada. Por esse motivo, defende que os testes sejam executados no próprio sistema embarcado, ainda que com uma frequência menor que os testes efetuados no computador.

Este trabalho visa aplicar essa ideia no processo de integração contínua de um projeto de software livre já estabelecido. O projeto escolhido é o leitor de livros eletrônicos KOReader (HOU et al., 2014), voltado para sistemas embarcados baseados em Linux com processador ARM. Atualmente, o KOReader é compatível com as plataformas Kindle, Kobo e Android, porém seus testes automatizados são executados apenas em ambiente simulado, na arquitetura x86. O objetivo geral deste trabalho é desenvolver uma solução capaz de monitorar a linha principal do repositório de código fonte do KOReader e, quando ocorrerem mudanças, automaticamente compilar o projeto e executar os testes em um dispositivo Kindle.

O trabalho é justificado tanto pelo seu viés técnico como pelo científico. Do lado técnico, aplicamos o conhecimento preexistente para trazer melhorias a um projeto livre e disponível para a população em geral. Por se tratar de um projeto colaborativo, é esperado que este seja particularmente beneficiado pela existência de um processo sólido de integração contínua, devido às características heterogêneas de seus desenvolvedores (HOLCK e JØRGENSEN, 2007; LANUBILE, 2009). Além disso, este trabalho contribui para divulgar técnicas aplicáveis a outros sistemas embarcados baseados em Linux, sistema operacional muito utilizado neste segmento do mercado (HENKEL, 2006).

No viés científico, coletamos e interpretamos novos dados empíricos, por meio de uma pesquisa experimental, que demonstram que alguns testes automatizados cujo

resultado é positivo em ambiente simulado falham quando efetuados na plataforma embarcada. Portanto, tais dados ajudam a sustentar a hipótese de que é importante executar os testes nos próprios dispositivos embarcados, dando um embasamento ainda maior às boas práticas defendidas pela literatura.

Os dados experimentais foram coletados a partir de um conjunto de testes preexistentes, que já eram utilizados em ambiente simulado no processo de integração contínua do KORreader. Além de determinar o número de testes mal sucedidos quando executados na plataforma embarcada e investigar o motivo pelo qual estes falharam, comparamos o tempo de execução dos testes no ambiente real com o tempo no ambiente simulado, demonstrando a viabilidade prática do processo de integração contínua proposto.

O texto está organizado da seguinte forma: o Capítulo 2 descreve em maiores detalhes o sistema embarcado e o projeto de software livre que são objetos de estudo deste trabalho.

O Capítulo 3 discute o desenvolvimento das técnicas destinadas a cumprir cada um dos seguintes objetivos específicos:

- Monitorar continuamente as alterações do repositório de código fonte do projeto.
- Compilar o software quando a linha principal do repositório sofrer alterações.
- Mapear no dispositivo alvo, via rede, um diretório do computador hospedeiro contendo o software compilado.
- Adaptar o *framework* de testes utilizado pelo projeto para ser executado diretamente na plataforma embarcada.
- Executar remotamente o *framework* de testes, coletando os resultados.

O Capítulo 4 apresenta o resultado dos testes, a metodologia de depuração dos testes que falharam, e as contribuições enviadas ao projeto KORreader.

O Capítulo 5 detalha a metodologia de coleta de tempos de execução, agrega os

dados em gráficos e realiza uma avaliação de desempenho dos testes, comparando o dispositivo com o ambiente simulado.

Por fim, o trabalho é concluído no Capítulo 6, com um resumo das melhorias obtidas, limitações, e direcionamento para trabalhos futuros.

2 SISTEMA ESTUDADO

2.1 Dispositivo Kindle

O Kindle é um dispositivo embarcado destinado à leitura de livros eletrônicos comercializado pela empresa Amazon. Trata-se de um dos mais conhecidos aparelhos com tela de tinta eletrônica (COMISKEY et al., 1998; CHEN et al., 2003), tecnologia que possibilita uma leitura agradável, menos cansativa, e com excelente contraste mesmo sob a luz do sol.

A tinta eletrônica dispensa a existência de uma fonte emissora de luz no dispositivo, uma vez que funciona por meio de partículas de tinta controladas eletronicamente, que interagem com a luz do próprio ambiente como se fossem os pigmentos contidos em folhas impressas de papel comum. Devido a esse fato, outra grande vantagem desse tipo de tela é o baixo consumo, pois não é necessário gastar energia para manter uma imagem estática. Ou seja, enquanto o usuário lê uma página de um livro, a carga da bateria praticamente não é reduzida, sendo consumida apenas nas mudanças de páginas.

Entretanto, por não possuir luz própria, os primeiros dispositivos com tela de tinta eletrônica não permitiam leitura no escuro. Por esse motivo, chegaram a ser lançadas algumas capas protetoras que vinham acompanhadas de uma lanterna LED para iluminar a tela do Kindle, até que no final de 2012 a Amazon introduziu o Kindle PaperWhite, versão do dispositivo adotada neste trabalho (Figura 1). A partir dessa versão, o Kindle passou a vir com uma fonte de luz LED integrada, cuja intensidade pode ser configurada por meio do software embarcado no dispositivo.

O Kindle PaperWhite possui um processador i.MX50 de 800 MHz com núcleo ARM Cortex-A8 (FREESCALE, 2013), 256 MB de memória RAM e 2 GB de memória Flash. O sistema operacional é Linux, com bibliotecas de sistema baseadas na distribuição Ubuntu para ARM. O aplicativo de leitura de livros da Amazon é desenvolvido em linguagem Java e executa em uma máquina virtual da Sun/Oracle específica para sistemas embarcados, denominada *Compact Virtual Machine* (CVM). Os formatos de livros eletrônicos suportados por esse aplicativo são:

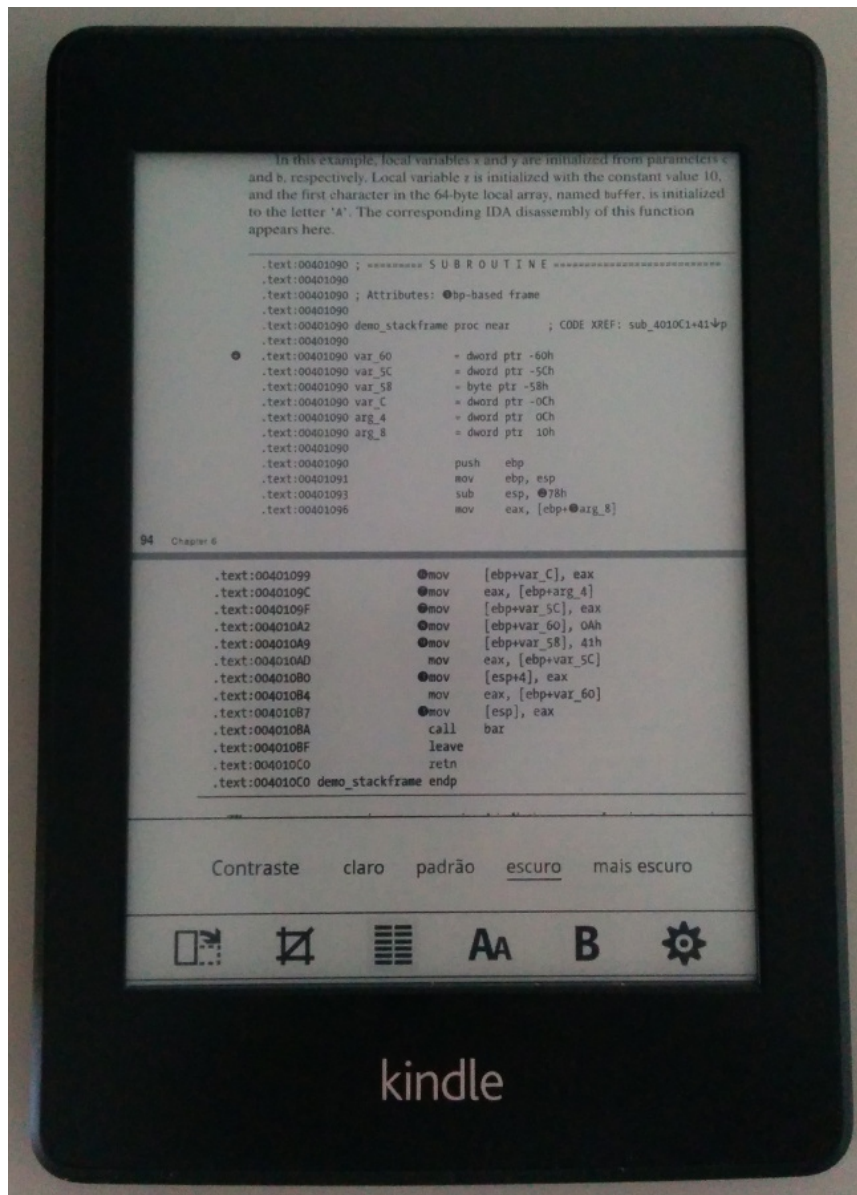


Figura 1 – Dispositivo embarcado Kindle PaperWhite, da empresa Amazon. Na foto, o aparelho aparece executando o software KOREader para a leitura de um livro em formato PDF.

- Mobipocket (MOBI e PRC);
- AZW / AZW3, formatos proprietários da Amazon baseados no formato Mobipocket;
- Arquivos de texto plano (TXT);
- Arquivos PDF.

Os livros disponíveis na loja online do Kindle são comercializados no formato AZW, que suporta refluxo de texto, adequando-se corretamente à tela do dispositivo e proporcionando uma leitura agradável. Entretanto, o aplicativo padrão do Kindle não suporta refluxo em arquivos no formato PDF. É necessário utilizar o recurso de *zoom* e mover constantemente o texto para conseguir ler a maioria dos livros nesse formato. Como a atualização da tela de tinta eletrônica é relativamente lenta, essas operações são desagradáveis e distraem o usuário de sua leitura.

Existem programas (WILLUS, 2011) capazes de preprocessar arquivos PDF no computador do usuário (de forma *offline*), efetuando o refluxo de texto antes que estes sejam transferidos para o Kindle. Entretanto, esse processo aumenta o espaço ocupado pelos arquivos, além de apresentar resultados imperfeitos (como na Figura 2), especialmente para artigos científicos ou livros técnicos com leiaute complicado.

Outra desvantagem do aplicativo padrão do Kindle é a falta de suporte a DjVu (BOTTOU et al., 1998), um dos melhores formatos em termos de taxa de compressão e velocidade de decodificação para livros escaneados. Desta forma, fica prejudicada a leitura de livros antigos, que não possuam reedição digital.

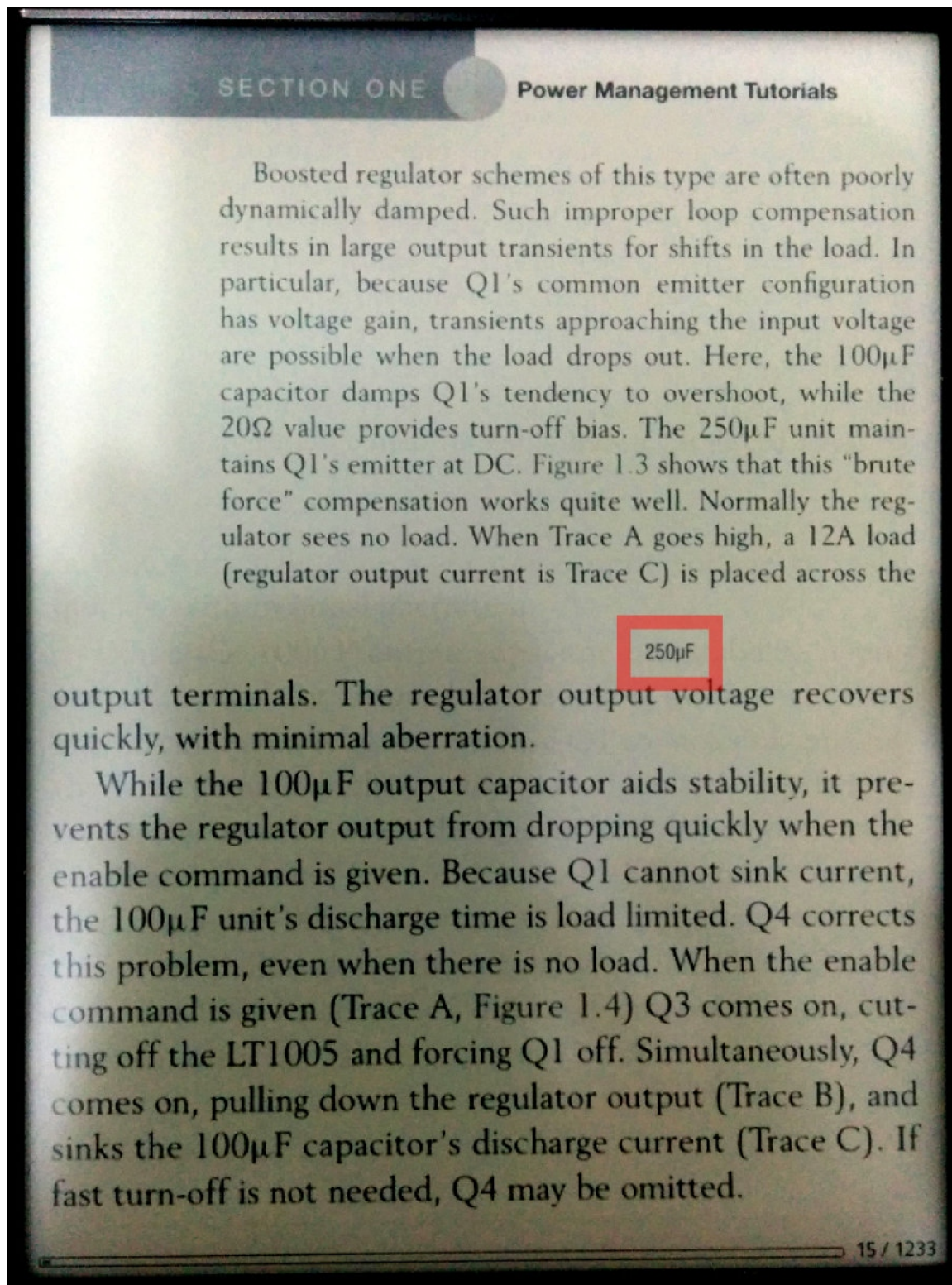


Figura 2 – Trecho de um livro eletrônico preprocessado pelo software de refluxo de texto k2pdfopt (WILLUS, 2011). O valor “250 μ F” (destacado em vermelho) fazia parte de uma figura de diagrama eletrônico, mas é incorretamente reconhecido como parte do texto e inserido no meio de uma frase. Além disso, o tamanho da fonte do texto muda bruscamente após esse ponto.

2.2 Projeto KOREader

O KOREader é um software embarcado alternativo para o Kindle e outros dispositivos para leitura de livros eletrônicos. Trata-se de um projeto de software livre criado inicialmente com o objetivo de suplantar as limitações do Kindle na leitura de acervos preexistentes, que não tenham sido reeditados pela Amazon ou disponibilizados por terceiros em formato adequado ao Kindle.

Recursos e implementação do software

O KOREader suporta os formatos PDF e DjVu com refluxo de texto integrado no próprio dispositivo, utilizando o mesmo método de Willus (2011). A vantagem com relação ao processamento *offline* é que, ao encontrar problemas ocasionados pelo refluxo que dificultem a compreensão de certa parte do texto, o usuário pode desativar o recurso com o simples toque em um menu, visualizando imediatamente a página original. O KOREader também é capaz de ler os formatos EPUB e CHM, naturalmente projetados para suportar refluxo de texto, além dos formatos MOBI e TXT, que já eram suportados pelo software original do Kindle.

Além da meta de suportar vários formatos, o projeto tem por objetivo implementar recursos inovadores que facilitem a vida do usuário. Um exemplo é um sistema multiplataforma de sincronização de livros entre dispositivos em uma rede local que começou a ser implementado recentemente. Esse recurso permite que o usuário mantenha automaticamente a mesma cópia de seu acervo em todos os seus dispositivos Kindle, Kobo e Android que estejam executando o KOREader.

Em vez de utilizar a máquina virtual CVM, como faz o aplicativo padrão da Amazon, o KOREader executa como um processo Linux nativo. O código do projeto é, em grande parte, implementado em Lua (IERUSALIMSKY et al., 1995; IERUSALIMSKY, 2013), uma linguagem de programação dinâmica projetada para ser de fácil aprendizagem e compreensão, expressar estruturas de dados e configurações de forma natural, e permitir uma interação simples com códigos escritos em C ou C++. A implementação de referência da linguagem é disponibilizada pela PUC-Rio na forma de um interpretador

pequeno e portátil, totalmente escrito em C. Entretanto, o KORReader adota outra implementação – o compilador *just-in-time* LuaJIT (PALL, 2007).

Grande parte do LuaJIT é escrito manualmente em linguagem montadora, limitando sua portabilidade com relação à implementação de referência. Ainda assim, o compilador está disponível para várias arquiteturas de processadores: ARM, MIPS, PowerPC e x86 (tanto 32-bits quanto 64-bits). Na arquitetura ARM, o LuaJIT pode ser até 65 vezes mais rápido que a implementação de referência, dependendo do algoritmo executado (Figura 3). Mesmo com esse ganho de desempenho, o executável do LuaJIT é pequeno e consome pouca memória. Outra vantagem do LuaJIT é seu inteligente mecanismo de FFI (*Foreign Function Interface*), que possibilita uma integração ainda mais fácil com a linguagem C – tipos de dados e protótipos de funções podem ser declarados

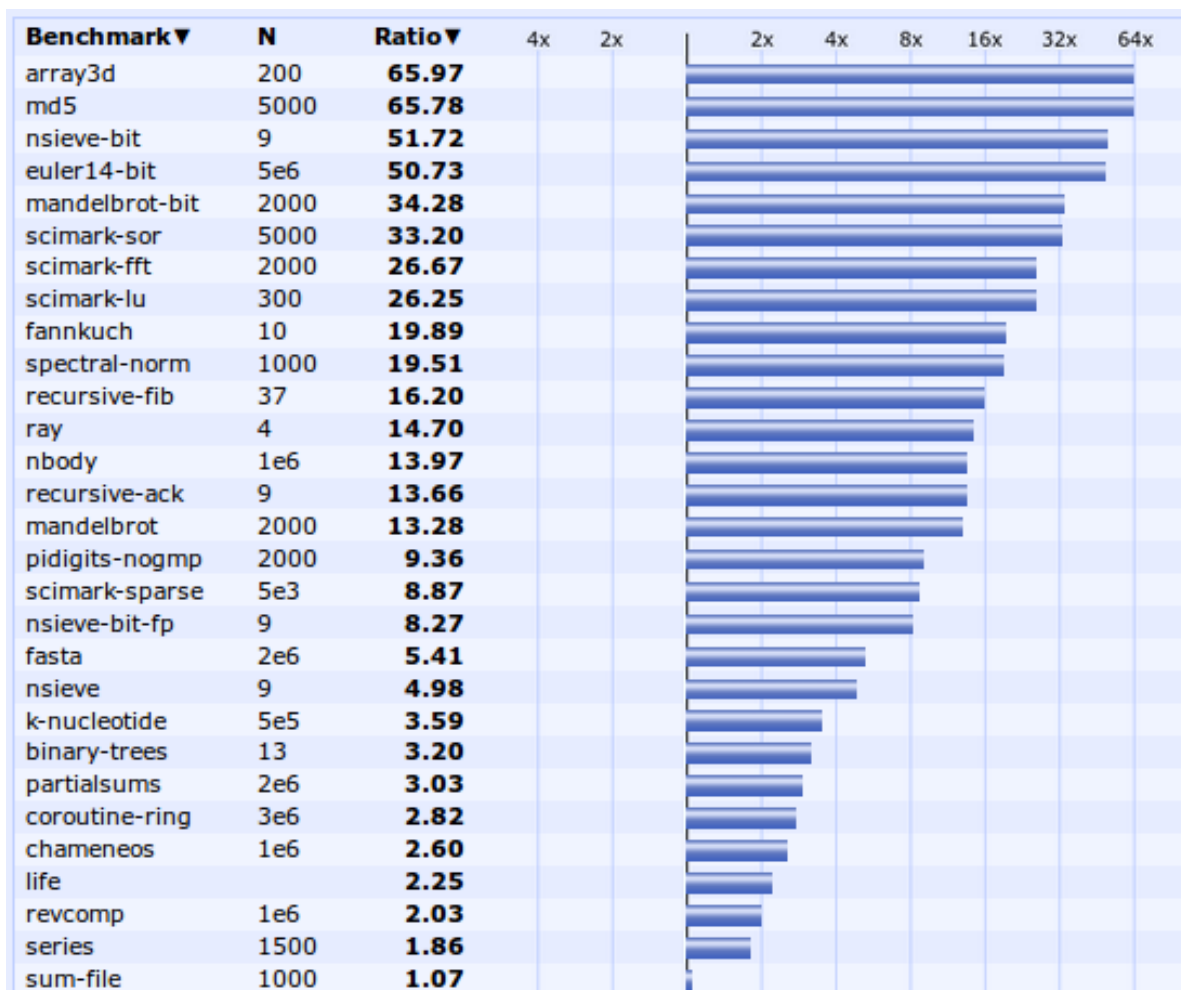


Figura 3 – Benchmark do LuaJIT versus a implementação padrão do Lua em arquitetura ARM. Fonte: Pall (2014).

diretamente em um subconjunto de C, tornando-se automaticamente acessíveis pelo código Lua.

Organização e práticas da equipe

O KOREader é um projeto colaborativo bastante heterogêneo, que conta com a contribuição de desenvolvedores de diversos países, como China, Alemanha, Croácia, França, Armênia, Brasil, Holanda e Itália. O projeto é hospedado no GitHub, e a maioria das discussões entre os desenvolvedores acontece de forma aberta no próprio site.

Toda alteração no repositório do projeto é enviada na forma de uma *pull request*, solicitação que vem acompanhada de uma visualização das diferenças no código provocadas por aquela modificação e de um espaço para discussões (Figura 4).

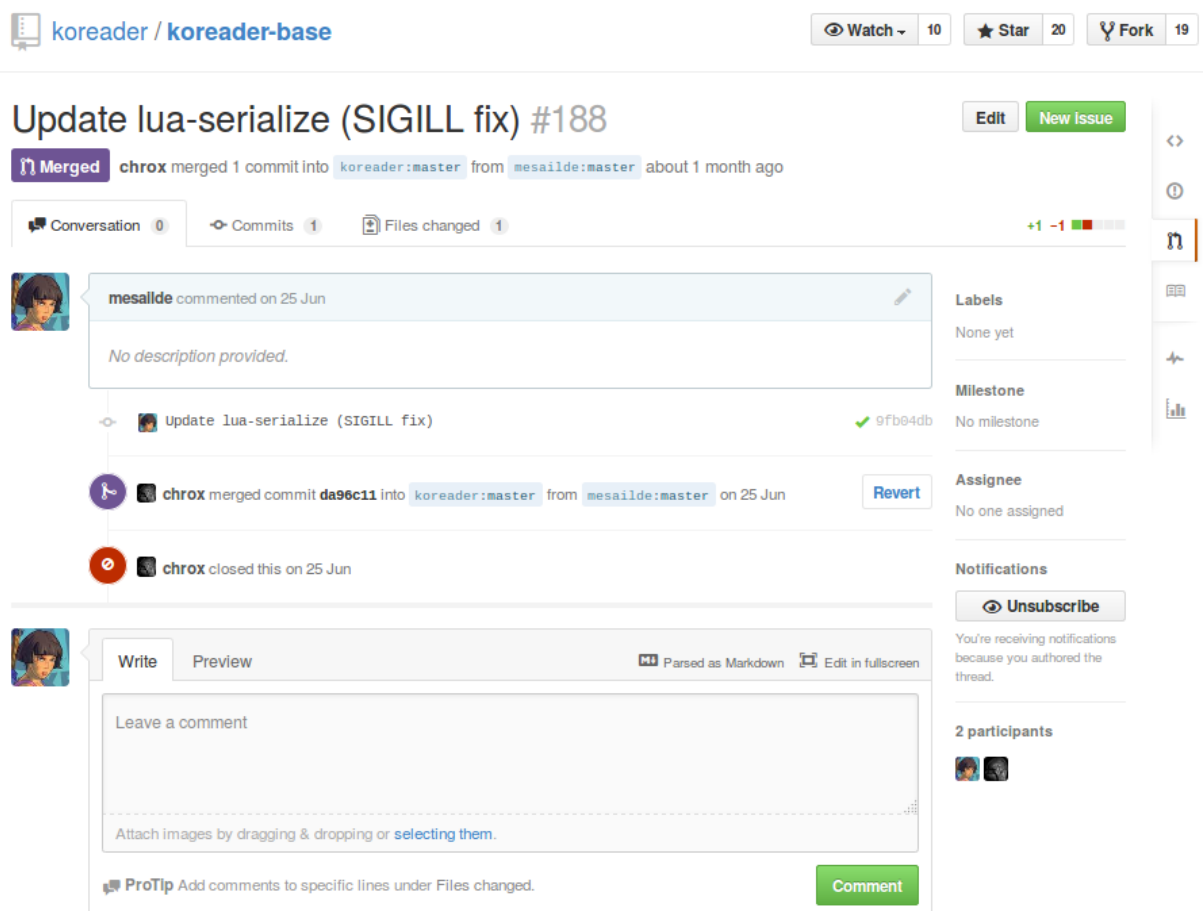


Figura 4 – Exemplo de uma *pull request* na plataforma GitHub. A página principal é um espaço para discussões. Clicando na aba *Files changed*, pode-se visualizar as alterações incluídas na *pull request*.

Pull requests podem ser enviadas por qualquer pessoa cadastrada no GitHub, mas precisam ser aprovadas por um membro autorizado do projeto, ou seja, um usuário que tenha permissões de escrita na linha principal do repositório, da mesma forma que ocorre com outros projetos hospedados na plataforma. Entretanto, a política de revisão de códigos do KORreader vai além, e estabelece que mesmo *pull requests* enviadas por membros autorizados devem passar pela aprovação de um membro diferente daquele que sugeriu a modificação no código.

O KORreader possui uma suíte de testes automatizados implementada com o auxílio da biblioteca Busted (LAWSON, 2014). O projeto adota a plataforma de integração contínua Travis CI (FUCHS et al., 2014), que se integra naturalmente ao GitHub. O Travis CI identifica toda *pull request* ou alteração na linha principal do repositório, inserindo imediatamente em sua fila um processo de compilação e testes automatizados de uma versão do código contendo as mudanças propostas ou efetuadas. Quando disparado, todo o processo é executado em uma máquina virtual isolada. O uso de uma máquina virtual tem duas vantagens: garante-se que uma certa versão do software não interferirá com o teste de uma versão posterior, prejudicando seu diagnóstico, e obtém-se um nível mais elevado de segurança, visto que qualquer usuário do GitHub pode enviar uma *pull request* contendo código malicioso, que será executado automaticamente, sem que antes seja revisado por um humano.

A desvantagem do Travis CI é que os testes executam em um servidor comum com arquitetura x86, onde seria muito trabalhoso e sujeito a erros simular completamente o comportamento do dispositivo embarcado. A proposta deste trabalho é, justamente, desenvolver uma solução análoga ao Travis CI, porém que execute os testes em um dispositivo Kindle.

3 DESENVOLVIMENTO DAS TÉCNICAS

As subseções a seguir discutem os pontos principais da solução de integração contínua desenvolvida com a finalidade de atingir os objetivos desta pesquisa. Os códigos referenciados ao longo do texto estão disponíveis nos anexos, ao final deste trabalho.

3.1 Monitorando alterações no repositório

Para monitorar alterações na linha principal de um repositório no GitHub, adotamos o mesmo método utilizado pelo Travis CI. Trata-se de um mecanismo denominado *webhooks*, no qual o GitHub envia uma requisição HTTP para um endereço definido pelo usuário sempre que ocorrer algum evento no repositório (Figura 5). O *webhook* foi configurado para ser ativado somente no evento *push*, que é disparado quando a linha principal ou algum outro ramo do repositório é alterado.

O evento *push* não ocorre quando o repositório recebe *pull requests*, apenas quando estas são aprovadas. Desta forma, ao contrário do Travis CI, o processo de compilação e testes automatizados de nossa solução não pode ser acionado como resultado da ação de um usuário que não possua autorização de escrita no projeto. Optamos por essa restrição devido ao fato de que os testes precisam ter permissão de superusuário no dispositivo Kindle para ter acesso a todo o hardware embarcado, tornando-se necessário executar apenas códigos de origem confiável, ou seja, gerados ou revisados por membros do projeto.

Para implementar o servidor que recebe as notificações, utilizamos o Turbo Lua (ABRAHAMSEN et al., 2014), um arcabouço de ferramentas (*framework*) para construir aplicativos de rede baseados na arquitetura de um único processo comandado por eventos (CORDEIRO, 2006) e na chamada de sistema *epoll* (LIBENZI, 2002), muito similar a plataformas populares como Node.js e Tornado.

Webhooks / Manage webhook


We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in [our developer documentation](#).

Payload URL

Content type

application/json

Secret

 **Warning:** SSL verification is not enabled for this hook.

Enable SSL verification

Which events would you like to trigger this webhook?

☒ Just the push event.

☐ Send me **everything**.

☐ Let me select individual events.

☒ **Active**
We will deliver event details when this hook is triggered.

Update webhook

Delete webhook

Figura 5 – Tela de configuração de *webhooks* do projeto no GitHub, onde deve ser inserido o endereço do servidor HTTP que monitora as alterações no repositório.

O servidor `ciserver.lua` disponibiliza um caminho `/push_hook` para receber notificações do GitHub e um caminho `/log` para permitir acesso aos registros de cada processo de compilação e teste automatizado (Figura 6).

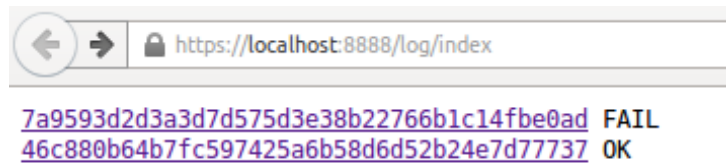


Figura 6 – Listagem dos registros de compilação e testes automatizados disponibilizados pelo servidor de integração contínua para cada alteração (*commit*) efetuada no repositório.

A Figura 7 mostra um exemplo de requisição recebida por meio do caminho `/push_hook`. Ao receber-se uma notificação por este caminho, primeiramente assegura-se que o pedido realmente proceda do GitHub, por meio da autenticação HMAC-SHA1 do conteúdo da requisição utilizando um segredo compartilhado (campo *Secret* na Figura 5) contida no cabeçalho `X-Hub-Signature`. Depois, verifica-se o tipo de evento no cabeçalho `X-GitHub-Event`. Caso seja um evento *push*, decodifica-se o conteúdo da requisição (que é recebido em formato JSON) e extrai-se o identificador da última alteração (*commit*) pertencente àquela versão, contido no campo “*after*”.

Cada identificador de *commit* recebido é validado (deve conter apenas dígitos hexadecimais) e inserido no final de uma fila. Um comando externo (`dispatch.sh`) é disparado sempre que possível, recebendo como argumento o identificador do *commit* que esteja na frente da fila.

Além da autenticação HMAC-SHA1, o servidor de integração contínua adota SSL como uma camada de segurança adicional. Por este motivo, antes de executar o servidor pela primeira vez, um par de chaves SSL deve ser gerado utilizando o `script ssl-genkeys.sh`.


```

POST /push_hook HTTP/1.1
Host: example.com
Accept: */*
User-Agent: GitHub-Hookshot/eddbeea
X-GitHub-Event: push
X-GitHub-Delivery: af72684c-18a3-11e4-9ef2-ecf5d3306f07
content-type: application/json
X-Hub-Signature: sha1=b00ba0a30f024f3eb76efbfcedd9f0a02a8dd5d2
Content-Length: 4919

{
  "ref": "refs/heads/master",
  "after": "ddbfec9198c669f88c22b6b6ce8124f9cc4f8cd0",
  "before": "0000000000000000000000000000000000000000",
  "created": true,
  "deleted": false,
  "forced": true,
  "compare": "https://github.com/usuario/hooktest/commit/ddbfec9198c6",
  "commits": [
    {
      "id": "ddbfec9198c669f88c22b6b6ce8124f9cc4f8cd0",
      "distinct": true,
      "message": "first commit",
      "timestamp": "2014-07-02T08:13:19-03:00",
      "url": "https://github.com/usuario/hooktest/commit/ddbfec9198c669f88c22b6b6ce8124f9cc4f8cd0",
      "author": {
        "name": "Usuário",
        "email": "email@example.com",
        "username": "usuario"
      },
      "committer": {
        "name": "Usuário",
        "email": "email@example.com",
        "username": "usuario"
      },
      "added": [ "README.md" ],
      "removed": [],
      "modified": []
    }
  ],
  "head_commit": { /* [...] */ },
  "repository": { /* [...] */ },
  "pusher": {
    "name": "usuario",
    "email": "email@example.com"
  }
}

```

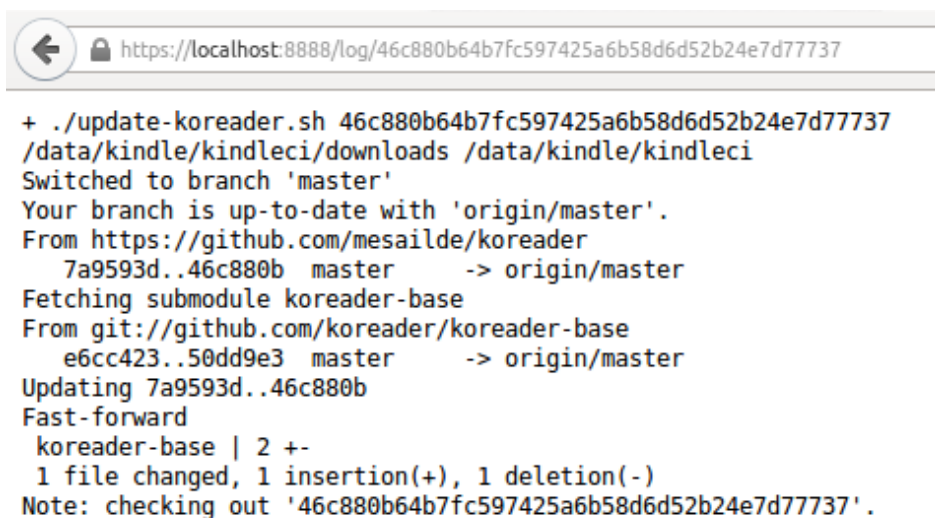
Figura 7 – Exemplo de requisição HTTP enviada pelo GitHub notificando a ocorrência de um evento do tipo *push*.

3.2 Compilando automaticamente o KOREader

O processo de compilação e testes automatizados é comandado por um conjunto de *scripts* escrito na linguagem do *shell* Bash (NEWHAM, 2005). O *script* principal é o `dispatch.sh`, executado pelo servidor de integração contínua, que por sua vez dispara os *scripts* denominados `update-koreader.sh`, `make-koreader.sh` e `run-tests.sh`. A Figura 8 mostra o início da saída gerada por esse processo.

O *script* `update-koreader.sh` verifica se já existe uma cópia local do repositório do KOREader. Caso não exista, efetua uma nova cópia (clona o repositório). Caso exista, atualiza a cópia existente. Por fim, o *script* faz com que a cópia de trabalho (*checkout*) corresponda com o *commit* passado como argumento. Isso é necessário caso o repositório receba novos conjuntos de alterações (*pulls*) enquanto outros testes já estiverem em curso, caso no qual o *commit* contido na frente da fila pode não corresponder à última versão do código disponível no repositório, que será testada apenas posteriormente.

O *script* `make-koreader.sh` utiliza um compilador cruzado (*cross-compiler*) para gerar executáveis do KOREader para arquitetura ARM, ainda que o servidor de integração contínua seja executado em uma máquina com arquitetura x86. A versão do compilador utilizada é definida no arquivo `config.env`.



```
+ ./update-koreader.sh 46c880b64b7fc597425a6b58d6d52b24e7d77737
/data/kindle/kindleci/downloads /data/kindle/kindleci
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
From https://github.com/mesailde/koreader
 7a9593d..46c880b master    -> origin/master
Fetching submodule koreader-base
From git://github.com/koreader/koreader-base
 e6cc423..50dd9e3 master    -> origin/master
Updating 7a9593d..46c880b
Fast-forward
 koreader-base | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
Note: checking out '46c880b64b7fc597425a6b58d6d52b24e7d77737'.
```

Figura 8 – Início de um registro de compilação e testes automatizados sendo acessado por meio do navegador.

São disponibilizadas duas configurações diferentes de compilação: *debug* e *release*. A configuração *debug* faz com que os executáveis finais contendam símbolos de depuração, que são referências a nomes de variáveis, funções e números de linha no código fonte, permitindo que depuradores como o gdb (GATLIFF, 1999) possam relacionar o código de máquina com o código fonte C ou C++ original. Além disso, ao ser utilizada essa configuração, o *script* habilita várias opções de compilação do LuaJIT que facilitam a identificação de erros de programação, que por padrão viriam desabilitadas devido a questões de desempenho:

- `LUA_USE_APICHECK`: Habilita verificações no LuaJIT que ajudam a encontrar erros em módulos escritos em C que estejam disponibilizados ao código Lua pelo método convencional (sem FFI), ou seja, com o uso da Lua C API.
- `LUA_USE_ASSERT`: Habilita asserções que verificam a integridade do funcionamento de componentes sensíveis do LuaJIT (por exemplo o coletor de lixo), que podem ser prejudicados por uma corrupção de memória causada por erros de programação no código do projeto.
- `LUAJIT_USE_VALGRIND`: Compatibiliza o uso do LuaJIT com a ferramenta Valgrind (NETHERCOTE e SEWARD, 2007), que detecta acessos inválidos à memória.
- `LUAJIT_USE_SYSMALLOC`: Utiliza o alocador de memória (`malloc`) do sistema em vez do alocador customizado do LuaJIT. Necessário para que o Valgrind consiga detectar acessos inválidos a blocos de memória alocados pelo código Lua.
- `LUAJIT_USE_GDBJIT`: Permite que o gdb seja usado para depurar instruções em linguagem de máquina geradas a partir do código Lua pelo compilador *just-in-time* do LuaJIT.

Já a configuração *release* elimina símbolos de depuração e otimiza o software visando a liberação de uma versão final. No entanto, adotamos por padrão a configuração *debug*, por esta permitir identificar um número maior de problemas durante os testes e facilitar qualquer depuração que se faça necessária.

Outra função do *script* é copiar os testes automatizados para a árvore de diretórios

onde os executáveis do KOREader foram gerados após a compilação, além dos dados necessários à execução de alguns testes, por exemplo os arquivos de treinamento da Tesseract, uma biblioteca de reconhecimento óptico de caracteres (*optical character recognition* – OCR).

O *script* `run-tests.sh` executa os testes automatizados no dispositivo, e é abordado mais à frente, na Subseção 3.5.

3.3 Mapeando o KOREader via rede para o dispositivo

Muitos dispositivos, incluindo o Kindle, utilizam memória Flash para armazenar seus softwares embarcados e outros dados. Entretanto, memórias Flash possuem uma quantidade limitada de ciclos de escrita em cada bloco de dados, e desgastam muito mais rápido que discos rígidos quando são expostas a escritas constantes.

Por esse motivo, seria imprudente copiar o KOREader para a memória interna do Kindle toda vez que uma nova versão fosse compilada, ainda mais quando os executáveis são compilados com símbolos de depuração, o que os torna muito maiores. Além disso, alguns testes automatizados podem gerar arquivos temporários, colaborando ainda mais para o desgaste da memória Flash do dispositivo.

Uma alternativa é mapear o diretório onde o KOREader foi compilado na máquina hospedeira (aquela que executa o servidor de integração contínua) para o dispositivo, fazendo com que os dados sejam transferidos sob demanda, via rede, para a memória RAM do Kindle.

O SSH (YLONEN, 1996) é um mecanismo de acesso remoto facilmente instalável e configurável em praticamente qualquer sistema do tipo UNIX. Apesar de originalmente projetado para permitir o acesso a uma linha de comandos do terminal remoto, existem hoje diversos serviços associados ao SSH, dentre eles o protocolo SFTP de transferência de arquivos. Um utilitário denominado `sshfs` (HOSKINS, 2006) permite mapear (montar) localmente qualquer diretório do terminal remoto que esteja acessível via SFTP.

De fato, essas ferramentas são tão ubíquas que foram portadas até mesmo para o Kindle. Um pacote de utilitários chamado Kindle USBNetwork (BIGUET, 2014) disponibiliza um *driver* que permite que a porta USB do Kindle seja enxergada pelo computador hospedeiro como se fosse uma placa de rede em vez de um dispositivo de armazenamento. Além desse *driver*, o pacote acompanha versões prontas para uso no Kindle do cliente e do servidor SSH e do utilitário sshfs.

O pacote USBNetwork é configurado por um conjunto de arquivos inserido no diretório `usbnet/etc` dentro da área comumente acessível quando o Kindle é enxergado como dispositivo de armazenamento pelo computador. Por padrão, o dispositivo utiliza o endereço IP 192.168.15.244 quando habilitada a rede sobre USB, mas isso pode ser alterado por meio do arquivo `usbnet/etc/config`. Esse endereço IP deve ser configurado com um apelido `kindle` na máquina hospedeira, adicionando a seguinte entrada ao arquivo `~/.ssh/config`:

```
Host kindle
    HostName 192.168.15.244
    User root
```

Um apelido (`master`) para o endereço do computador hospedeiro também deve ser definido no Kindle, por meio do arquivo `usbnet/etc/dot.ssh/config`. Se a interface de rede USB for configurada no computador com o endereço IP 192.168.15.201 (como na Figura 9), a seguinte entrada deve ser adicionada:

```
Host master
    HostName 192.168.15.201
    User kindle
```

Para permitir que o Kindle seja acessado pelo servidor de integração contínua automaticamente, evitando que uma senha seja requisitada a cada teste, é necessário copiar a chave pública SSH do computador hospedeiro para o arquivo `usbnet/etc/authorized_keys`. O inverso também é necessário – o Kindle precisa ter acesso a um usuário (no exemplo, o usuário `kindle`) da máquina hospedeira, de forma que possa mapear via rede o diretório onde o KOREader foi compilado. Para isso, é necessário gerar um par de chaves privada e pública (`id_rsa` e `id_rsa.pub`) com o

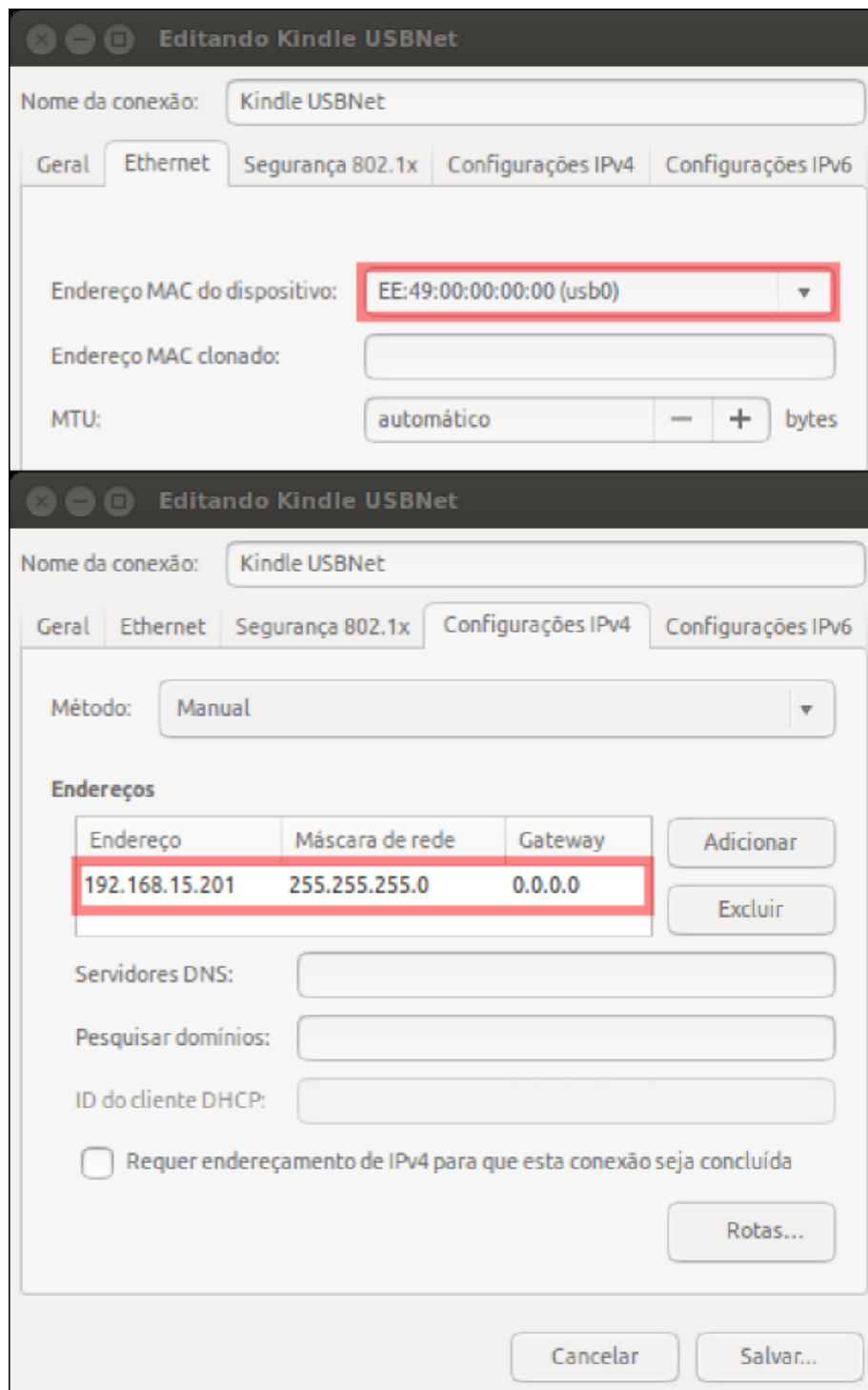


Figura 9 – Configuração da interface de rede USBNetwork na ponta do computador usando o NetworkManager.

utilitário `ssh-keygen` e copiá-las para o diretório `usbnet/etc/dot.ssh`. A chave pública do Kindle (`id_rsa.pub`) deve ser, então, inserida no arquivo `~/.ssh/authorized_keys` do computador hospedeiro.

3.4 Adaptando o Valgrind e o Busted para o Kindle

O *script* `make-infra.sh` é responsável por compilar a infraestrutura necessária à execução do *framework* de testes Busted no dispositivo Kindle, além da ferramenta Valgrind (não-correlata ao Busted). Idealmente, esse *script* deve ser executado apenas uma vez, quando o servidor de integração contínua estiver sendo instalado, característica que o diferencia do `make-koreader.sh`.

O processo de compilação do Valgrind não tem muitos segredos. É necessário apenas um pequeno truque para que os *scripts* de configuração identifiquem o compilador como se este incluísse a versão da arquitetura ARM no sufixo de seu executável (ou seja, “`armv7-`”), considerando que a maioria dos compiladores cruzados adota apenas “`arm-`” como sufixo.

Entretanto, para executar, o Valgrind necessita ter disponível uma versão da biblioteca padrão da linguagem C (`libc`) que inclua símbolos de depuração. Em vez de compilar nossa própria versão da biblioteca, um processo demorado e sujeito a erros, identificamos a versão da biblioteca por meio do seguinte comando, executado no Kindle:

```
[root@kindle root]# strings /lib/libc.so.6 | grep 'GNU C'
GNU C Library (Ubuntu EGLIBC 2.12.1-0ubuntu6) stable release
version 2.12.1, by Roland McGrath et al.
Compiled by GNU CC version 4.4.5 20100909 (prerelease).
```

Um pacote intitulado `libc6-dbg` contendo essa mesma versão da biblioteca foi identificado em repositórios antigos da distribuição Ubuntu Linux para ARM. O *script* descompacta esse pacote, instalando-o no local adequado dentro da árvore de diretórios a ser disponibilizada via rede para o Kindle.

Para a instalação do Busted, o *script* necessita que uma versão para ARM do LuaJIT ou da implementação de referência do Lua esteja disponível na árvore de diretórios. Por esse motivo, compilamos o LuaJIT ainda que outra cópia vá ser gerada posteriormente pelo processo de compilação do KOREader.

Por fim, utilizamos o LuaRocks, um repositório de bibliotecas para Lua, para instalar o Busted e todas as suas dependências. Para isso, o *script* gera um arquivo de configuração definindo uma árvore secundária do LuaRocks, separada daquela que seria utilizada para bibliotecas instaladas na máquina hospedeira. Além disso, configura uma série de variáveis de ambiente que forçam o compilador cruzado para ARM a ser utilizado durante a instalação de bibliotecas que contenham código em C, como a LuaFileSystem, da qual o *framework* Busted depende.

3.5 Executando remotamente o Busted

O Busted é executado remotamente pelo *script* `run-tests.sh`. Opcionalmente, pode ser fornecida a esse *script* uma opção de linha de comando `-use-valgrind`, que o instrui a executar o Busted de forma monitorada pelo Valgrind, permitindo detectar acessos inválidos à memória ocorridos durante os testes. Entretanto, essa opção não é utilizada por padrão, pois a execução de programas sob o Valgrind é muito mais demorada.

Ao ser executado, o *script* cria dinamicamente um segundo *script* de mesmo nome, porém localizado dentro do diretório a ser mapeado para o dispositivo Kindle. O *script* gerado é responsável por configurar as variáveis de ambiente necessárias para que o Busted encontre a localização de suas dependências, que haviam sido instaladas por meio do LuaRocks.

O dispositivo Kindle é, então, acessado via SSH a partir da máquina hospedeira. Verifica-se se o diretório a ser mapeado já está montado no dispositivo, executando o `sshfs` em caso negativo. Em seguida, executa-se o *script* gerado, que se encarrega de disparar os testes automatizados do Busted.

4 APLICAÇÃO DAS TÉCNICAS

Este capítulo descreve o resultado dos testes automatizados executados no Kindle utilizando as técnicas desenvolvidas no capítulo anterior. É demonstrado que testes que executam normalmente em um computador podem apresentar problemas quando executados no dispositivo em si. São explicadas as técnicas de depuração utilizadas para resolver esses problemas e as alterações enviadas como contribuição ao projeto KOREader.

4.1 Suítes de teste disponíveis

O KOREader dispõe de duas suítes de teste, chamadas de “testes de *backend*” e “testes de *frontend*”:

- Os testes de *backend* tem como seu alvo algumas das funções das bibliotecas utilizadas pelo KOREader, módulos em C que fazem parte do projeto, além de rotinas em Lua que façam uso da FFI do LuaJIT.
- Os testes de *frontend* visam avaliar o código como um todo, incluindo rotinas que trabalham com renderização da interface gráfica.

Desta forma, os testes de *backend* são mais próximos do conceito de teste unitário, enquanto os de *frontend* são basicamente testes de integração.

4.2 Identificação de testes limitados pelo hardware

Dentre os testes de *frontend*, identificaram-se alguns que não eram capazes de executar no Kindle devido a limitações de memória:

- *PDF rendering benchmark*
- *PDF reflowing benchmark*
- *Cache module*

Monitorando os recursos de hardware do dispositivo com o auxílio do utilitário `top` em uma sessão SSH separada, percebeu-se que a memória era esgotada durante os testes acima. A característica em comum entre esses testes é o uso do mesmo arquivo de dados (`sample.pdf`). Abrindo-se esse arquivo diretamente no KOREader, percebe-se que ao visualizar logo uma das primeiras páginas o software é encerrado pelo mesmo motivo – falta de memória.

A solução para este problema foi alterar o arquivo de dados usado por esses testes para um que pudesse ser lido corretamente no KOREader em ambiente real (Kindle). Foi escolhido o arquivo `2col.pdf`, que já acompanhava os dados da suíte de testes do projeto. O *script* `quirks.sh` é responsável por realizar essa modificação no código dos testes automatizados antes de executá-los.

Um estudo mais aprofundado poderia tentar otimizar a biblioteca MuPDF utilizada pelo KOREader para que esta pudesse renderizar o arquivo `sample.pdf` utilizando menos memória, porém essa tarefa acabou por não ser incluída no escopo deste trabalho.

4.3 Depuração de erros de instrução ilegal

Mesmo após adaptação para os limites de memória do dispositivo, dois testes da suíte de *frontend* relacionados ao *cache module* ainda falhavam, desta vez com o software sendo encerrado pelo sinal SIGILL (instrução ilegal, malformada ou privilegiada).

O primeiro passo para encontrar a origem do problema foi executar os testes dentro do depurador gdb. Para isso, modifica-se o arquivo `run-tests.sh`, inserindo-se o comando `"gdbserver :2345"` antes da chamada ao LuaJIT, da mesma forma que se aciona o Valgrind por intermédio da variável `${VALGRIND_CMD}` presente no código do

script. O gdbserver carrega o programa na memória do Kindle e, antes de continuar a execução, fica esperando uma conexão via rede de um computador hospedeiro que faça uso do cliente gdb-multiarch.

```
(gdb) set architecture arm
(gdb) set gnutarget elf32-littlearm
(gdb) file luajit
(gdb) set solib-search-path libs:common:/home/mesailde/kindle-ci/build/
luajit/lib/lua/5.1
(gdb) set debug-file-directory /home/mesailde/kindle-ci/build/valgrind/
debug
(gdb) target remote 192.168.15.244:2345
(gdb) continue

[...]
```

Program received signal SIGILL, Illegal instruction.
0x40706e68 in ?? () from common/serialize.so

```
(gdb) bt
#0 0x40706e68 in ?? () from common/serialize.so
#1 0x40706e5c in ?? () from common/serialize.so
Backtrace stopped: previous frame identical to this frame (corrupt stack?)

(gdb) disassemble $pc-32,+64
Dump of assembler code from 0x40706e48 to 0x40706e88:
0x40706e48: mov    r2, #0
0x40706e4c: mov    r0, r4
0x40706e50: str    r2, [r1, #-8]!
0x40706e54: mov    r2, #4
0x40706e58: bl     0x40704eb8
0x40706e5c: subs   r5, r0, #0
0x40706e60: beq    0x40706f44
0x40706e64: vldr   s0, [r5]
=> 0x40706e68: vcvtf.f64.s32    d0, s0
0x40706e6c: add    sp, sp, #12
0x40706e70: pop    {r4, r5, r6, r7, r8, r9, pc}
0x40706e74: add    r5, sp, #8
0x40706e78: mov    r0, r1
0x40706e7c: mov    r8, #0
0x40706e80: mov    r9, #0
0x40706e84: mov    r2, #8
End of assembler dump.
```

Figura 10 – Saída do depurador gdb ao analisar o erro de instrução ilegal que ocorria na revisão 217f56590b5eb5fc08a286d636a8a237ab68e930 da biblioteca lua-serialize quando executada em arquitetura ARM. Esse erro não ocorria em ambientes x86.

O gdb-multiarch precisa de alguns comandos de configuração para identificar a arquitetura de processador remota, o executável principal, o endereço IP da outra ponta e os caminhos de diretórios onde as bibliotecas podem ser encontradas para leitura, como ilustra a Figura 10.

Quando o programa é abortado, o gdb-multiarch volta a mostrar sua linha de comandos, e um *backtrace* pode ser obtido por meio do comando *bt*. O *backtrace* mostra em qual ponto o código parou, revelando que este pertence à biblioteca lua-serialize.

Em seguida, o comando *disassemble* é utilizado para exibir, em linguagem montadora, as instruções existentes ao redor do ponto onde o programa foi abortado. Uma seta (*=>*) aponta para a próxima instrução que seria executada caso a falha não houvesse ocorrido.

Com essas informações, sabe-se que a falha ocorreu durante a execução da instrução “*vldr s0, [r5]*”, porém não se sabe a qual ponto do código C essa instrução corresponde, pois a biblioteca não foi compilada com símbolos de depuração.

Ainda assim, é possível continuar em busca da origem do problema, e de uma forma bastante instrutiva, que nos força a explicar a causa raiz. Entretanto, é importante deixar claro que o procedimento a seguir seria bem mais simples se o projeto houvesse sido compilado com a configuração *debug* passada ao *script* *make-koreader.sh*, pois seria possível identificar imediatamente (no *backtrace*) a linha do código C onde o problema ocorre.

```
(gdb) x/64cx 0x40746e58
0x40746e58: 0x16 0xf8 0xff 0xeb 0x00 0x50 0x50 0xe2
0x40746e60: 0x37 0x00 0x00 0x0a 0x00 0x0a 0x95 0xed
0x40746e68: 0xc0 0x0b 0xb8 0xee 0x0c 0xd0 0x8d 0xe2
0x40746e70: 0xf0 0x83 0xbd 0xe8 0x08 0x50 0x8d 0xe2
0x40746e78: 0x01 0x00 0xa0 0xe1 0x00 0x80 0xa0 0xe3
0x40746e80: 0x00 0x90 0xa0 0xe3 0x08 0x20 0xa0 0xe3
0x40746e88: 0xf8 0x80 0x65 0xe1 0x05 0x10 0xa0 0xe1
0x40746e90: 0x08 0xf8 0xff 0xeb 0x00 0x70 0x50 0xe2
```

Figura 11 – Leitura do conteúdo de uma posição de memória com o depurador gdb.

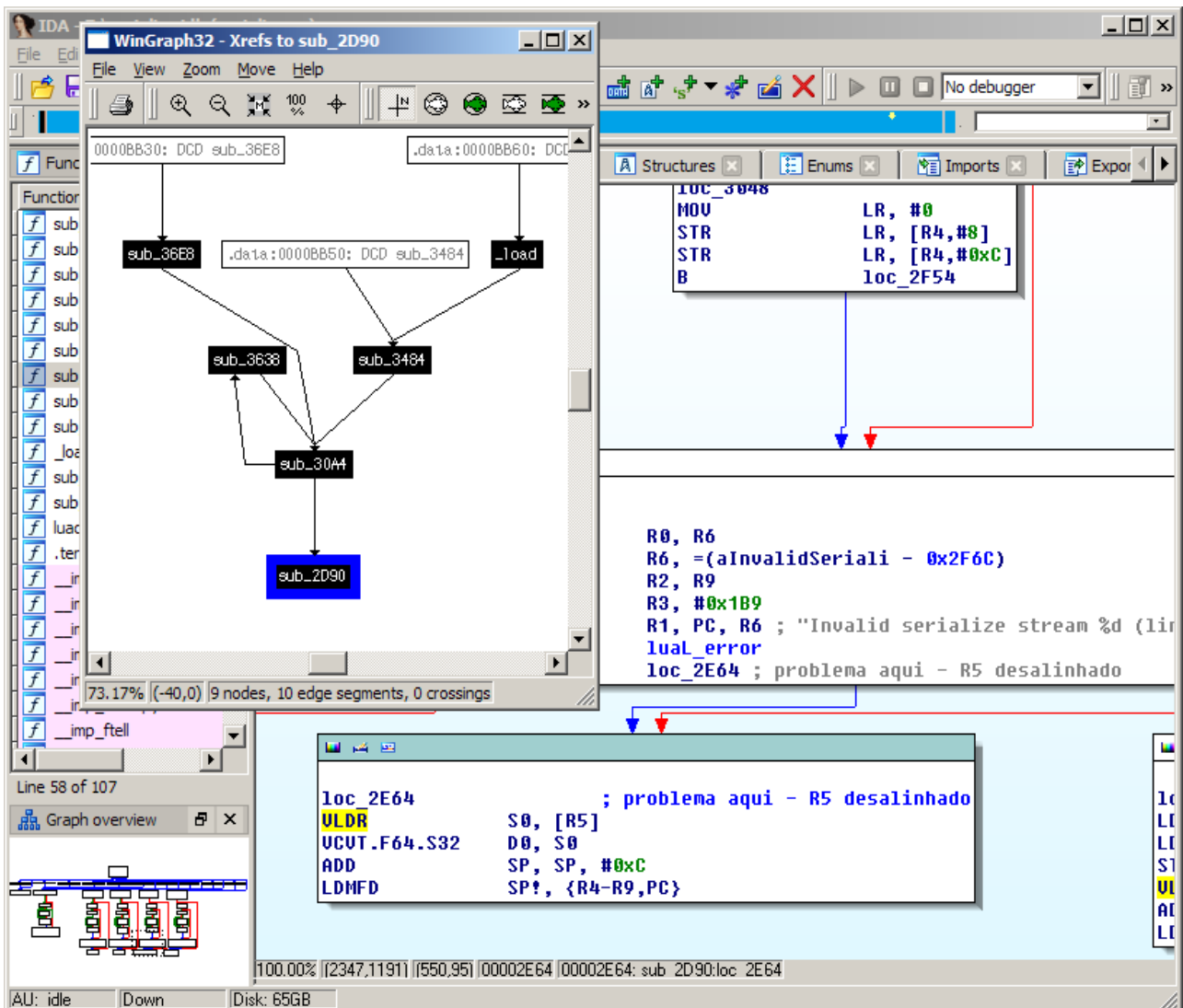


Figura 12 – Trecho do código da biblioteca lua-serialize em linguagem de máquina analisado com o auxílio da ferramenta IDA Pro.

Primeiramente, o conteúdo da memória em torno do endereço onde ocorreu a falha é consultado como na Figura 11. Essa sequência de bytes pode ser procurada em um programa de desmontagem de executáveis (*disassembler*), por exemplo o IDA Pro (Figura 12), de forma a identificar qual ponto da biblioteca estava sendo executado naquele momento. Uma alternativa a esse método é identificar o endereço no qual a biblioteca estava carregada na memória durante aquela execução, com o comando `info sharedlibrary` do `gdb` (Figura 13). O endereço da instrução (`0x40706e64`) deve ser subtraído do endereço onde a biblioteca está carregada (`0x40704c08`) e somado ao

```
(gdb) info sharedlibrary
warning: Could not load shared library symbols for /lib/libpthread.so.0.
Do you need "set solib-search-path" or "set sysroot"?
From          To          Syms Read  Shared Object Library
-----
                No          /lib/ld-linux.so.3
                No          /usr/lib/libenvload.so
                No          /lib/libm.so.6
                No          /lib/libdl.so.2
                No          /lib/libgcc_s.so.1
                No          /lib/libc.so.6
0x401d902c     0x401db098  Yes       /home/mesailde/kindle-
ci/build/luajit/lib/luajit/5.1/lfs.so
0x40704c08     0x40707840  Yes (*)   common/serialize.so
0x407110a0     0x40712030  Yes (*)   /home/mesailde/kindle-ci/work/
koreader/koreader-arm-kindle5-linux-gnueabi/koreader/common/
libluacompat52.so
0x4071c834     0x4071ead4  No        libs/libkoreader-pdf.so
                No          /lib/libpthread.so.0
0x407673b0     0x40837870  No        /home/mesailde/kindle-ci/work/
koreader/koreader-arm-kindle5-linux-gnueabi/koreader/libs/libmupdf.so
0x4097e708     0x409d1ab8  No        /home/mesailde/kindle-ci/work/
koreader/koreader-arm-kindle5-linux-gnueabi/koreader/libs/
libk2pdfopt.so.2
0x40d2d118     0x40d599c4  No        /home/mesailde/kindle-ci/work/
koreader/koreader-arm-kindle5-linux-gnueabi/koreader/libs/libjpeg.so.9
0x40d6c77c     0x40dc3134  No        /home/mesailde/kindle-ci/work/
koreader/koreader-arm-kindle5-linux-gnueabi/koreader/libs/
libfreetype.so.6
0x40e55a90     0x40fe75d8  No        libs/libtesseract.so.3
0x410e4dd0     0x4120d050  No        libs/liblpt.so.3
0x41280af8     0x412f344c  No        /home/mesailde/kindle-
ci/work/koreader/koreader-arm-kindle5-linux-gnueabi/koreader/
libstdc++.so.6
0x413156d0     0x41326d30  No        /home/mesailde/kindle-ci/work/
koreader/koreader-arm-kindle5-linux-gnueabi/koreader/libs/libz.so.1
0x4133b1f0     0x41364878  No        libs/libpng16.so.16
(*): Shared library is missing debugging information.
```

Figura 13 – O gdb é capaz de listar as bibliotecas dinâmicas em uso por um executável, juntamente com o endereço onde estão carregadas.

endereço VMA (*virtual memory area*) da seção de código (.text) da biblioteca (0xc08), que pode ser consultado com o auxílio da ferramenta de linha de comando objdump ou por meio do próprio *disassembler*:

endereço_instrução – endereço_carga_biblioteca + VMA = endereço_binário

0x40706e64 – 0x40704c08 + 0xc08 = 0x2e64

```

437:  case 4: {
438:      int n = 0;
439:      int * pn = rb_read(rb,&n,4);
440:      if (pn == NULL)
441:          _invalid_stream(L,rb);
442:      return *pn;
443:  }

```

Figura 14 – Arquivo de código fonte serialize.c em torno da linha 441.

Observando o código no *disassembler* (Figura 12), nota-se que existe uma chamada `luaL_error` com a mensagem “Invalid serialize stream %d (line: %d)” muito próxima ao ponto onde o código falha. O terceiro argumento passado a essa chamada (contido no registrador R3) é substituído na parte da mensagem que informa o número da linha, e vale `0x1B9` ou, na base decimal, 441. A Figura 14 mostra o código fonte em torno dessa linha. Comparando com o código em linguagem montadora, chega-se à conclusão de que a instrução `vldr` é gerada como parte da compilação da linha 442 do código:

- `vldr s0, [r5]`
Carrega um inteiro contido no endereço dado por `pn` (contido no registrador `r5`) para o registrador `s0`, executando portanto a operação de desreferência de ponteiro (`*pn`);
- `vcvt.f64.s32 d0, s0`
Converte o inteiro `s0` para um ponto flutuante de precisão dupla (64 bits), armazenando-o no registrador `d0`. Essa operação é realizada pois a função em questão retorna o tipo `double`.
- `add sp, sp, #12`
Restaura o ponteiro de pilha para o ponto original. Essa operação é realizada como parte do retorno da função.
- `pop {r4, r5, r6, r7, r8, r9, pc}`
Retorna da função, restaurando os registradores listados para os valores originais, contidos na pilha.

Tabela 1 - Requisitos de alinhamento de instruções de carga/armazenamento. Fonte: Adaptado de ARM (2014, p. 108).

Instruções	Verificação de alinhamento	Resultado se verificação falhar e ...	
		SCTLR.A for igual a 0	SCTLR.A for igual a 1
LDRB, LDREXB, LDRBT, LDRSB, LDRSBT, STRB, STREXB, STRBT, SWPB, TBB	Nenhum	–	–
LDRH, LDRHT, LDRSH, LDRSHT, STRH, STRHT, TBH	<i>Halfword</i> (2 bytes)	Acesso não-alinhado	Falha
LDREXH, STREXH	<i>Halfword</i> (2 bytes)	Falha	Falha
LDR, LDRT, STR, STRT PUSH, somente codificação T3 ou A2 POP, somente codificação T3 ou A2	<i>Word</i> (4 bytes)	Acesso não-alinhado	Falha
LDREX, STREX	<i>Word</i> (4 bytes)	Falha	Falha
LDREXD, STREXD	<i>Doubleword</i> (8 bytes)	Falha	Falha
Todas as formas de LDM e STM, LDRD, RFE, SRS, STRD, SWP PUSH, exceto codificação T3 ou A2 POP, exceto codificação T3 ou A2	<i>Word</i> (4 bytes)	Falha	Falha
LDC, LDC2, STC, STC2	<i>Word</i> (4 bytes)	Falha	Falha
VLDM, VLD R, VPOP, VPUSH, VSTM, VSTR	<i>Word</i> (4 bytes)	Falha	Falha
VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4, todos com alinhamento padrão	Tamanho do elemento	Acesso não-alinhado	Falha
VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4, todos com :<align> especificado	Como especificado pelo :<align>	Falha	Falha

A Tabela 1, retirada da documentação da arquitetura ARM, indica que a instrução `vldr` possui requisitos estritos de alinhamento. O valor do registrador R5 precisa sempre ser um múltiplo de quatro, senão o programa realmente será abortado.

Em um código de serialização de dados, essa condição pode facilmente deixar de ser atendida. A Figura 15 ilustra uma situação em que o tipo de um campo serializado é representado por um byte, com o valor do campo inserido logo a seguir. No caso de um campo numérico contendo um inteiro de 32 bits a ser lido com `vldr`, o endereço facilmente se desalinhará de múltiplos de quatro bytes, devido a estar disposto após uma informação de um único byte.

O motivo pelo qual esse alinhamento é esperado pelo código de máquina pode ser explicado pelo padrão da linguagem C, como transcrito abaixo:

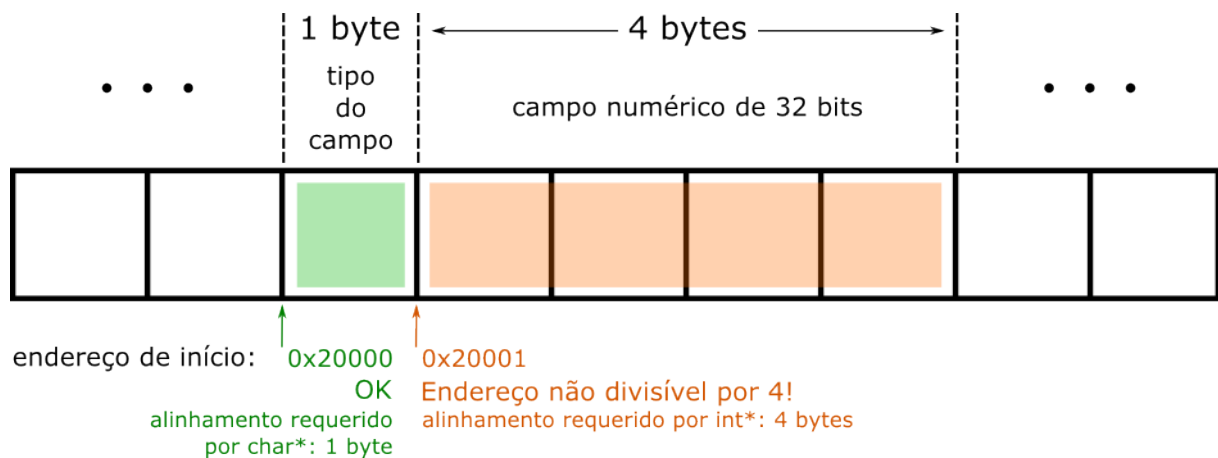


Figura 15 – Estrutura de serialização de objetos onde um campo é representado por um byte indicando seu tipo seguido de seu valor. Caso o ponteiro para o valor do campo seja desreferenciado diretamente, é grande o risco de acesso não alinhado à memória.

Complete object types have alignment requirements which place restrictions on the addresses at which objects of that type may be allocated. An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type: stricter alignment can be requested using the `_Alignas` keyword.

(ISO/IEC 9899:201x, p. 48)

Ou, traduzido para o Português:

Tipos completos de objetos possuem requisitos de alinhamento que impõem restrições sobre os endereços nos quais objetos daquele tipo podem estar alocados. Um alinhamento é um inteiro definido pela implementação que representa o número de bytes entre endereços sucessivos nos quais um dado objeto pode ser alocado. Um tipo de objeto impõe um requisito de alinhamento sob cada objeto daquele tipo: alinhamentos mais estritos podem ser requisitados utilizando a palavra-chave `_Alignas`.

Muitos programadores C que não possuem experiência com desenvolvimento para sistemas embarcados não se atentam a essa questão, pois o alinhamento necessário é definido pela implementação, e a maioria dos compiladores para a arquitetura x86, amplamente utilizada em computadores comuns, não define requisitos estritos de alinhamento, devido à flexibilidade que a arquitetura dá para que acessos não-alinhados sejam realizados.

```

==25521== Invalid write of size 1
==25521==    at 0x4831F94: memcpy (mc_replace_strmem.c:883)
==25521==    by 0x5260E7D: rb_read (serialize.c:151)
==25521==    by 0x5260F39: _get_buffer (serialize.c:471)
==25521==    by 0x52624DB: _push_value (serialize.c:538)
==25521==    by 0x526282D: _unpack_one (serialize.c:556)
==25521==    by 0x5262517: _push_value (serialize.c:499)
==25521==    by 0x526274B: _deserialize (serialize.c:674)
==25521==    by 0x447BF: lj_BC_FUNCC (lj_vm.s:917)
==25521==    by 0x39E87: lua_pcall (lj_api.c:1041)
==25521==    by 0xB509: docall (luajit.c:121)
==25521==    by 0xBECB: pmain (luajit.c:288)
==25521==    by 0x447BF: lj_BC_FUNCC (lj_vm.s:917)
==25521== Address 0xbdc347df is on thread 1's stack

```

Figura 16 – Trecho da saída do Valgrind ao monitorar-se a execução dos testes automatizados no dispositivo Kindle, mostrando a ocorrência de um acesso inválido à memória que não ocorria em ambiente simulado.

4.4 Identificação de acessos inválidos à memória

Habilitando-se o uso do Valgrind, por meio da opção `-use-valgrind` suportada pelo `run-tests.sh`, é possível monitorar acessos inválidos à memória, ou seja, acessos a endereços fora de áreas que tenham sido alocadas pelo programa.

Foi identificado um caso (Figura 16) bastante interessante de acesso inválido, que não ocorria quando os mesmos testes eram executados em ambiente simulado, no computador. A causa pode ser identificada em torno da linha 471 (Figura 17), identificada no *backtrace* do Valgrind (desta vez, a biblioteca lua-serialize havia sido compilada com símbolos de depuração).

Observando o código fonte, é possível inferir-se a causa: a vetor `tmp` é alocado na pilha, devido à forma como é declarado. A pilha tem tamanho variável dependendo do ambiente no qual o código é executado, e é compreensível que um sistema embarcado, que dispõe de uma quantidade reduzida de memória, adote um tamanho menor como padrão para a pilha.

```

469: static void
470: _get_buffer(lua_State *L, struct read_block *rb, int len) {
471:     char tmp[len];
472:     char * p = rb_read(rb,tmp,len);
473:     lua_pushlstring(L,p,len);
474: }

```

Figura 17 – Arquivo de código fonte serialize.c em torno da linha 471.

4.5 Solução dos problemas

O problema do acesso não-alinhado à memória foi resolvido alterando apenas a função interna `rb_read` da biblioteca `lua-serialize`. A função foi modificada para retornar sempre um ponteiro que já era fornecido como argumento para a função. Em todos os pontos onde a função é chamada, esse ponteiro faz referência a uma variável do tipo correto alocada na pilha. Portanto, o ponteiro é naturalmente alinhado. O código foi alterado para fazer uma cópia dos dados (usando a função `memcpy` da biblioteca padrão) para esse ponteiro sempre que necessário.

A questão do acesso inválido à memória foi resolvida substituindo a alocação na pilha por uma alocação com `malloc`.

Ambas alterações foram enviadas como contribuição (MATIAS, 2014) ao repositório do `lua-serialize`. A Tabela 2 apresenta o resultado dos testes automatizados antes e depois da nova versão da biblioteca ser incorporada ao KOREader.

Tabela 2 - Número de testes bem sucedidos e mal sucedidos quando executados diretamente no Kindle, antes e depois das contribuições enviadas ao projeto.

Identificador do <i>commit</i>	<i>Pull Request</i>	Testes bem sucedidos	Testes mal sucedidos
68e4a5eafd633494e14da8c229599fc6212fa3fa	#667	50	2
8e5a856841d2aa7de81d4c5ab3aa5de77141d5df	#668	52	0

5 AVALIAÇÃO DE DESEMPENHO

O próprio *framework* Busted é capaz de medir os tempos de execução de cada conjunto de testes executado. Esse dado é exibido juntamente com o número de testes bem sucedidos. Com a finalidade de realizar uma estatística simples, coletamos esse valor para várias rodadas de testes. Toda a avaliação foi realizada entre os dias 25 e 30 de junho de 2014, com a última versão disponível à época do repositório do KORreader.

Tabela 3 - Configurações do computador utilizado nos testes de desempenho.

Modelo	Notebook ASUS A43E-VX049R
Processador	Intel(R) Core(TM) i5-2410M CPU, capaz de operar às frequências de 2,30 GHz, 1,80 GHz, 1,60 GHz, 1,40 GHz, 1,20 GHz, 1000 MHz ou 800 MHz
Memória	SODIMM DDR3 Síncrono 1333 MHz (0,8 ns) (pente de 4 GiB + pente de 2 GiB, totalizando 6 GiB)
Disco	SAMSUNG HM500JI 500 GB
Rede sem fio	Qualcomm Atheros AR9285 Wireless Network Adapter (PCI-Express)

Para cada suíte de testes, realizamos 50 medidas de tempo. No computador, cujas características e configurações são apresentadas na Tabela 3, foi utilizado o comando `cpufreq-set` para travar a frequência do processador em 800 MHz, de forma a garantir uma maior estabilidade das medidas. Alguns dos testes automatizados envolvem comunicação com a rede e, por esse motivo, o Kindle foi mantido conectado tanto à rede Wi-Fi (ligada a um roteador de Internet) quanto à interface de rede USB (utilizada para monitorar o dispositivo via SSH).

Alguns testes de *frontend* envolvem o uso de um *cache* em disco. Portanto, são realizados dois tipos de teste: com *cache* frio e com *cache* quente. Nos testes com *cache* frio, o diretório de *cache* em disco é removido após cada rodada. Nos testes com *cache* quente, esse diretório é mantido intacto entre as rodadas.

No Anexo 2 – Tempos de execução, apresentamos todas as medidas realizadas. Na Tabela 4, calculamos a média e o desvio padrão dessas medidas, permitindo avaliar a diferença de desempenho entre o computador hospedeiro e o Kindle na execução de testes equivalentes.

Tabela 4 - Resumo dos tempos de execução dos testes em um Kindle e em um computador.

Suíte de testes	Cache	Dispositivo	Histograma	Tempo médio (s)	Desvio padrão (s)
<i>Backend</i>	Frio	Kindle	Figura 18	92,96	0,24
<i>Frontend</i>	Frio	Kindle	Figura 19	20,46	0,05
<i>Frontend</i>	Quente	Kindle	Figura 20	5,77	0,07
<i>Backend</i>	Frio	Computador (PC)	Figura 21	20,65	0,68
<i>Frontend</i>	Frio	Computador (PC)	Figura 22	3,00	0,01
<i>Frontend</i>	Quente	Computador (PC)	Figura 23	1,04	0,01

De acordo com a Tabela 4, o computador foi 4,5 vezes mais rápido que o Kindle nos testes de *backend*, 6,8 vezes nos testes de *frontend* com cache frio e 5,5 vezes nos testes de *frontend* com cache quente. Apesar de o processador do computador estar com frequência reduzida para 800MHz, o Kindle parece ter um desempenho razoável para um dispositivo de tamanha eficiência energética. A redução na velocidade dos testes é aceitável para um processo de integração contínua, pois o gargalo continua sendo a compilação, que demora dezenas de minutos.

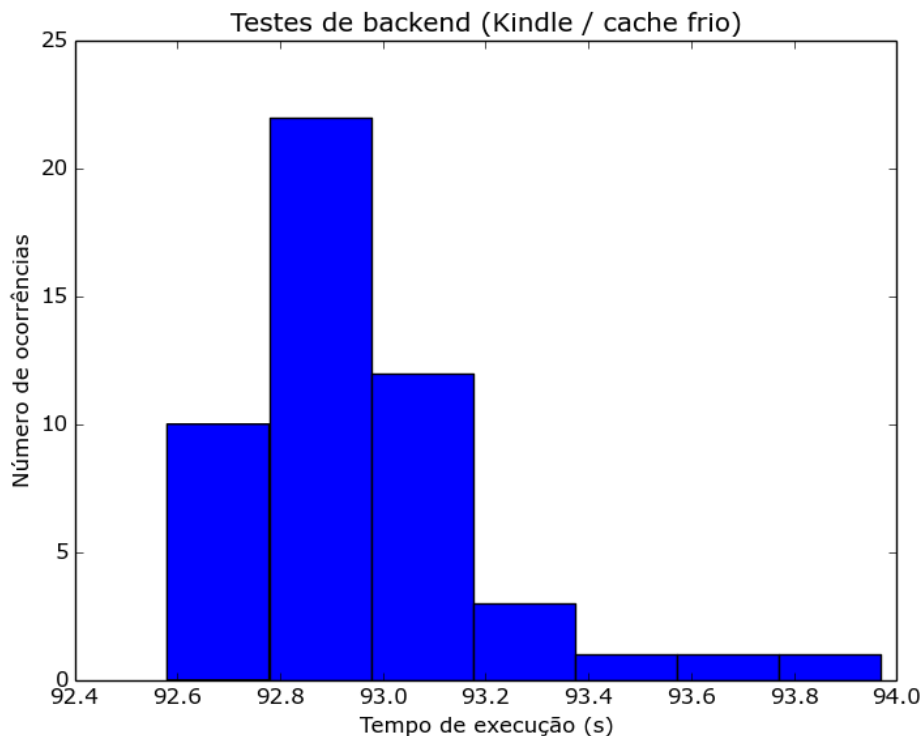


Figura 18 – Histograma de tempos de execução para os testes de *backend* no Kindle. O tempo médio foi de 92,96 segundos, cerca de 4,5 vezes mais lento que no computador. Entretanto, os valores de tempo ficaram menos espalhados que nos testes em computador, gerando um desvio padrão menor e indicando um maior determinismo do Kindle na execução destes testes.

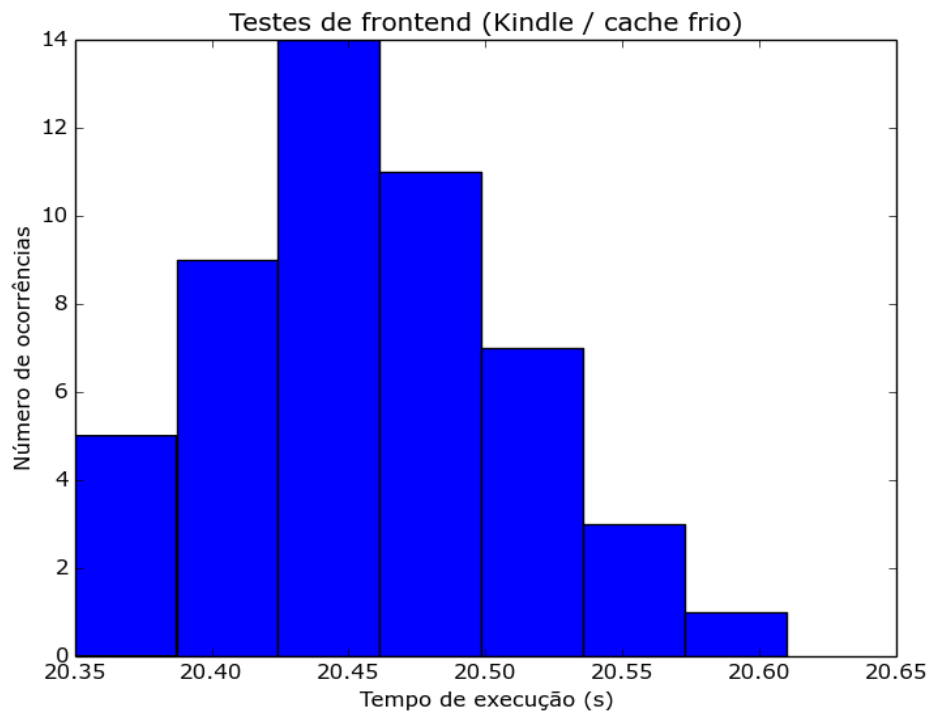


Figura 19 – Histograma de tempos de execução para os testes de *frontend* no Kindle, com *cache* frio. O tempo médio foi de 20,46 segundos, cerca de 6,8 mais lento que no computador.

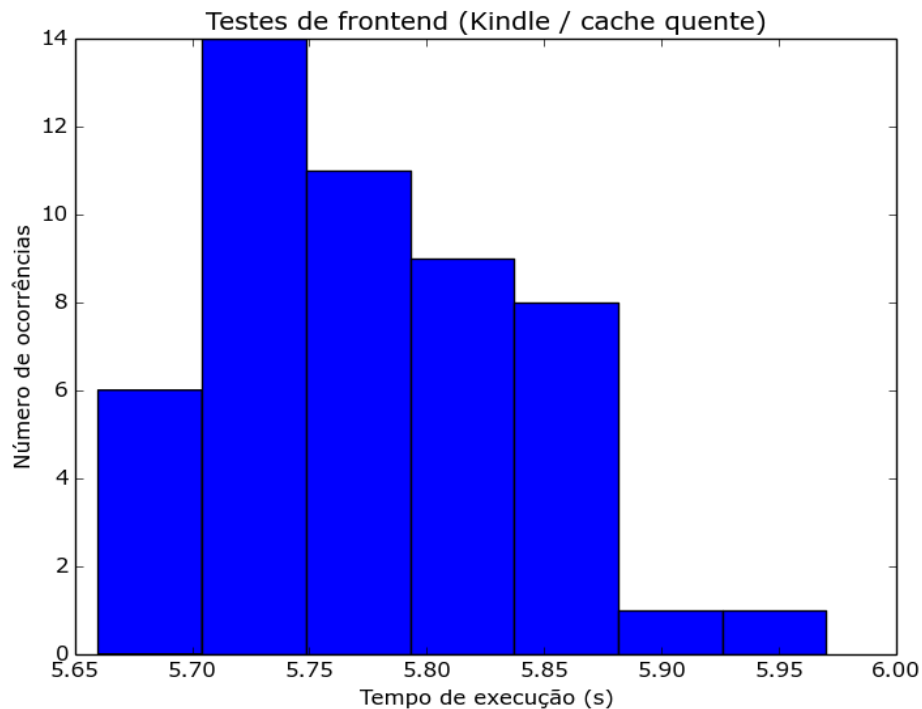


Figura 20 – Histograma de tempos de execução para os testes de *frontend* no Kindle, com *cache* quente. O tempo médio foi de 5,77 segundos, cerca de 5,5 mais lento que no computador. A redução de desempenho no Kindle (com relação ao computador) é, portanto, menor em testes que envolvem aproveitamento do *cache*, provavelmente devido à eficiência razoável das leituras realizadas a partir da memória Flash.

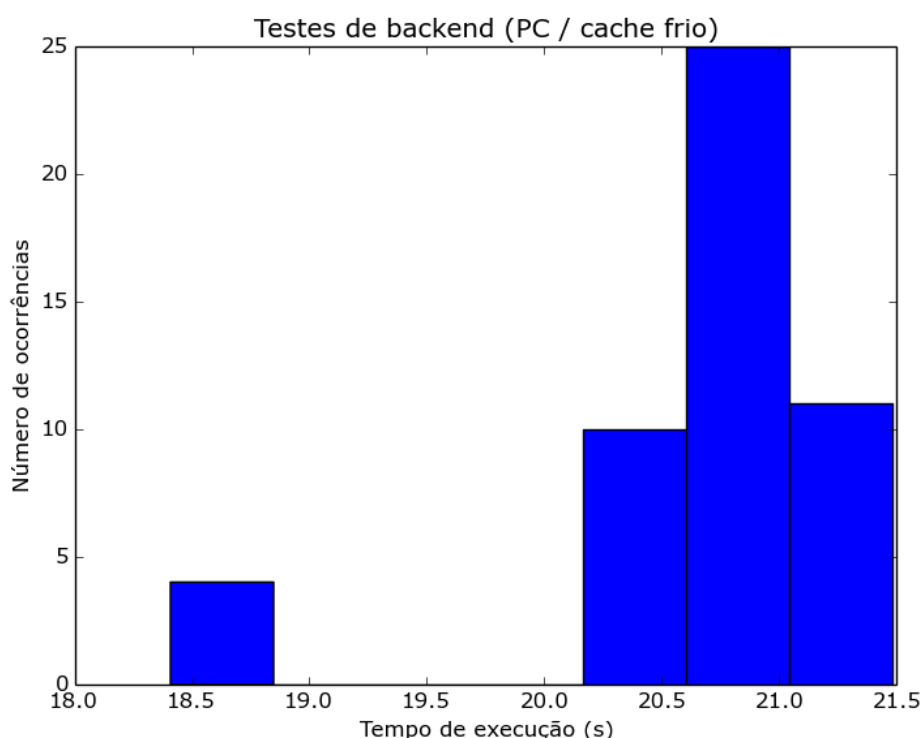


Figura 21 – Histograma de tempos de execução para os testes de *backend* no computador hospedeiro. O tempo médio foi de 20,65 segundos. Dentre os 50 tempos, 4 foram muito menores que os outros, valendo em torno de 18,5 segundos. Esse comportamento provavelmente ocorre devido aos testes de rede que fazem parte de suíte de *backend*.

Um dado interessante, apesar de não relacionado diretamente aos objetivos deste trabalho, é que o desvio padrão do tempo de execução dos testes de *backend* no Kindle é inferior ao no computador. De fato, observando a Figura 21 nota-se que os tempos de execução no computador estão espalhados ao redor de dois valores médios distintos. Uma possível explicação para esse fato é a existência de testes envolvendo rotinas de rede dentro da suíte de testes de *backend*. Tanto o Kindle quanto o computador estavam conectados a um roteador por meio de rede sem fio (Wi-Fi). Esse resultado indicaria que o Kindle responde de forma mais determinística que o computador a requisições de rede, o que seria natural, visto que um dispositivo embarcado geralmente possui menos tarefas ocupando o sistema operacional e causando uma dispersão dos tempos de resposta a chamadas de sistema. Ainda assim, o histograma da Figura 18, que corresponde ao mesmo conjunto de testes executado no Kindle, parece ter uma cauda prolongada para a direita, apesar de suas quatro primeiras barras formarem uma silhueta bastante característica de uma distribuição gaussiana, indicando que os efeitos causados pelos testes de rede também devem estar presentes no Kindle, apenas em menor proporção.

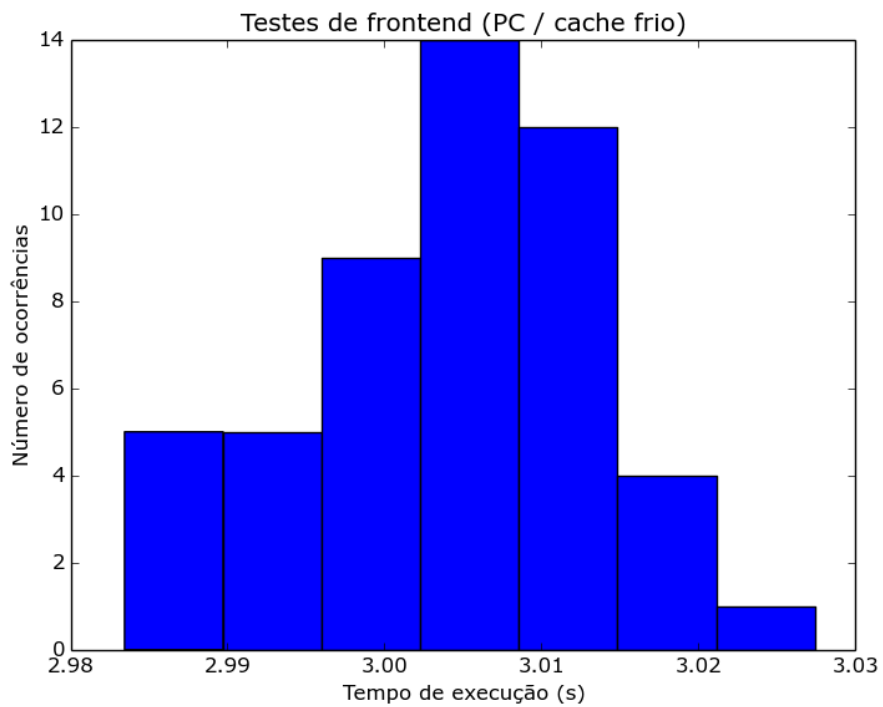


Figura 22 – Histograma de tempos de execução para os testes de *frontend* no computador hospedeiro, com *cache* frio. O tempo médio foi de 3,00 segundos. Não são observadas características especialmente peculiares na distribuição de tempos.

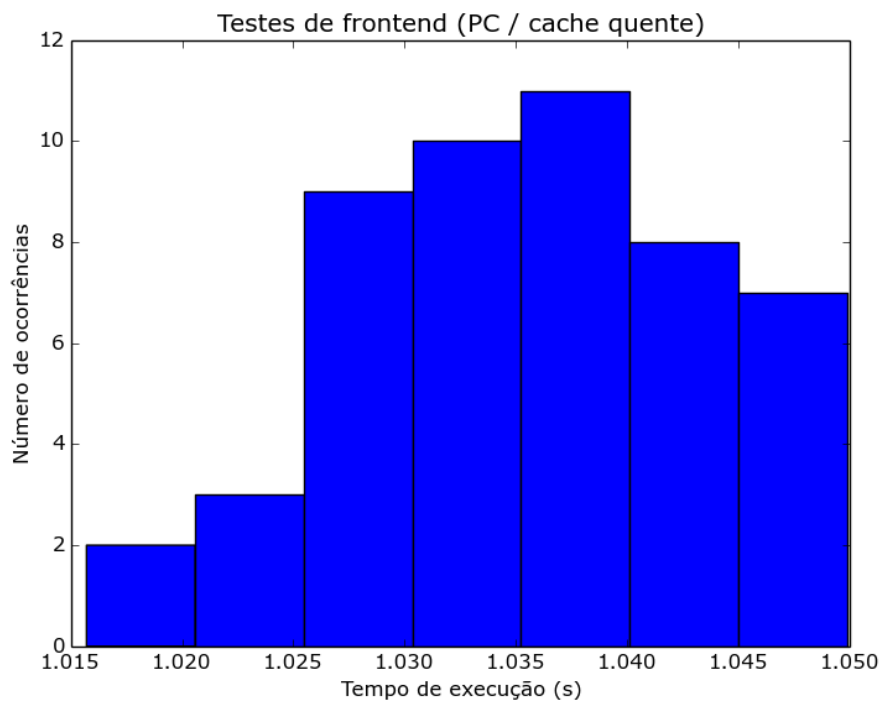


Figura 23 – Histograma de tempos de execução para os testes de *frontend* no computador hospedeiro, com *cache* quente. O tempo médio foi de 1,04 segundos. Não são observadas características especialmente peculiares na distribuição de tempos.

Os demais histogramas, apresentados nas figuras 19, 20, 22 e 23, não apresentam características visuais especialmente peculiares, pelo que consideramos que os dados de valor médio e desvio padrão da Tabela 4 são capazes, por si só, de representar de forma bastante completa os resultados de tais experimentos.

6 CONCLUSÕES

Este trabalho descreveu uma solução completa e funcional para implementar, em um projeto de software livre já existente e estabelecido (KORreader), um processo de integração contínua no qual os testes automatizados são executados no próprio dispositivo.

Nossos experimentos demonstraram que, em um caso prático, alguns tipos de erros de programação escapam aos testes automatizados realizados de forma simulada em um computador, concordando com Grenning (2007) e outros autores. Um dos casos encontrados pode ser categorizado como ocorrendo devido a diferenças entre arquiteturas de processador, e o outro devido a configurações distintas do ambiente.

Mostramos que os resultados dos testes automatizados efetuados no dispositivo foram úteis, permitindo enviar contribuições ao projeto. Defendemos também a viabilidade da adoção do processo, pois neste caso o aumento de tempo de execução de 4,5 a 6,8 vezes com relação aos testes realizados em ambiente simulado é plenamente tolerável, por não ser o gargalo do processo de integração contínua, uma vez que toda a suíte de testes pode ser concluída em menos de dois minutos, enquanto a compilação demora, de qualquer forma, dezenas de minutos.

Uma limitação do método proposto é não ser capaz de lidar com testes em código de origem não confiável, característica interessante de soluções como o Travis CI, que permitem aos desenvolvedores verificar o resultado dos testes automatizados ao receber uma contribuição de terceiros desconhecidos, antes mesmo de ler o código enviado pelo contribuidor. Entretanto, é um grande desafio implementar em um sistema embarcado os mecanismos de isolamento necessários a esse tipo de recurso. Trabalhos envolvendo essa questão seriam de grande interesse científico e tecnológico.

REFERÊNCIAS

ABRAHAMSEN, J. et al. **Turbo**: a framework for LuaJIT 2 to simplify the task of building fast and scalable network applications. GitHub, 2014. Disponível em: <<https://github.com/kernelsauce/turbo>>. Acesso em: 29 jun. 2014.

ARM. **ARM architecture reference manual**: ARMv7-A and ARMv7-R edition. ARM Holdings, 2014. Disponível em: <<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html>>. Acesso em: 29 jun. 2014.

BIGUET, D. **USBNetwork package for Kindle Touch/Paperwhite**. MobileRead Forums, 2014. Disponível em: <<http://www.mobileread.com/forums/showthread.php?t=186645>>. Acesso em: 29 jun. 2014.

BOTTOU, L. et al. High quality document image compression with “DjVu”. **Journal of Electronic Imaging**, v. 7, n. 3, p. 410–425, 1998.

BROEKMAN, B.; NOTENBOOM, E. **Testing embedded software**. Pearson Education, 2003.

CHEN, Y. et al. Electronic paper: Flexible active-matrix electronic ink display. **Nature**, v. 423, n. 6936, p. 136–136, 2003.

COMISKEY, B. et al. An electrophoretic ink for all-printed reflective electronic displays. **Nature**, v. 394, n. 6690, p. 253–255, 1998.

CORDEIRO, D. de A. **Estudo de escalabilidade de servidores baseados em eventos em sistemas multiprocessados**: um estudo de caso completo. 2006. 103 f. Dissertação (Mestrado) – Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2006.

FOWLER, M.; FOEMMEL, M. **Continuous integration**. Thought-Works, 2006. Disponível em: <<http://martinfowler.com/articles/continuousIntegration.html>>. Acesso em: 29 jun. 2014.

FREESCALE. i.MX50 Applications Processor for Consumer Products. Freescale Semiconductor, 2013. Disponível em: <http://cache.freescale.com/files/32bit/doc/data_sheet/IMX50CEC.pdf>. Acesso em: 29 jun. 2014.

FUCHS, S. et al. **Travis CI**: free hosted continuous integration platform for the open source community. Travis CI GmbH, 2011. Disponível em: <<https://travis-ci.org>>. Acesso em: 29 jun. 2014.

GATLIFF, B. Embedding with GNU: GNU debugger. **Embedded Systems Programming**, v. 12, p. 80–95, 1999.

GRENNING, J. Applying test driven development to embedded software. **IEEE Instrumentation & Measurement Magazine**, v. 10, n. 6, p. 20–25, 2007.

HENKEL, J. Selective revealing in open innovation processes: the case of embedded Linux. **Research policy**, v. 35, n. 7, p. 953–969, 2006.

HOLCK, J.; JØRGENSEN, N. Continuous integration and quality assurance: a case study of two open source projects. **Australasian Journal of Information Systems**, v. 11, n. 1, p. 40–53, 2007.

HOSKINS, M. E. SSHFS: super easy file access over SSH. **Linux Journal**, v. 2006, n. 146, p. 4, 2006.

HOU Q. et al. **KOReader**: a document viewer application. GitHub, 2014. Disponível em: <<https://github.com/koreader/koreader>>. Acesso em: 29 jun. 2014.

IERUSALIMSKY, R. et al. Lua – an extensible extension language. **Software: Practice & Experience**, v. 26, p. 635–652, 1995.

IERUSALIMSKY, R. **Programming in Lua**. 3 ed. Rio de Janeiro: Lua.org, 2013.

ISO/IEC 9899:201x. Committee Draft, 12 abr. 2011. Disponível em: <<http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>>. Acesso em: 29 jun. 2014.

KARLESKY, M. et al. Mocking the embedded world: test-driven development, continuous integration, and design patterns. In: **Proc. Emb. Systems Conf**, CA, USA. 2007.

LANUBILE, F. Collaboration in distributed software development. In: DE LUCIA, A.; FERRUCI, F. (Ed.). **Software Engineering**. Springer Berlin Heidelberg, 2009. p. 174–193.

LAWSON, J. **Busted**: elegant Lua unit testing. GitHub, 2014. Disponível em: <<https://github.com/Olivine-Labs/busted>>. Acesso em: 29 jun. 2014.

LIBENZI, D. **Improving network I/O performance**. 2002. Disponível em: <<http://www.xmailserver.org/linux-patches/nio-improve.html>>. Acesso em: 29 jun. 2014.

MATIAS, M. S. de O. **General fix for alignment issues on ARM**. 2014. Disponível em: <<https://github.com/chrox/lua-serialize/pull/1/files>>. Acesso em: 29 jun. 2014.

NETHERCOTE, N.; SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: **ACM Sigplan Notices**. ACM, 2007. p. 89–100.

NEWHAM, C. **Learning the bash shell**: Unix shell programming. 3 ed. O'Reilly Media Inc., 2005.

PALL, M. **The LuaJIT project**. 2007. Disponível em: <<http://luajit.org>>. Acesso em: 29 jun. 2014.

PALL, M. **Performance: ARM**. 2014. Disponível em: <http://luajit.org/performance_arm.html>. Acesso em: 9 jul. 2014.

WILLUS. **K2pdfopt**. 2011. Disponível em: <<http://www.willus.com/k2pdfopt>>. Acesso em: 29 jun. 2014.

YLONEN, T. SSH – secure login connections over the Internet. In: **Proceedings of the 6th USENIX Security Symposium**. 1996. p. 37–42.

ANEXOS

Anexo 1 – Código fonte

ciserver.lua

```
#!/usr/local/bin/luajit
local server_port = 8888
local api_secret = "secret"
local ci_script = "./dispatch.sh"
local log_dir = "./log/"

local ffi = require "ffi"
ffi.cdef[[
struct _IO_FILE;
typedef struct _IO_FILE FILE;

FILE *popen(const char *command, const char *type);
int fileno(FILE *stream);
int pclose(FILE *stream);
]]

local band, bor, rsh = bit.band, bit.bor, bit.rshift
local C = ffi.C

TURBO_SSL = true
local turbo = require "turbo"
local ioloop = turbo.ioloop.instance()

local SubProcess = class("SubProcess")

function SubProcess:initialize(cmd, bufsize)
    bufsize = bufsize or 16*1024
    self.buf = ffi.new('char[?]', bufsize)
    self.pipe = C.popen(cmd, "r")
    self.pipefd = C.fileno(self.pipe)
    turbo.socket.set_nonblock_flag(self.pipefd)
    ioloop:add_handler(self.pipefd, bor(turbo.ioloop.READ,
        turbo.ioloop.ERROR), self._eventHandler, self)
end

function SubProcess:_eventHandler(fd, events)
    if band(events, turbo.ioloop.READ) ~= 0 then
        local n = C.read(fd, self.buf, ffi.sizeof(self.buf))
        local data = ffi.string(self.buf, n)
    end
end
```

```

        self:recv(data)
    end
    if band(events, turbo.ioloop.ERROR) ~= 0 then
        ioloop:remove_handler(fd)
        local status = C.pclose(self.pipe)
        local termsig = band(status, 0x7f)
        local exitcode = band(rsh(status, 8), 0xff)
        self:exit(termsig, exitcode)
    end
end

local CIPProcess = class("CIPProcess", SubProcess)

function CIPProcess:initialize(taskqueue, task)
    self.taskqueue = taskqueue
    self.task = task
    self.logfile = io.open(log_dir..task.commit, "w")
    self.logfile:setvbuf("line")
    local cmd = ci_script.." "..task.commit.." 2>&1"
    SubProcess.initialize(self, cmd)
end

function CIPProcess:recv(data)
    self.logfile:write(data)
end

function CIPProcess:exit(termsig, exitcode)
    self.logfile:write(string.format(
        "\n\n** Terminated with signal=%d, exitcode=%d\n",
        termsig, exitcode)
    )
    self.logfile:close()

    local indexfile = io.open(log_dir.."index", "a")
    local commit = self.task.commit
    local status = (termsig == 0 and exitcode == 0) and "OK" or "FAIL"
    indexfile:write(
        string.format('<a href="%s">%s</a> %s<br />\n',
            commit, commit, status)
    )
    indexfile:close()

    self.taskqueue.curtask = nil
    self.taskqueue:dispatch()
end

local TaskQueue = class("TaskQueue", turbo.structs.deque)

function TaskQueue:enqueue(task)
    self:appendleft(task)
end

```

```

        if not self.curtask then
            self:dispatch()
        end
    end
end

function TaskQueue:dispatch()
    local task = self:pop()
    if not task then return end
    self.curtask = CIPProcess:new(self, task)
end

local taskqueue = TaskQueue:new()

local GitHubHandler = class("GitHubHandler", turbo.web.RequestHandler)

function GitHubHandler:authcheck()
    local signature = self.request.headers:get("X-Hub-Signature")
    if signature ~= "sha1="..turbo.hash.HMAC(api_secret,
                                                self.request.body) then
        error(turbo.web.HTTPError(401, "Unauthorized"))
    end
end

function GitHubHandler:post()
    self:authcheck()
    local event = self.request.headers:get("X-GitHub-Event")
    local body = turbo.escape.json_decode(self.request.body)
    if event == "push" then
        self:pushevent(body)
    end
end

function validate_commit(commit)
    if commit:match("[^0-9a-f]") then
        error(turbo.web.HTTPError(403,
            "Invalid commit identifier"))
    end
end

function GitHubHandler:pushevent(body)
    local commit = body.after
    validate_commit(commit)
    taskqueue:enqueue({commit = commit})
end

local LogHandler = class("LogHandler", turbo.web.RequestHandler)

function LogHandler:get(commit)
    if not commit or commit == "" or commit == "index" then
        local file = io.open(log_dir.."index", "r")
    end
end

```



```

self:write('<div style="font-family: monospace">\n')
if file then
    self:write(file:read("*a"))
    file:close()
end
if taskqueue.curtask then
    local commit = taskqueue.curtask.task.commit
    self:write(string.format(
        '<i><a href="%s">%s</a> in progress</i><br />\n',
        commit, commit)
    )
end
self:write('</div>')
else
    validate_commit(commit)
    local file = io.open(log_dir..commit, "r")
    if not file then
        error(turbo.web.HTTPError(404, "Not found"))
    end
    self:add_header("Content-Type",
        "text/plain; charset=UTF-8; imeanit=yes")
    self:add_header("X-Content-Type-Options", "nosniff")
    self:write(file:read("*a"))
    file:close()
end
end

local application = turbo.web.Application:new({
    {"^/push_hook$", GitHubHandler},
    {"^/log/(.*)$", LogHandler},
})

application:listen(server_port, nil, {
    ssl_options = {
        key_file = "./sslkeys/server.key",
        cert_file = "./sslkeys/server.crt"
    }
})

ioloop:start()

```

ssl-genkeys.sh

```

#!/bin/bash
set -xe
mkdir -p sslkeys log
cd sslkeys
openssl genrsa -des3 -out server.enc.key -passout pass:qwerty123 1024

```

```
openssl rsa -in server.enc.key -out server.key -passin pass:qwerty123
openssl req -new -key server.key -out server.csr -subj '/'
openssl x509 -req -days 3650 -in server.csr -signkey server.key \
    -out server.crt
```

config.env

```
CROSS_PATH=/opt/arm-kind5-linux-gnueabi/bin
CROSS_PREFIX=arm-kind5-linux-gnueabi
MAKEFLAGS=-j4
DEV_MOUNTPOINT=/mnt/us/kindleci
```

common.sh

```
fetch() {
    local destfile=downloads/"$1"
    while [[ ! -f "$destfile" || `md5sum "$destfile"|cut -d\ -f1` != \
        "$2" ]]; do
        rm "$destfile" 2>/dev/null
        wget -O "$destfile" "$3" || return $?
    done
}
```

dispatch.sh

```
#!/bin/bash
set -xe
./update-koreader.sh "$1"
./make-koreader.sh debug
./run-tests.sh
```

update-koreader.sh

```
#!/bin/bash
# Fetch or update koreader
set -e
pushd downloads
if [[ ! -d koreader ]]; then
    git clone https://github.com/koreader/koreader.git
    cd koreader
else
```

```

    cd koreader
    git checkout -f master
    git pull
fi
if [[ ! -z "$1" ]]; then
    git checkout -f "$1"
fi
make fetchthirdparty
popd

```

make-koreader.sh

```

#!/bin/bash
. config.env
. common.sh

export PATH=$PATH:$CROSS_PATH

BUILD_DIR="$(readlink -f build)"
ARM_ARCH="-march=armv7-a -mtune=cortex-a8 -mcpu=neon -mfloat-abi=softfp \
-mthumb"

if [[ "$1" == "debug" ]]; then
    STRIP="echo"           # do not strip executables
    CCDEBUG="-O0 -g"       # for LuaJIT
    BASE_CFLAGS="-O2 -g"   # for everything else
    # Extra CFLAGS for LuaJIT
    XCFLAGS="-DLUAJIT_USE_GDBJIT -DLUA_USE_APICHECK -DLUA_USE_ASSERT"
    XCFLAGS="-DLUAJIT_USE_VALGRIND -DLUAJIT_USE_SYSMALLOC"
    XCFLAGS="$XCFLAGS -I${BUILD_DIR}/valgrind/include"
elif [[ "$1" == "release" ]]; then
    STRIP="${CROSS_PREFIX}-strip"
    BASE_CFLAGS="-O2 -fomit-frame-pointer -frename-registers -fweb -pipe"
    CCDEBUG=""
    XCFLAGS=""
else
    echo "usage: $0 debug|release"
    exit 1
fi

rm -rf work/koreader 2>/dev/null
cp -r downloads/koreader work

pushd work/koreader
unset MAKEFLAGS
make TARGET=kindle CHOST="${CROSS_PREFIX}" \
    BASE_CFLAGS="${BASE_CFLAGS}" \
    ARM_BACKWARD_COMPAT_CFLAGS="" \
    ARM_BACKWARD_COMPAT_CXXFLAGS="" \

```

```

    ARM_ARCH="${ARM_ARCH}" \
    STRIP="${STRIP}" \
    CCDEBUG="${CCDEBUG}" XCFLAGS="${XCFLAGS}" \
    kindleupdate || exit $?
popd

rm -rf build/koreader
pushd build
unzip ../work/koreader/koreader-*.zip 'koreader/*' || exit $?
cd koreader

# Copy toolchain's libstdc++
find "$CROSS_PATH/.." -name libstdc++.so.6 -exec cp '{}' libs \;

# Copy tests
mkdir -p spec/front
cp -r ../../work/koreader/spec/unit spec/front
cp -r ../../work/koreader/test spec/front/data
ln -sf ../data spec/front/unit/data # link otherwise busted crashes
mkdir -p spec/base
cp -r ../../work/koreader/koreader-base/spec/unit spec/base
mv spec/base/unit/data spec/base
ln -sf ../data spec/base/unit/data # link otherwise busted crashes
popd

# Extract Tesseract English data
# (needed for spec/base/unit tests)
tessver=3.01
fetch tesseract-ocr-${tessver}.eng.tar.gz \
89c139a73e0e7b1225809fc7b226b6c9 https://tesseract-\
ocr.googlecode.com/files/tesseract-ocr-${tessver}.eng.tar.gz
tar --strip-components=1 -zxvf downloads/tesseract-ocr-\
${tessver}.eng.tar.gz -C build/koreader/data

```

run-tests.sh

```

#!/bin/bash
. config.env

# Run compatibility quirks if they exist
if [[ -x ./quirks.sh ]]; then ./quirks.sh || exit $?; fi

USE_VALGRIND=0
if [[ $# > 0 ]]; then
    if [[ $1 == "--use-valgrind" ]]; then
        USE_VALGRIND=1
        shift
        VALGRIND_ARGS="$@"
    else

```

```

        echo "usage: $0 [--use-valgrind [valgrind_args]]"
        exit 1
    fi
fi

VALGRIND_CMD=""
if [[ $USE_VALGRIND == 1 ]]; then
    VALGRIND_CMD="$DEV_MOUNTPOINT/valgrind/bin/valgrind \
$VALGRIND_ARGS \
--extra-debuginfo-path=$DEV_MOUNTPOINT/valgrind/debug \
--suppressions=$DEV_MOUNTPOINT/valgrind/lib/valgrind/lj.supp "
fi

BUSTED_SCRIPT="$DEV_MOUNTPOINT/luajit/bin/busted_bootstrap"

cat > build/run-tests.sh << EOF
#!/bin/sh
export LUA_PATH='./?.lua;$DEV_MOUNTPOINT/luajit/share/lua/5.1/?.lua;\
$DEV_MOUNTPOINT/luajit/share/lua/5.1/?/init.lua'
export LUA_CPATH='./?.so;$DEV_MOUNTPOINT/luajit/lib/lua/5.1/?.so;\
$DEV_MOUNTPOINT/luajit/lib/lua/5.1/loadall.so'
export TESSDATA_PREFIX="$DEV_MOUNTPOINT/koreader/data"
cd "$DEV_MOUNTPOINT/koreader"
set -xe
${VALGRIND_CMD} ./luajit $BUSTED_SCRIPT spec/base/unit
${VALGRIND_CMD} ./luajit $BUSTED_SCRIPT spec/front/unit
EOF
chmod +x build/run-tests.sh

if ! ssh kindle [ -x "$DEV_MOUNTPOINT/run-tests.sh" ]; then
    BUILD_PATH="$(readlink -f build)"
    ssh kindle mkdir -p "$DEV_MOUNTPOINT" || exit $?
    ssh kindle sshfs "master:$BUILD_PATH" "$DEV_MOUNTPOINT" || exit $?
fi

# Needed for zeromq tests
ssh kindle /usr/sbin/iptables -A INPUT -i lo -j ACCEPT

ssh kindle "$DEV_MOUNTPOINT/run-tests.sh"
STATUS=$?

ssh kindle /usr/sbin/iptables -D INPUT -i lo -j ACCEPT

exit $STATUS

```

quirks.sh

```

#!/bin/bash
for f in build/koreader/spec/front/unit/*.lua; do

```

```

    sed -i 's,sample\.pdf,2col.pdf,g' $f || exit $?
done

```

make-infra.sh

```

#!/bin/bash
. config.env
. common.sh

export MAKEFLAGS
export PATH=$PATH:$CROSS_PATH

mkdir -p downloads work build
BUILD_DIR="$(readlink -f build)"

move_up_mountpoint() {
    pushd $BUILD_DIR
    local dir=".$DEV_MOUNTPOINT"
    mv "$dir"/* .
    while [[ ! -z "$dir" && "$dir" != "." ]]; do
        rmdir "$dir"
        dir="$(dirname "$dir")"
    done
    popd
}

build_valgrind() {
    [[ -f work/.done.valgrind ]] && return 0
    local ver=3.9.0
    fetch valgrind-${ver}.tar.bz2 0947de8112f946b9ce64764af7be6df2 \
http://valgrind.org/downloads/valgrind-${ver}.tar.bz2
    rm -rf work/valgrind-${ver} 2>/dev/null
    tar -C work -jxf downloads/valgrind-${ver}.tar.bz2 || return $?
    pushd work/valgrind-${ver}
    CC=${CROSS_PREFIX}-gcc \
    CPP=${CROSS_PREFIX}-cpp \
    CXX=${CROSS_PREFIX}-g++ \
    LD=${CROSS_PREFIX}-ld \
    AR=${CROSS_PREFIX}-ar \
    ./configure --target=${CROSS_PREFIX} \
        --host=$(echo $CROSS_PREFIX | sed s,arm-,armv7-,) \
        --prefix="$DEV_MOUNTPOINT/valgrind" || return $?
    make || return $?
    make install DESTDIR="$BUILD_DIR" || return $?
    popd
    move_up_mountpoint
    touch work/.done.valgrind
}

```

```

install_libc6_dbg() {
    [[ -f work/.done.libc6dbg ]] && return 0
    local ver=2.12.1-0ubuntu6
    fetch libc6-dbg_${ver}_armel.deb aa6bb85226e6154ea6b30c1a3b8f9adc \
http://launchpadlibrarian.net/55372239/libc6-dbg_${ver}_armel.deb
    mkdir -p work/libc6dbg
    pushd work/libc6dbg
    ar x ../../downloads/libc6-dbg_${ver}_armel.deb || return $?
    tar -zxf data.tar.gz || return $?
    cp -r usr/lib/debug "$BUILD_DIR/valgrind" || return $?
    popd
    touch work/.done.libc6dbg
}

build_luajit() {
    [[ -f work/.done.luajit ]] && return 0
    local ljdir="lua-jit-2.0"
    pushd downloads
    if [[ ! -d "$ljdir" ]]; then
        git clone http://lua-jit.org/git/lua-jit-2.0.git "$ljdir" || \
return $?
        cd "$ljdir"
    else
        cd "$ljdir"
        git pull
    fi
    git checkout v2.1 || return $?
    rm -rf "../../work/$ljdir" 2>/dev/null
    git clone . "../../work/$ljdir" || return $?
    popd
    pushd "work/$ljdir"
    make HOST_CC="gcc -m32" CROSS="${CROSS_PREFIX}-" \
TARGET_CFLAGS="-march=armv7-a -mtune=cortex-a8 -mcpu=neon -marm" \
PREFIX="$DEV_MOUNTPOINT/lua-jit" \
amalg || return $?
    make install PREFIX="$DEV_MOUNTPOINT/lua-jit" DESTDIR="$BUILD_DIR" \
|| return $?
    cp src/lj.supp "$BUILD_DIR/valgrind/lib/valgrind"
    popd
    move_up_mountpoint
    touch work/.done.luajit
}

build_busted() {
    [[ -f work/.done.busted ]] && return 0
    LUAROCKS_CONFIG="work/luarocks_conf.lua"
    cat > "$LUAROCKS_CONFIG" << EOF
rocks_trees = {
    [$BUILD_DIR/lua-jit]
}
variables = {
    CC = [${CROSS_PREFIX}-gcc],

```

```

CPP = [[${CROSS_PREFIX}-cpp]],
CXX = [[${CROSS_PREFIX}-g++]],
LD = [[${CROSS_PREFIX}-gcc]],
AR = [[${CROSS_PREFIX}-ar]],
LUA_INCDIR = [[${BUILD_DIR}/luajit/include/luajit-2.1]],
LUA_LIBDIR = [[${BUILD_DIR}/luajit/lib]],
LUA_BINDIR = [[${BUILD_DIR}/luajit/bin]],
LUAROCKS_UNAME_M= [[armv7l]],
CFLAGS = [[-march=armv7-a -mtune=cortex-a8 -mcpu=neon -marm
-shared -fPIC]]
}
EOF
    export LUAROCKS_CONFIG
    luarocks install busted || return $?
    touch work/.done.busted
}

build_valgrind || exit $?
install_libc6_dbg || exit $?
build_luajit || exit $?
build_busted || exit $?

```


Anexo 2 – Tempos de execução

Tempos no Kindle

<i>Testes de backend (tempos em segundos)</i>				
93.30000000	92.94000000	92.81000000	92.58000000	92.67000000
92.89000000	93.01000000	93.27000000	92.94000000	92.73000000
92.88000000	92.76000000	93.07000000	92.80000000	92.69000000
92.85000000	92.91000000	92.93000000	93.05000000	93.01000000
93.01000000	92.97000000	92.86000000	93.30000000	92.73000000
92.66000000	92.96000000	93.11000000	92.95000000	92.96000000
92.92000000	92.97000000	92.65000000	92.90000000	92.80000000
92.77000000	93.44000000	92.98000000	92.84000000	93.97000000
93.11000000	93.01000000	92.75000000	93.05000000	92.86000000
93.00000000	93.60000000	92.85000000	93.06000000	92.94000000

<i>Testes de frontend – cache frio (tempos em segundos)</i>				
20.61000000	20.53000000	20.46000000	20.47000000	20.46000000
20.48000000	20.40000000	20.43000000	20.48000000	20.37000000
20.43000000	20.35000000	20.46000000	20.41000000	20.55000000
20.50000000	20.43000000	20.44000000	20.40000000	20.49000000
20.43000000	20.44000000	20.46000000	20.48000000	20.51000000
20.43000000	20.42000000	20.40000000	20.39000000	20.45000000
20.37000000	20.51000000	20.40000000	20.37000000	20.54000000
20.53000000	20.40000000	20.56000000	20.47000000	20.37000000
20.47000000	20.50000000	20.41000000	20.51000000	20.44000000
20.49000000	20.47000000	20.44000000	20.47000000	20.48000000

<i>Testes de frontend – cache quente (tempos em segundos)</i>				
5.76000000	5.83000000	5.80000000	5.85000000	5.77000000
5.80000000	5.82000000	5.76000000	5.71000000	5.97000000
5.72000000	5.75000000	5.75000000	5.73000000	5.74000000
5.80000000	5.72000000	5.66000000	5.83000000	5.71000000
5.86000000	5.73000000	5.83000000	5.72000000	5.90000000
5.70000000	5.86000000	5.85000000	5.69000000	5.71000000
5.79000000	5.72000000	5.70000000	5.71000000	5.73000000
5.76000000	5.88000000	5.85000000	5.72000000	5.76000000
5.84000000	5.85000000	5.76000000	5.78000000	5.77000000
5.70000000	5.83000000	5.69000000	5.73000000	5.83000000

Tempos no computador

<i>Testes de backend (tempos em segundos)</i>				
20.53594100	21.12205000	21.48410900	20.98347900	18.51756100
20.56368900	21.18651800	20.56919800	20.59427300	20.77285000
21.15884300	21.29989500	20.99013500	20.98294300	20.66037900
21.08569500	20.53556900	20.66190500	20.94875300	20.83627200
18.41785900	20.54293800	20.84025000	20.62480200	20.60583800
20.93218200	21.29615700	21.07386300	20.70215600	20.56199200
21.26728300	20.84761400	20.39713400	21.02595300	20.68041300
20.63229100	20.79425900	18.62047100	21.12013300	20.58947900
21.05202200	20.65753500	20.92179400	20.74661300	20.89732000
20.87041500	20.37801500	20.71944800	20.69106700	18.40808700

<i>Testes de frontend – cache frio (tempos em segundos)</i>				
2.99070600	3.00386000	3.00388800	3.00190500	3.00383800
3.00786700	3.01671400	3.00094600	3.00936600	3.01710800
3.02741200	2.99933500	3.00061200	3.00501500	2.99933600
2.99549700	2.99684600	2.99740900	3.01146300	3.00087600
3.01828600	3.01246200	2.99218400	3.00662500	3.01371900
2.99969600	3.01407900	3.01175400	3.00665400	3.00985200
3.00584800	2.98857000	2.98789200	3.02103100	2.99085400
3.01020000	3.00853900	2.99048300	3.01263200	3.01193500
3.00422200	2.98401900	3.00421800	3.01169100	2.98667200
3.00496300	3.00618400	3.01374600	2.98343900	3.00356700

<i>Testes de frontend – cache quente (tempos em segundos)</i>				
1.02949200	1.03401200	1.03598200	1.03970100	1.03266400
1.04986700	1.03861600	1.03637800	1.03501400	1.03277800
1.03048500	1.02787300	1.03154400	1.03822900	1.01570900
1.04980300	1.04715500	1.04452800	1.03884900	1.04273900
1.03581800	1.02338700	1.04820600	1.03011500	1.03282000
1.02868300	1.04126200	1.03612400	1.02034900	1.02715600
1.02363900	1.04148400	1.04086000	1.02901600	1.03300700
1.04729200	1.03943900	1.04108200	1.02567400	1.03120300
1.03029300	1.04922200	1.02855100	1.04071300	1.04770900
1.02326200	1.04257100	1.03195400	1.03727400	1.03810900