Universidade
Estadual de Londrina

RAFAEL DE PAULA HERRERA

UMA ESTRATÉGIA PARA MIGRAÇÃO E IMPLANTAÇÃO DE ALTA DISPONIBILIDADE EM SISTEMAS DE INFORMAÇÕES LEGADOS

Londrina
2014

RAFAEL DE PAULA HERRERA

UMA ESTRATÉGIA PARA MIGRAÇÃO E IMPLANTAÇÃO DE ALTA DISPONIBILIDADE EM SISTEMAS DE INFORMAÇÕES LEGADOS

Dissertação apresentada ao Programa de Mestrado em Ciência da Computação Departamento de Computação da Universidade Estadual de Londrina, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Alan Salvany Felinto

Londrina
2014

Catalogação elaborada pela Divisão de Processos Técnicos da Biblioteca Central da Universidade Estadual de Londrina

Dados Internacionais de Catalogação-na-Publicação (CIP)

H565e Herrera, Rafael de Paula.
Uma EStratégia para migração e implantação de alta disponibilidade em sistemas de informações legados / Rafael de Paula Herrera. – Londrina, 2014.
91 f. : il.

Orientador: Alan Salvany Felinto.
Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual de Londrina, Centro de Ciências Exatas, Programa de Pós-Graduação em Ciência da Computação, 2014.
Inclui bibliografia.

1. Engenharia de software – Teses. 2. Processamento eletrônico de dados – Processamento distribuído – Teses. 3. Fluxo de dados – Transmissão – Teses. 4. Tolerância a falha (Computação) – Teses. 5. Sistemas operacionais distribuídos (Computadores) – Teses. I. Felinto, Alan Salvany. II. Universidade Estadual de Londrina. Centro de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU 519.68.02

RAFAEL DE PAULA HERRERA

UMA ESTRATÉGIA PARA MIGRAÇÃO E IMPLANTAÇÃO DE ALTA DISPONIBILIDADE EM SISTEMAS DE INFORMAÇÕES LEGADOS

Dissertação apresentada ao Programa de Mestrado em Ciência da Computação Departamento de Computação da Universidade Estadual de Londrina, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

BANCA EXAMINADORA

____
Orientador. Prof. Dr. Alan Salvany Felinto
Universidade Estadual de Londrina – UEL

____
Prof. Dr. Jacques Duílio Brancher
Universidade Estadual de Londrina – UEL

____
Prof. Dr. Wesley Attrot
Universidade Estadual de Londrina – UEL

____
Prof. Dr. Ronaldo Augusto de Lara Gonçalves
Universidade Estadual de Maringá – UEM

Londrina, 01 de Agosto de 2014

Em memória de Danilo Visin †

AGRADECIMENTOS

Aos meus pais Jonas Tadeu de Paula Herrera e Maria Alice Visin de Paula Herrera, por todo amor, paciência, compreensão, ensinamentos de vida, preocupações, amizade e cuidados desempenhados desde o momento de meu nascimento até quando decidi sair de casa com o objetivo de conquistar o mundo.

À minha esposa Juliana Cristina Marcarini, por todo seu amor, dedicação, compreensão, companheirismo e amizade. Sua presença em minha vida foi o grande fator determinante para que pudesse me tornar uma pessoa cada vez melhor e que fosse capaz de tirar forças, enfrentar e vencer todos os obstáculos que me foram impostos até o dia de hoje.

Ao meu Orientador, Prof. Dr. Alan Salvany Felinto, por ter acreditado em meu potencial, mesmo diante dos momentos mais difíceis, por toda sua dedicação, paciência, ensinamentos e principalmente pela amizade que construímos ao longo de minha jornada acadêmica.

Ao Prof. Dr. Jacques Duílio Brancher e Prof. Dr. Wesley Atrott pelos valiosos ensinamentos que me foram passados desde a época de graduação, pela amizade e pelo grande apoio nos momentos decisivos nessa caminhada.

Ao Programa de Mestrado do Departamento de Computação da Universidade Estadual de Londrina e a todos os funcionários que contribuíram com o desenvolvimento deste trabalho.

Aos amigos que sempre estiveram comigo em todos os momentos, especialmente aos nominados como: *Baraka, Bernault, Caipora, Carioca, Chucky, Craudião, Danz, Delphiman, Diemão, Dizus, Frangão, Gay, Gaúcho, Gordinho, Guarú, Joe, Lucão, Lucius, MapaMan, Massa, Mephisto, NS, Pajé, Passarinho, Paulão, Pinga, Pirata, Renatão, Trevis, Urso, Will, Xará, Zé Loko, Zombie* e muitos outros que não consigo me recordar, mas que certamente serão lembrados.

“The world ain't all sunshine and rainbows.
It is a very mean and nasty place.
It will beat you to your knees and keep you there permanently if you let it.
You, me or nobody is going to hit as hard as life.
But it ain't about how hard you're hit, it is about how hard you can get hit and keep moving forward, how much can you take and keep moving forward.
That's how winning is done!”

Rocky Balboa

HERRERA, Rafael de Paula. **Uma estratégia para migração e implantação de alta disponibilidade em sistemas de informações legados**. 2014. 93 f. Trabalho de Conclusão de Curso (Mestrado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2014.

RESUMO

Sistemas de Informações Legados são peças-chave para muitas Organizações e sua modernização não é uma tarefa trivial, representando um grande fator de risco. Neste trabalho, foi utilizado um Sistema de Informação Legado como Corpo de Prova e nele aplicamos um conjunto de técnicas que auxiliaram o processo de migração para uma infra-estrutura mais moderna, sem que suas operações fossem interrompidas. Para isso, foi desenvolvido um Middleware Distribuído, capaz de mitigar Pontos Críticos que pudessem inviabilizar o sucesso desta operação. Assim, foi possível concretizar o anseio de se modernizar o ambiente onde a solução encontrava-se implantada. Também foi atingido o objetivo estratégico de se possuir Alta-Disponibilidade entre múltiplas instâncias do serviço, tornando uma ponta crítica das operações completamente Tolerantes a Falhas.

Palavras-chave: Alta disponibilidade. Middleware distribuído. Migração. Sistemas de informações legados. Tolerância a falhas.

HERRERA, Rafael de Paula. **A strategy on migrating and deploying high availability to legacy information systems**. 2014. 93 p. Final Course Work (Master of Science in Computer Science) – State University of Londrina, Londrina, 2014.

ABSTRACT

Legacy Information Systems play key-roles to several Organizations and its upgrade is not a trivial task, representing a risk factor to its owners. In this work, there was set a Legacy Information System as our Body of Proof and then we managed a set of techniques to assist the migration process to a better infra-structure, without the operations being disrupted. To accomplish this task, a Distributed Middleware was developed and it was able to mitigate the Critical Points that could turn this operation into a fail. So it was possible to materialize the anxiety on modernizing the environment where the whole solution was located. There was also completed the strategic target of owning High-Availability between multiple instances of service, turning an entire critic operations endpoint into a completely Fault Tolerant one.

Key-words: Distributed middleware. Fault tolerance. High availability. Legacy information systems. Migration.

LISTA DE ILUSTRAÇÕES

Figura 1.1 – Lidando com um Sistema de Informação Legado 13
Figura 3.1 – Possíveis Inconsistências 37
Figura 3.2 – DDSs em Lista Circular 41
Figura 3.3 – Instância de Um Nó 41
Figura 3.4 – Entidades das DDSs e Relacionamentos 43
Figura 3.5 – Mapeamento de Requisições e Consultas SQL 44
Figura 3.6 – Estágios de Processamento Internos 45
Figura 3.7 – Proxy para Filas Internas 45
Figura 3.8 – Efetivador de Operações no BD 46
Figura 3.9 – Pool de Conexções com o BD 47
Figura 3.10 – Visão Geral da Solução 49
Figura 3.11 – Provedor de Serviços x Dispositivo Móvel 50
Figura 3.12 – BAS x MAS 51
Figura 3.13 – Provedor de Serviços x Cliente 52
Figura 4.1 – Clientes x Dispositivos Afetados 54
Figura 4.2 – LIS e Nova Infra-Estrutura 55
Figura 4.3 – Fluxo de Testes para Protocolos de Comunicação 56
Figura 4.4 – Tempo de Bloqueio Vs. Quantidade de Operações (20k) 59
Figura 4.5 – Tempo de Bloqueio Vs. Quantidade de Operações (100k) 60

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
BAS	*Border Application Server*
BD	Banco de Dados
CAS	*Customer Aplication Server*
DDS	*Distributed Data Structures*
DHT	*Distributed Hash Tables*
DI	*Dependencies Injection*
DIP	*Dependency Inversion Principle*
DP	*Design Patterns*
EAS	*Edge Aplication Server*
FTP	*File Transfer Protocol*
GSM	*Global Systems for Mobile Communications*
GPRS	*General Packet Radio Service*
HA	*High-Avaiability*
HTTP	*HyperText Transfer Protocol*
HTTPS	*HyperText Transfer Protocol Secure*
IoC	*Inversion of Control*
IP	*Internet Protocol*
JDK	*Java Development Kit*
LAN	*Local Area Network*
LIS	Sistemas de Informações Legados
MAS	*Main Application Server*
MD	*Middleware* Distribuído
OOP	*Objected Oriented Programming*
PF	*Packet Filter*
P2P	*Peer to Peer*
RAM	*Random Access Memory*
RDBMS	*Relational DataBase Management System*
RIAA	*Recording Industry Association of America*
Rx	*Reception*
SFS	*Shared File Systems*
SGBD	Sistema Gerenciador de Bancos de Dados

SQL	*Structured Query Language*
SSD	*Solid State Disk*
TCP	*Transmission Control Protocol*
Tx	*Transmission*
UDP	*User Datagram Protocol*
UUCP	*Unix-to-Unix-copy protocol*
WAN	*Wide Area Network*
WWW	*World Wide Web*

SUMÁRIO

1 INTRODUÇÃO 12

2 FUNDAMENTAÇÃO TEÓRICA 16

2.1 TRABALHOS RELACIONADOS 16

2.2 ESTRATÉGIAS DE MIGRAÇÃO 18

2.3 REDES PONTO A PONTO 20

2.3.1 Histórico 21

2.3.2 Popularização 22

2.3.3 Arquitetura e Topologia 24

2.3.4 Aplicabilidade 26

2.4 ESTRUTURAS DE DADOS DISTRIBUÍDAS 27

2.4.1 Tabelas Hash Distribuídas 28

2.5 INVERSÃO DE CONTROLE E INJEÇÃO DE DEPENDÊNCIAS 29

2.5.1 Recorrência Comportamental 30

2.5.2 Técnicas de Implementação 32

3 METODOLOGIA DE DESENVOLVIMENTO 35

3.1 MIDDLEWARE DISTRIBUÍDO 35

3.1.1 Alta Disponibilidade 35

3.1.2 Pontos Críticos 37

3.1.3 Arquitetura 40

3.2 ESTUDO DE CASO 48

4 RESULTADOS 53

5 CONCLUSÃO E TRABALHOS FUTUROS 61

REFERÊNCIAS 63

1 INTRODUÇÃO

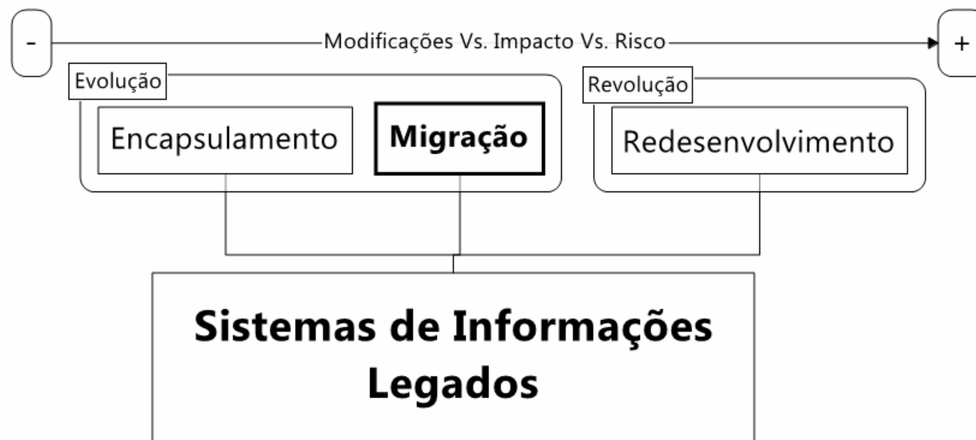
Sistemas de Informações são peças-chave para a evolução sustentável de praticamente todo segmento da indústria. Quando essenciais em uma organização, seu uso contínuo representa grande parte do Retorno do Investimento como um dos resultados mais perceptíveis ao longo da cadeia produtiva [26]. Em linhas gerais, o Retorno do Investimento (*Return of Investment* ou ROI) pode ser expresso como a razão entre os ganhos decorrentes de um investimento e o montante investido [11]. A mantenabilidade de um sistema pode ser resumida como o conjunto de fatores que influenciam em sua continua manutenção, permitindo o cumprimento de seus objetivos operacionais com mínimas despesas [9]. Uma relação diretamente proporcional entre o ROI que um Sistema de Informações proporciona e sua mantenabilidade, pode ser estabelecida, visto que a substituição por completo de um Sistema de Informações, já em produção, pode acarretar em perdas de recursos já investidos, justificando a importância de garantir sua preservação e, consequentemente, sua sobrevida.

Ao longo do tempo, Sistemas de Informações tendem a serem classificados como Sistemas de Informações Legados (*Legacy Information Systems* ou LIS), em decorrência de várias razões, que podem ser abrangidas desde o fato de que *Hardwares* e/ou Softwares em que se baseiam se tornam obsoletos, até fatores sociais como a falta de um completo entendimento sobre seu ciclo de vida completo ou de sua base de código, que podem ser causados pela ausência ou perda de documentação e/ou mesmo pela falta de mão-de-obra especializada [2, 14]. Para se garantir a sobrevida aos Sistemas de Informações Legados, se faz necessário escolher entre abordagens que visam encapsular, evoluir ou re-desenvolver suas bases de código. Adicionalmente, para sistemas já em produção, onde se deseja realizar migrações sem interrupções de serviço, estratégias guiadas por *middlewares* especialmente projetados para este tipo de tarefa mostram grande versatilidade de implementação, o que aumenta as chances de sucesso em manobras dessa ordem [1, 2, 3, 15, 23].

A Figura 1.1 retrata as diferentes abordagens que podem ser consideradas quando se está lidando com um LIS [14]. É possível realizar um isolamento completo do mesmo [5] ou migrá-lo para um ambiente mais moderno, ambas as abordagens são caracterizadas como uma evolução da solução. Já uma abordagem completamente distinta, onde o LIS é descartado e se concebe uma nova solução, por meio de re-desenvolvimento, está caracterizada como revolução, devido às quebras de paradigma que podem estar

associadas [2]. Todas estas opções possuem um risco envolvido e, este é diretamente proporcional à quantidade de mudanças exigidas e ao impacto que trazem ao ambiente.

Figura 1.1 – Diferentes abordagens ao se lidar com um LIS.



A evolução de sistemas, em linhas gerais, pode ser associada ao grau de dificuldade encontrado na capacidade de ser realizar uma decomposição de seu todo em partes relativamente menores [4]. Ou seja, uma solução monolítica pode ser decomposta, no intuito de: *(i)* facilitar sua administração em ambiente de produção, *(ii)* homogeneizar seus testes durante as fases de desenvolvimento e *(iii)* possibilitar a execução de modificações singulares em sua base de código, refletindo em comportamentos mais previsíveis e atômicos. No entanto, as características qualitativas que determinam o grau de sucesso que esta abordagem oferece são implícitas a cada tipo de solução e por isso precisam ser estudadas individualmente, pois carregam especificidades e níveis de complexidade inerentes à sua natureza enquanto Sistemas de Informações Legados [24].

Determinados tipos de sistemas, não possuem margem para migrações onde pode-se estabelecer janelas de tempo onde ocorrerão indisponibilidades com a interrupção dos serviços. Por este motivo, estratégias especiais são empregadas para que ambos os sistemas, o Legado e seu Sucessor, possam ser executados simultaneamente até que um possa suplantar o outro de maneira mutuamente exclusiva [29]. Técnicas baseadas em Arquitetura Orientada a Serviço fazem com que seja possível se estabelecer ilhas intercomunicáveis de Software, implementadas afim de se executar uma re-engenharia que visa a evolução de um sistema, não necessitando de sua substituição por completo e oferecendo um reuso substancial de suas funcionalidades [20]. Dessa maneira, é realizado um isolamento de partes do sistema, disponibilizando meios para troca de dados utilizando-se dialetos bem estabelecidos por Interfaces de Programação de Aplicações.

Por outro lado, a implementação de mecanismos distribuídos e, consequentemente, tolerantes a falhas não é uma tarefa trivial quando o contexto são aplicações que dependem exclusivamente do armazenamento de estados internos de fluxo de execução em memória. Para garantir a consistência de operações em um serviço que uma vez venha a se tornar distribuído, levando em conta a possibilidade de falhas em servidores de aplicação, seus dados devem ser replicados ao longo de um número suficiente de nós na rede. Sendo dependente de múltiplos links de Internet e tendo sua infra-estrutura física sujeita a manutenções periódicas, torna-se inefetivo se empregar alguma solução cuja performance esteja limitada exclusivamente à Rede de Área Local exclusiva a um único Data Center.

Sistemas cujas fontes de dados são Sistemas de Gerenciamento de Bancos de Dados Relacionais, podem apresentar fatores que impactam na performance como um todo, podendo esta ser degradada pelo tempo de bloqueio ocasionado na espera por execução de operações. Caso a arquitetura de seu fluxo seja orientada ao disparo de novas *thread*s, isso pode agravar ainda mais o tempo de resposta. Sistemas de Informações Legados podem sofrer principalmente com alta-concorrência em operações de leitura e escrita sobre suas Bases de Dados Relacionais e estas podem ter sua demanda regulada por meio de um fluxo que também necessite de manutenções que agreguem técnicas de chamadas assíncronas e utilização de *callbacks*, aliviando a carga dos servidores e aumentando o desempenho dos serviços oferecidos.

Com base neste conjunto de asserções, foi elaborado no presente trabalho, um modelo conceitual de migração, capaz de ser empregado em Sistemas de Informações Legados, oferecendo uma sobrevida suficiente até que possam ser adequados a arquiteturas compatíveis com crescentes demandas. Desta maneira, estes poderão se enquadrar em padrões tecnológicos de mercado mais modernos com relação aos que vigoravam nas datas de suas concepções. A estratégia de migração empregada é guiada por um *Middleware* distribuído, que foi desenvolvido levando em consideração as particularidades de um Sistema de Informações Legado específico, oferecendo níveis performáticos para operações críticas e sendo tolerante a falhas. Um conjunto de ferramentas e técnicas auxiliares é exposto como apêndices às validações sobre coesão comportamental em todo o ciclo de vida do sistema, durante suas fases de transição.

Visando um delineamento eficiente, este trabalho foi dividido de acordo com os seguintes tópicos:

- **Capítulo 2:** São descritos os trabalhos que influenciaram a concepção da solução aplicada em campo. Estratégias de migração implantadas em Sistemas de Informações Legados são abordadas, assim como Redes P2P são exploradas, por suas características fornecerem meios para que uma arquitetura distribuída pudesse ser empregada no desenvolvimento de um *Middleware*, capaz de eliminar pontos singulares de falha no sistema analisado. Estruturas de Dados Distribuídas são analisadas como parte fundamental das técnicas de armazenamento em *Grid* de Memória. O padrão de projeto Inversão de Controle é apresentado como parte fundamental ao desacoplamento dos componentes e como forma de concretizá-lo, foi utilizada a técnica de Injeção de Dependências.

- **Capítulo 3:** É realizado um Estudo de Caso, onde um *Middleware* Distribuído foi desenvolvido como um dos produtos principais deste trabalho. O modo como foi projetada e concebida sua arquitetura é exposto em detalhamento de alto-nível. Adicionalmente, é descrito o ambiente onde se encontrava implantado o sistema tido como objeto de estudos neste trabalho, mostrando sua complexidade e sistemas integrados, fornecendo uma visão geral da problemática envolvida nas melhorias propostas.

- **Capítulo 4:** Os Resultados obtidos por meio da execução do estudo de técnicas, neste trabalho apresentadas, e a implantação das soluções desenvolvidas são discutidas, fornecendo uma visão geral sobre as decisões de projeto tomadas em conjunto com análises de desempenho.

- **Capítulo 5:** O presente trabalho é finalizado, sendo apresentados os objetivos cumpridos com seu desenvolvimento e sugestões para trabalhos futuros, a fim de contribuir cientificamente com o estudo sobre Sistemas de Informações Legados.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 TRABALHOS RELACIONADOS

Realizar a manutenção de um LIS nem sempre é tarefa trivial e requer conhecimentos sobre o intercâmbio de dados entre os sistemas, várias abordagens sobre como lidar com este tipo de situação são descritas em [2]. Sua principal contribuição foi fornecer diferentes pontos de vista sobre como montar uma estratégia de migração, tendo visto as que foram apresentadas, com seus pontos altos e baixos. Foi criado um Middleware Distribuído, levando-se em consideração o conjunto de boas práticas levantadas em [18, 23] e, seu principal objetivo era cuidar que problemas de performance não impedissem a solução de crescer de acordo com a demanda de operações por Bancos de Dados Relacionais [14].

Tendo como base que a evolução de um sistema está relacionada ao quão difícil é sua decomposição [4], uma boa solução deve tomar uma solução monolítica e transformá-la em peças menores de *software*. Isso permite a realização de testes e modificações de modo mais atômico, sendo que seus parâmetros que determinam o nível de sucesso desta abordagem não são explícitos ou quantitativos [24].

Uma abordagem sobre paralelismo entre uma solução legada e sua migração é discutida em [16], nela os aspectos da regra de negócio, infra-estrutura e desenvolvedores são analisados de modo a compor uma estratégia baseada em SOA. Foi utilizada a premissa de que um modelo de migração precisa ser gradual enquanto todas as partes têm seu funcionamento garantido [27], isso forneceu diretrizes sobre como organizar uma sequência de passos apropriada para o problema discutido no presente trabalho.

Uma estratégia de migração foi desenvolvida com *screen-scraping*[1] de um sistema Web antigo e adaptado para uma interface nova, de modo gradual [7], que complementou a visão sobre estratégias de isolamento sistêmico. Uma pesquisa foi realizada sobre transformar aplicações legadas em SOA [10] e uma estratégia de duas fases transforma um LIS a um Webservice [22], em ambos os casos nota-se como uma evolução gradual pode ser empregada para melhor gerenciar as mudanças propostas.

Os passos utilizados para se tornar uma solução consolidada em um sistema SOA público são discutidas em [26], onde os componentes afetados compunham uma solução de larga-escala e foram transpostos para este distinto paradigma. Uma fase de engenharia

[1] *Screen-Scraping* é uma técnica onde um programa de computador extrai dados da saída de outro programa de computador, se assemelhando com uma “captura de tela”.

reversa revela as saídas necessárias para que fosse possível a nova composição. Tendo isso em vista, pudemos delinear uma fase de testes capaz de garantir o funcionamento de características antigas, ao mesmo tempo em que novas características pudessem ser incorporadas à base de código.

Tratar sistemas como *Black-Boxes* foi abordado em [12], com a utilização de sistemas multi-agentes para o encapsulamento de funcionalidades. Foi utilizada uma técnica muito semelhante durante os testes de protocolo realizados neste trabalho. Os mesmos foram capazes de garantir coesão durante a fase de implementação. Em [5] uma abordagem semelhante é utilizada no intuito de extender o tempo de vida de Sistemas de Informações Legados, discutindo a efetividade quando empregadas tais técnicas.

A re-engenharia de sistemas e evolução destes para SOA é posta em [20] por meio de um estudo de casos que mostra como a reconstrução de arquitetura foi utilizada em um sistema, base na tomada de decisões. Migração Vs. Integração com SOA e a reconstrução de arquiteturas para esse paradigma são abordados em [21].

Um framework que integra interfaces gráficas de Sistemas de Informações Legados com Web Services foi apresentado em [31]. Para este tipo de construção, foi necessário empregar técnicas de encapsulamento que trataram o sistema analisado como uma caixa preta. Sua exposição em forma de SOA remete à técnicas de *Wrapping* analisadas na concepção deste trabalho.

A replicação de dados é discutida em [6], mostrando as principais diferenças entre abordagens utilizadas pela academia e a indústria. Performance, disponibilidade e administração são discutidos neste sentido. Técnicas da academia e da indústria são cruzadas, afim de se mostrar como podem cooperar no sentido de que a replicação de dados utilizando *Middlewares* pode ser mais difundida em ambos os meios.

Em [25, 17] são discutidos aspectos de diferentes estruturas de dados, tais como filas e estágios de processamento, para serviços voltados à Internet. O percursor destas pesquisas é o trabalho [28], que forneceu diretrizes para a elaboração de uma arquitetura interna baseada em *pipelines*, o que torna a vazão de processamento muito maior quando comparada à abordagens exclusivamente centradas em threads de processamento para novas mensagens.

2.2 ESTRATÉGIAS DE MIGRAÇÃO

Uma vez que um Sistema de Informações possua características capazes de classificá-lo como sendo um Legado para uma Organização, é recorrente que existam dúvidas sobre como lidar com o fato de que tal sistema possa, muitas vezes, ser parte essencial da cadeia produtiva à qual a referida Organização é compreendida. Existem inúmeras abordagens que podem ser aplicadas em situações assim, como sua completa re-engenharia, substituição total, isolamento ou a utilização de técnicas que visam dar continuidade ao seu processo evolutivo.

Praticar a re-engenharia em Sistemas de Informações Legados (LIS) é uma tarefa que pode ocasionar uma elevada complexidade operacional, pois pode necessitar dispor de meios eficientes para validar a regra de negócio empregada em cada uma das funcionalidades já existentes, verificar extensivamente quais módulos atendem sua utilização e classificar dentre estes quais são inativos, oferecer meios para que as integrações existentes entre seus produtos finais e recursos externos continuem funcionado e, por fim, sendo incapaz de garantir que problemas conhecidos e recorrentes não serão apenas transpostos para uma realidade diferente, abrigada por uma arquitetura mais moderna.

A substituição total de um LIS por outra solução pode ser adotada, desde que a Organização esteja disposta a arcar com as conseqüências de uma decisão desta natureza. Caso o LIS em questão seja dotado de uma base de conhecimento armazenada em meios incompatíveis, do ponto de vista de um candidato que possa lhe substituir, certamente será preciso realizar um trabalho que envolva re-modelagem e adaptações dos dados correntes para um meio que seja adequado. Essa abordagem pode envolver altos custos de consultoria, dependendo do nível e da quantidade de customizações envolvidas na adaptação de uma solução escolhida para sua realidade, seja sua licença classificada como proprietária ou livre.

É comum que se isole um LIS, protegendo-o de interações diretas com outros sistemas ou usuários. Essa idéia consiste basicamente em se manter um agente intermediário, capaz de realizar o papel de Gateway ou *Proxy*-Reverso entre o LIS e os recursos com que este necessita se comunicar.

Ao realizar o isolamento de um LIS, é importante saber se este se comunica com humanos, pois nesse caso é possível empregar técnicas de *screen-scraping*. Deste modo, são extraídas as mensagens que antes deveriam ser exibidas diretamente na tela do usuário, para que sejam ou transformadas ou encaminhadas para uma nova Interface Homem-Máquina, onde a exibição de tais informações ocorrerá de maneira apropriada.

De modo complementar, as entradas de dados que antes eram recebidas por meio de dispositivos padrão, antes acionados pelo usuário, tal como é feito com o teclado, são intercambiadas por mecanismos auxiliares, capazes de interceptar controles de interface específicos presentes no LIS, como *input-texts* ordinários. Uma vez providos dos dados necessários para completar uma determinada operação que exige a entrada dos mesmos, estes mecanismos auxiliares se encarregam de prover um fluxo de entrada dos dados em questão para os controles de interesse, disparando eventos de submissão para processamento e reagindo mediante às exceções, quando necessário, simulando assim um ciclo legítimo de interações que antes ocorriam somente entre o LIS em questão e seus operadores humanos.

Ainda explorando as conseqüências que implicam eventuais isolamentos sistêmicos, agora compreendidos somente por interações entre máquinas, surge a necessidade de elementos intermediadores que, a exemplo da abordagem anterior, sejam capazes de estabelecer ligações entre todos os pontos envolvidos nos processos de comunicação que sejam necessários durante a execução de tarefas que envolvam dados provenientes do LIS.

Esse modo de lidar com o problema sugere a criação de uma camada intermediária capaz de responder às requisições recebidas e, se preciso for, requisitar dados de outros serviços e encaminhá-los ao LIS, tendo antes realizado uma normalização nestes de modo que sejam convertidos para o formato esperado ao qual opera o referido sistema, sendo assim capaz de interpretar as informações e reagir de maneira apropriada. Essa camada intermediária acaba se comportando como um *Gateway* entre aplicações distintas, necessitando conhecer tanto o dialeto empregado pelo protocolo de comunicação ao qual o LIS adota quanto os demais já em utilização por outros sistemas já existentes.

Também é possível que esta camada seja concebida na forma de um *Web Service*, estabelecendo uma Interface pública e bem definida para a comunicação com serviços externos. Assim, todos os sistemas que necessitem trocar informações com o LIS deverão lidar com o dialeto de seu intermediário. Esse ponto é vantajoso estrategicamente somente se constarem na Organização, planos futuros de desligamento do LIS em questão, sendo este completamente absorvido, em termos de funcionalidades, ou pela camada intermediária, e então esta seria promovida à categoria de novo serviço, ou pelos demais serviços presentes no ecossistema.

Em contraposição a todos os pontos apresentados como possíveis soluções, existe a possibilidade se executar uma migração. Esse tipo de conceito envolve a transposição de dados ou fluxos de trabalho para novos moldes arquiteturais. Uma migração de dados

requer ajustes em termos de aplicação. Já uma migração de fluxos de trabalho pode não exigir o mesmo para seus dados.

Uma vantagem encontrada em termos de migração é que esta abordagem possui sobre as outras o fato de não desperdiçar o investimento previamente realizado para a concepção do então nominado LIS. Outro aspecto positivo consiste na diminuição do risco envolvido em mudanças estruturais, realizando-se então modificações pontuais e integrando-as gradualmente à base de código, em termos de funcionalidades chave.

Esse caráter baseado em mudanças evolutivas traz ao LIS em questão, a possibilidade de obter uma sobrevida maior e continuar sendo utilizado por mais tempo, até que todas as suas atribuições funcionais sejam absorvidas ou substituídas, pouco a pouco, por implementações mais robustas. O parâmetro de uma migração bem-sucedida pode ser descrito em linhas gerais como a implantação de mudanças que envolvam o LIS, em ambiente produção, mas que estas modificações não causem indisponibilidades nas operações do ecossistema de Sistemas de Informações da Organização ao qual pertence.

2.3 REDES PONTO A PONTO

Redes Ponto a Ponto (*Peer to Peer*, ou P2P) são modalidades de redes de computadores organizadas de maneira descentralizada. Nós individuais deste tipo de rede, geralmente referidos como *peers*, agem de maneira dual: são ao mesmo tempo provedores e consumidores de recursos. Essa característica se contrapõe ao modelo clássico de serviços, chamado Cliente-Servidor, onde os nós clientes sempre requisitam recursos que se encontram centralizados em servidores bem definidos.

Uma característica relevante encontrada em Redes P2P é que usualmente todos os nós interconectados entre si possuem as mesmas funcionalidades e igual valor perante o todo. Tarefas são partilhadas entre os membros da rede e cada unidade contribui com uma porção de seus recursos, no intuito de se obter um resultado final esperado, não necessitando assim que existam servidores centrais ou computadores com a finalidade de coordenar tarefas para máquinas específicas.

Pode-se projetar uma rede dessa natureza para que cada um de seus nós processe partes de um grande problema. Uma vez que uma tarefa seja classificada como complexa o suficiente para tornar sua execução exaustiva para uma máquina apenas, é possível que esta seja designada para ser executada em uma Rede P2P, afim de que seja completada mais rapidamente. Para que sua execução seja possível em termos de uma Rede

P2P, é preciso que a tarefa em questão seja paralelizável. Assim, cada uma das partes em que esta puder ser dividida será então distribuída para um número de membros participantes da rede.

Também é possível planejar Redes P2P para armazenamento e troca de informações. Neste caso, os recursos compartilhados têm outra natureza e sua principal característica é a necessidade de dispor de dispositivos especializados em armazenar dados em cada nó da rede. Sendo assim, os sistemas envolvidos irão disponibilizar como recursos às redes aos quais pertençam, seus Discos Rígidos ao invés de seus Processadores. Caso os dados a serem disponibilizados tenham alta volatilidade, pode-se considerar o uso de Discos de Estado Sólido (*Solid State Disks* ou SSDs) ou mesmo Memória de Acesso Randômico (*Random Access Memory* ou RAM) como recursos compartilhados.

2.3.1 Histórico

As Redes P2P foram amplamente utilizadas em aplicações de domínio específico por aproximadamente 30 anos antes que pudessem se tornar populares. A própria Internet foi projetada no final dos anos 60 para atuar como uma Rede P2P: A ARPANET Original se conectava à Universidade da Califórnia de Los Angeles, ao Instituto de Pesquisa de Stanford, à Universidade da Califórnia de Santa Barbara e à Universidade de Utah não como uma Rede Cliente-Servidor, mas como forma de *peers*, onde cada nó tinha uma importância e participação equivalente na rede.

As primeiras aplicações populares da Internet, como àquelas baseadas nos serviços FTP ou Telnet dispunham da capacidade de agir como Clientes e Servidores simultaneamente, pois todos os nós eram capazes de acessar uns aos outros, apresentando um padrão homogêneo e simétrico de utilização. Isso significa que nos estágios iniciais da mini-computação e dos mainframes, os servidores também agiam como clientes.

Um exemplo histórico foi a criação da *Usenet*, em 1979, realizada por dois estudantes de graduação da Universidade de Duke e um estudante de graduação da Universidade da Carolina do Norte. Baseando-se em um protocolo de cópia entre Sistemas *Unix* (*Unix-to-Unix-copy protocol* ou UUCP), uma máquina *Unix* poderia discar automaticamente para outra máquina Unix, trocar arquivos e se desconectar. Desta forma, os estudantes de ambas Universidades eram capazes de postar e receber notícias sob determinados tópicos, estabelecendo uma comunicação efetiva entre instituições. Por fim, a

Usenet cresceu de apenas dois pontos para centenas de milhares de pontos, retendo as características básicas de um modelo P2P.

Um *peer* da *Usenet* era capaz de se conectar com o restante do mundo indicando como pontos de troca de informação outros nós conhecidos, que de maneira comum eram listas de interesse hospedadas por seus provedores de conexão (*Internet Service Providers* ou ISP). Cada ISP estabelecia conexão com outros pontos de troca de informação, atualizando periodicamente os dados que eram então disponibilizados para a busca de seus usuários.

Durante a idealização da World Wide Web (WWW), Tim Berners-Lee compartilhava de uma visão onde a rede que estava sendo projetada se aproximaria a um modelo P2P, pois cada usuário era visto como consumidor de conteúdo e ao mesmo tempo como potencial produtor deste. Cada utilizador da WWW poderia estabelecer ligações entre conteúdos distintos, independente de sua localização, por meio de um mecanismo então denominado como *HyperLink*. Estando ligados uns aos outros, os *HyperLinks* formariam a noção de Teia (*Web*) entre conteúdos que, estando localizados em peers capazes de servi-los e recebê-los simultaneamente caracterizavam uma rede nos moldes P2P. Essa noção era possível pois levamos em conta que naquele momento histórico cada nó dispunha de acesso pleno aos demais, sem dispositivos de rede intermediando os acessos ou realizando tarefas de segurança das informações, tais como conhecemos nas topologias atuais como quando se utiliza *Firewalls*, *Load-Balancers* ou mesmo *Proxies*-Reversos.

2.3.2 Popularização

No mundo corporativo, várias empresas adotaram a utilização de Redes P2P e notadamente temos o exemplo da Intel, que em um estudo de caso revelou ter economizado cerca de 500 milhões de dólares. Para que isso fosse possível, essa empresa desenvolveu um sistema chamado *NetBatch*, que foi distribuído por mais de 10 mil computadores localizados em 25 localidades distintas ao redor do mundo. O sistema apresentado tem como intuito a capacidade de rodar simulações, fornecendo a seus engenheiros um grande poder computacional distribuído. A Intel utiliza tecnologias com base em modelos P2P desde os anos 90 para cortar custos relacionados com a elaboração dos projetos de seus chips e somente este caso isolado de utilização mostrou uma possibilidade concreta de economia nas despesas com aquisição de novos mainframes por um período de dois anos [8].

A popularização da Internet para fins de utilização pessoal e comercial, sem que estivesse atrelada às aplicações de uso específico se difundiu em meados dos anos 90. Neste período, as circunstâncias fizeram com que as redes P2P raramente fossem utilizadas por usuários convencionais. Estes, por sua vez, passaram a ter acesso a uma variedade de aplicações que permitiam a ocorrência de atividades em larga escala antes praticamente inexistentes entre si, como a troca de *E-Mails* e principalmente visualização de páginas da Web. A *WWW* encontrava-se em um estágio evolutivo onde sua arquitetura predominante era a de Servidores e Clientes como entidades distintas: Um usuário dispunha de um navegador capaz de realizar o acesso a um conteúdo hospedado em um Servidor *Web*, propriedade de uma instituição que tinha por finalidade lhe provisionar conteúdo ou serviços. Uma gama de novas possibilidades surgiu, desde a visualização de notícias até a concretização de vendas, o que mostrou cada vez mais forte sua característica de *downstream*, tendo-se como referencial os provedores de serviços enviando dados em direção aos seus usuários.

Tendo em vista um cenário onde se predominam arquiteturas de Clientes e Servidores completamente independentes entre si, as Redes P2P tais como conhecemos atualmente se popularizaram somente no final dos anos 90. Shawn Fanning, um calouro da Universidade de *Northearnstern* criou em 1999 um software cujo intuito era compartilhar músicas entre seus usuários. Esta aplicação, batizada como *Napster*, alcançou utilização em escala global e permitiu que seus usuários trocassem arquivos de música no formato popularmente denominado como MP3 (*MPEG-1* ou *MPEG-2 Audio Layer III*). Cada usuário, ou peer como eram referidos na rede, era capaz de servir e receber músicas de maneira simultânea.

Esse episódio ficou mundialmente conhecido, pois nunca antes foi constatado um fenômeno com os mesmos precedentes. A empresa responsável pelo desenvolvimento e distribuição deste software de compartilhamento de músicas foi legalmente processada pela Associação da Indústria de Gravação da América (*Recording Industry Association of America* ou RIAA), gravadores, selos de gravação e artistas. Sem sucesso, após batalhas judiciais e uma tentativa de se tornar um serviço pago, a empresa responsável por esta tecnologia foi obrigada a cessar suas atividades em Julho de 2001.

No período final de operações do *Napster*, surgiram no mercado novas aplicações com o propósito de compartilhar arquivos por meio da Internet, a exemplo do sucesso obtido com o seu predecessor, utilizando de Redes P2P como base tecnológica. A troca de arquivos então foi estabelecida de maneira arbitrária, contando com softwares

especializados em músicas, notadamente o *Audiogalaxy* e outros que não se limitavam ao compartilhamento de formatos específicos como *Kazaa*, *Gnutella* e *iMesh*.

Notadamente, verificamos que a rede *Gnutella* possui em sua arquitetura diferenças que fizeram desta imune às apelações judiciais, como as quais foram capazes de obrigar o término de operações da rede *Napster*. Sua Rede P2P foi concebida de maneira completamente descentralizada, diferente da rede *Napster* que continha *peers* essenciais para seu funcionamento. Esses nós eram desta maneira centralizados e detidos pela empresa que por eles era responsável. Sendo assim, a rede *Napster* era tecnicamente semi-descentralizada, pois havia uma parte totalmente independente em sua estrutura, composta por seus usuários, representando o maior volume de *peers* interconectados na rede. Sem que existisse uma autoridade central para redes como *Gnutella*, suas operações persistem até os dias de hoje e não são suscetíveis a bloqueios tecnológicos garantidamente efetivos e imunes a técnicas de evasão como tunelamentos ou utilização de *proxies*.

2.3.3 Arquitetura e Topologia

Uma Rede P2P é projetada partindo-se da premissa que nós iguais, ou peers, possam funcionar simultaneamente como Clientes e Servidores de recursos para outros nós da rede a qual estejam interconectados. Esse modelo difere em sua essência do clássico Cliente-Servidor, onde os recursos encontram-se centralizados no Servidor e seus Clientes requisitam tais recursos conforme a demanda. Um exemplo onde ocorre a troca de arquivos entre Cliente e Servidor, utilizando o modelo clássico, é totalmente compreendido pelo fluxo que determina o Protocolo de Transferência de Arquivos (*File Transfer Protocol* ou FTP) onde os clientes necessitam utilizar um software específico para estabelecer a conexão com o servidor que, por sua vez, necessita executar ininterruptamente um *daemon* especializado em servir arquivos para os clientes que nele se conectarem, por meio da porta de serviços a qual lhe fora destinada (21, por padrão) e que utilizem o dialeto que estabelecido pelo protocolo para todas as interações.

Como noção básica do significado de um overlay de rede, temos que este consiste basicamente na construção de uma rede virtual no topo de outra rede, como por exemplo, a Internet pôde um dia ser construída como uma camada acima da rede telefônica. Desta maneira, temos que as próprias Redes P2P são *overlays* virtuais construídos sob o topo da Internet. Os nós presentes nesta rede especial formam um subconjunto de nós, também presentes na rede física.

Os dados que trafegam em uma Rede P2P podem ser intercambiados entre nós com o auxílio de uma rede TCP/IP (*Transfer Comand Protocol* e *Internet Protocol* respectivamente), no entanto existe a camada de aplicação que abstrai toda a comunicação ocorrida, dando a impressão aos nós pertencentes à rede de que esse comportamento ocorreu de modo transparente. O modo como são concebidos os overlays de rede é capaz de dizer qual é a categoria de Rede P2P implementada, devido a diferenças elementares entre mecanismos de indexação e descoberta de nós.

A noção de classificação de uma Rede P2P pode variar de acordo com a maneira como seus nós se interconectam por meio à rede *overlay*. Também exercem influência nessa classificação o modo como os recursos são indexados e distribuídos. São duas as possíveis classificações para Redes P2P: Desestruturadas e Estruturadas.

As Redes P2P Desestruturadas não implicam topologias específicas em suas redes overlay, elas simplesmente são constituídas por nós que estabeleceram conexões entre si de maneira randômica. Notadamente temos o protocolo da rede *Gnutella*, que segue esta definição. Pela ausência de uma estrutura global, essa modalidade de rede é mais fácil de ser construída e permite otimizações que levem em consideração as diferentes regiões da rede overlay. Pelo fato de que cada nó atua da mesma maneira em uma rede deste porte, essa configuração é robusta e reage bem face à altas taxas de conexões e desconexões por parte de seus integrantes.

No entanto, o fato de não serem definidas estruturas em se tratando de redes overlay para as Redes P2P Desestruturadas, as tornam ineficientes durante consultas pela localização de determinados dados, disparando uma grande quantidade de solicitações entre seus membros, o que causa um alto volume de tráfego decorrente de mensagens como estas sendo repassadas ao longo de toda rede, fato este que aumentando a utilização de memória e processamento de cada nó que processe a requisição de busca. Também não há garantias de retorno do resultado buscado, pois os nós que contém os dados podem estar passando por momentos de desconexão enquanto a busca é realizada. Exemplificando o compartilhamento de arquivos em Redes P2P, existe uma tendência de conteúdos populares serem encontrado mais rapidamente, por estarem presentes em um maior número de *peers*, ao passo que conteúdos mais restritos terão chances menores de serem encontrados em buscas eventuais.

Em contrapartida, existem Redes P2P que são classificadas como Estruturadas, pois impõe à rede overlay uma organização que obedece a uma topologia específica. Seu protocolo garante meios para os quais os clientes possam realizar buscas de maneira eficiente ao longo dos nós da rede, mesmo que o recurso desejado seja escasso. A

ocorrência mais comum de Redes P2P estruturadas são aquelas pertencentes às implementações de Tabelas *Hash* Distribuídas (*Distributed Hash Tables* ou DHT). Nessa abordagem, são utilizados algoritmos de *Hash* Consistente para determinar a localização exata onde os recursos serão armazenados perante todos os nós que constituem a rede em sua totalidade. Isso permite que buscas sejam realizadas com auxílio de Tabelas *Hash*.

Como características principais, as Tabelas Hash utilizam internamente uma combinação especial chamada Par *Chave x Valor* (*Key x Value Pair* ou simplesmente *K x V*). Esse arranjo é tecnicamente denominado como *Array* Associativo e permite armazenar seus registros de maneira eficiente. A Chave representa uma representação rotular pela qual se deseja obter um determinado dado, chamado de Valor. Internamente, o Valor ou dado bruto é armazenado em um segmento específico, determinado de acordo com o resultado obtido pelo processamento de uma função de *Hash* aplicada sobre a Chave com que se associa.

Buscas realizadas em Tabelas *Hash* são notadamente eficientes quando comparadas com buscas realizadas em outras estruturas de dados. A complexidade que lhes é conferida por meio de notação assintótica equivale a $O(1)$ para o melhor caso e $O(n)$ para o pior caso. Por essa razão fundamental, as DHT são alternativas eficientes a busca de dados dispersos ao longo dos nós em uma Rede P2P: As Chaves que rotulam os Valores buscados, após passarem por uma Função de *Hash* Consistente, apontam diretamente para os nós da rede que detém as informações desejadas. Mas, para conseguir manter a eficiência durante as buscas, é necessário que os nós mantenham atualizadas as informações sobre seus vizinhos que contém características similares às suas, quanto aos dados que carregam, isso torna essa modalidade de rede ineficiente quanto às altas freqüências com que múltiplos nós podem entrar e sair da rede arbitrariamente.

2.3.4 Aplicabilidade

Em termos de Arquitetura de *Software*, a utilização de Redes P2P oferece a vantagem de remover com facilidade os Pontos Únicos de Falhas presentes em sistemas cujo modelo seja o clássico Cliente-Servidor. Com o advento de regiões críticas em código que por ventura possam guardar estados de transações em curso, há o risco de se perder estes dados em caso de interrupção abrupta de serviço. Também existem circunstâncias onde se dispõe de somente uma instância de um serviço em especial que, caso esteja indisponível afetará a todos os seus clientes, sem que alternativas lhes sejam oferecidas enquanto ocorrem manutenções.

Conforme o grau de criticidade das operações envolvidas em Sistemas de Informações, se faz necessário que sejam elaboradas políticas de contingência para casos de falhas em partes de uma solução como um todo. Uma saída para casos como este é oferecer pontos alternativos para os clientes que dependem da solução, como é o caso de múltiplos servidores hospedando o mesmo serviço.

Ao se disponibilizar várias instâncias de um mesmo serviço para os clientes, é importante manter entre todas elas um funcionamento uniforme e consistente, caso existam regiões críticas de código responsáveis por transações ou dados de estado importantes durante a troca de informações.

Uma vez identificados os Pontos Únicos de Falhas, como sendo em suma dados que não podem se perder e, em caso de falhas, devem estar replicados ao longo de todos os nós que provêm serviços, as Redes P2P são uma alternativa e resolvem este problema de modo conciso. Quando um nó apresentar falhas, este pode ser rejeitado pela rede e os demais passam a responder por ele, balanceando entre si as responsabilidades. Em caso de uma queda súbita de um nó problemático, devido à falhas ocasionadas por inúmeras razões, o mesmo funcionamento anteriormente descrito é acionado, tendo os nós vizinhos da rede assumindo as responsabilidades pelo nó faltante.

Conforme aumentar ou diminuir a demanda pelo serviço, podem-se então instanciar mais servidores providos com o mesmo *daemon*, agora capaz de operar em redes P2P, e a capacidade de operações da rede é igualmente escalada em termos de suas necessidades, seja adicionando nós para suprir uma alta demanda ou desligando de alguns nós por alguma forma de ociosidade sazonal presente na rede.

2.4 ESTRUTURAS DE DADOS DISTRIBUÍDAS

Estima-se que em torno de 39% da população mundial, no ano de 2013, utilize algum serviço com base na Internet. Para suprir toda a demanda de utilização, são utilizados mecanismos capazes de garantir a massificação e alcance de tais serviços. Se torna desejável que para tal categoria de serviços, sejam empregados parâmetros de qualidade similares aos que são encontrados em sistemas de telefonia e de distribuição de energia. Desta maneira, Sistemas de Informações podem se tornar resilientes à falhas parciais ao mesmo tempo em que se mantém altamente disponíveis. Também são características desejáveis a habilidade dos Sistemas de Informações se tornarem escalares em termos amplos e até mesmo

populacionais, enquanto são mantidas: *(i)* a consistência dos dados de sua base de usuários e *(ii)* sua gerência operacional.

É grande o esforço envolvido na tarefa de tornar um serviço capaz de atingir a todos os requisitos apresentados, visando garantir sua longevidade operacional. A situação torna-se ainda mais crítica quando existe a necessidade de se gerenciar grandes volumes de estados persistentes em memória. É preciso que tais estados permaneçam consistentes e disponíveis, mesmo que diante de condições críticas como falhas ou indisponibilidades. Para atingir este objetivo, uma das técnicas utilizadas, consiste em se provisionar de uma camada auto-gerenciada de armazenamento distribuído, sendo esta denominada como Estruturas de Dados Distribuídas. Assim, dados essencias às operações dos sistemas envolvidos, são concentrados ao longo de múltiplas localidades, em instâncias completamente independentes entre-si e autônomas quanto seu funcionamento.

Estruturas de Dados Distribuídas apresentam maneiras transparentes de acessar os dados que armazenam. De modo comum, interligam computadores distintos e organizados, preferencialmente em topologias de cluster. Para os clientes que consomem seus serviços, aparentam ser as mesmas estruturas de dados com as quais estão acostumados a lidar, porém todos os processos envolvidos no armazenamento e recuperação de valores são abstraídos por um contrato de interfaces estabelecido entre ambas as partes. Tais estruturas podem ser classificadas como um ponto único de acesso para um armazenamento distribuído, sem que seus clientes precisem saber lidar com a criação de réplicas dos dados na rede ou políticas de contingência no caso de falhas. Estas características, aliadas aos níveis de abstração oferecidos, podem facilitar drasticamente o desenvolvimento de Sistemas de Informações com base na Internet e suas particularidades.

2.4.1 Tabelas Hash Distribuídas

Inúmeras são as possíveis implementações para Estruturas de Dados Distribuídas. As características de uso, eficiência e versatilidade encontradas em Tabelas *Hash*, as tornam um tipo de Estrututuras de Dados muito popular, se fazendo presentes em inúmeros Sistemas de Informações. Devemos encarar suas características como sabores de uma implementação real do conceito abstrato encontrado em Mapas. Assim como as Tabelas *Hash*, são encontradas diversas implementações similares, também baseadas nos conceitos encontrados em Mapas, como por exemplo Mapas em Árvore, Mapas Enumerados, Mapas *Hash* e Dados Tabulares.

Nos Mapas, encontramos propriedades como um armazenamento baseado em tuplas T, formadas pela combinação de Chave (K) e Valor (V), representada por *K x V*. A recuperação de um determinado valor *V*, previamente armazenado, ocorre com base em sua chave associada *K*. Para estes dois casos, em especial, temos a definição de métodos *put(K, V)* e *get(K)*, que respectivamente, armazenam um valor *V* utilizando como base uma chave *K* e obtém um valor *V* armazenado com base na chave *K*. Temos esquematicamente uma típica definição de contrato de Interface levando-se em conta, por exemplo, a API (*Application Programming Interface*) disponibilizada no JDK (*Java Development Kit*) 7.

Podemos interpretar uma Interface como um contrato. Quando uma aplicação utiliza este contrato para construir seu fluxo, temos que as implementações podem ser livremente substituidas de acordo com o que for mais conveniente no momento. Isso é possível, pois para seguir uma interface, uma classe deve estar de acordo estrito com o que lhe é imposto, como forma de requisitos comportamentais expressos por meio de métodos e seus valores de entrada e saída. Assim, temos uma relação estabelecida entre Interfaces e Classes que as implementam, no sentido de que as primeiras determinam a forma das segundas, sugerindo comportamentos, enquanto que as segundas implementam concretamente o que lhes fora requisitado pelas primeiras. Implementar uma Interface, impõe de modo irrevogável a obrigatoriedade de atender a todos os requisitos (métodos) alí idealizados.

2.5 Inversão de Controle e Injeção de Dependências

A Inversão de Controle (*Inversion of Control*, ou IoC) descreve técnicas empregadas em modalidades de projetos onde determinadas porções do código-fonte são escritas de maneira que possam receber controles de fluxo externos, provenientes muitas vezes de bibliotecas genéricas e/ou reutilizáveis [13, 19].

De modo geral, é comum que seja estabelecido um paralelo entre este estilo de Arquitetura de *Software*, onde são é presente a forte influência dos elementos de IoC, com a clássica Programação Procedimental. Nesta última, verificamos que todo código customizado é responsável por expressar o propósito das chamadas dentro de bibliotecas reutilizáveis, que por sua vez são capazes de lidar com tarefas genéricas. Em contrapartida, as técnicas de IoC fazem com que o código reutilizável seja chamado dentro do código customizado ou de problema específico. Abordagens dessa natureza são realizadas com o intuito de aumentar a modularidade de um *software*, permitindo sua reutilização e extensão de acordo com o estágio em que se encontra seu ciclo de vida.

Existe uma relação entre IoC e o Princípio da Inversão de Dependências (*Dependency Inversion Principle*, ou DIP), no entanto IoC apresenta diferenças conceituais com relação ao DIP que se refere de uma maneira generalista às diretrizes de como realizar a decomposição de um software em módulos distintos. Tal decomposição assume como premissa que construções em Alto-Nível, antes altamente influenciadas diretamente por detalhes encontrados em construções em Baixo-Nível, devem ter sua construção agnóstica às dependências em Baixo-Nível, evitando que sua implementação seja influenciada por detalhes provenientes de particularidades encontradas em suas dependências. Ambas as construções, em Alto e Baixo-Nível, devem depender somente de abstrações contratuais, sendo que estas são estabelecidas de maneira mútua e são compartilhadas entre ambas para fins de comunicação.

2.5.1 Recorrência Comportamental

Alguns comportamentos se repetem em inúmeros cenários durante o cotidiano de desenvolvimento de softwares. É possível exemplificar diversas situações onde as etapas para a construção das soluções são compreendidas por sequências de passos conhecidos. Sendo estes facilmente reproduzíveis, são recorrentes em diversos programas, como por exemplo: A função principal de uma aplicação que realiza chamadas para uma biblioteca especializada em menus. Tal biblioteca, que por sua vez tem a capacidade de exibir na saída padrão uma lista onde constem todas as opções disponíveis sobre a execução de fluxo. Adicionalmente, é retornado o código escolhido que em seguida é processado pela função principal. Este tipo de problema ocorre em interfaces baseadas em modo-texto.

Utilizando uma abordagem influenciada por IoC, teríamos um framework encarregado de tratar eventos comuns, assim como a chegada de informações por meio de um dispositivo de entrada, ou mesmo o encaminhamento do código de retorno para uma interface de saída apropriada. No caso acima, teríamos como entrada os dados recebidos de um teclado, mas o comportamento que é acionado mediante eventos específicos pode ser sobrescrito de acordo com as necessidades.

Associa-se então, um código específico para quando o evento é disparado, como o selecionar de uma opção que é dada ao usuário. Dessa maneira, este comportamento de fluxo é inserido de modo genérico: o *framework* sabe apenas que deve disparar uma chamada ao acionamento de um determinado evento, mas qual chamada será utilizada é um dado que se configura dinamicamente.

A chamada é executada com o código escolhido pois este, obedece a um contrato de Interfaces, o qual dirá sob uma abstração em Alto-Nível o que deve ser feito sob uma nomenclatura genérica. Os detalhes de implementação sobre como a tarefa será executada ficam a encargo do desenvolvedor, que deverá descrever passo a passo como o fluxo se dará de acordo com os recursos que dispõe.

Outro exemplo onde se configura a utilização desta solução é a recorrente troca de drivers de Bancos de Dados. Em um determinado momento, uma aplicação pode estar sujeita a um Sistema Gerenciador de Bancos de Dados (SGBD) e em um futuro pode ser portada para outra tecnologia, de acordo com os interesses da instituição a qual esteja implantada. Para isso, customizações são necessárias mas, com contratos de interface bem estabelecidos, cada implementação de uso pode atender às expectativas se realizadas de maneira adequada.

Deste modo, é possível manter a regra de negócio de uma aplicação e substituir um motor específico. Tendo como base uma Interface bem estabelecida, teríamos diferentes implementações que atenderiam a um único contrato. Isso se resume a receber entradas específicas e fornecer saídas também específicas. Toda parte do *software* onde seja possível se estabelecer um comportamento genérico e substituível, é então definida desta maneira utilizando tais Interfaces.

Para que se possa trocar facilmente as implementações, mantendo o comportamento e confiando nas Interfaces e nos contratos comportamentais que estas estabelecem, utiliza-se uma técnica chamada Injeção de Dependências (*Dependencies Injection*, ou DI) [13, 30]. Esta é difundida principalmente entre linguagens cujo paradigma seja exclusivamente a Programação Orientadas a Objetos (*Objected Oriented Programming*, ou OOP). Sua utilização também pode ocorrer em linguagens cujo paradigma seja Funcional ou Procedimental, mas são requeridas adaptações que se adequem às especificidades encontradas em cada cenário.

Enquanto OOP, as dependências às quais as classes principais estão sujeitas são literalmente injetadas. Esta injeção se dá por meio de Construtores ou Métodos Modificadores, usualmente classificados encontrados em meio ao código com o prefixo “set” seguido do nome da propriedade a qual alteram. Também são encontrados em meio à passagem de parâmetros de métodos ordinários, quando em conjunção à *Factories*. Sendo assim, durante o processo de se instanciar uma Classe para um Objeto, o *framework* que cuida da dinâmica de DI é capaz de ligar propriedades internas das classes Instâncias específicas que atendam as necessidades especificadas pelas Interfaces.

Isso permite que as aplicações sejam estendidas tantas vezes quanto seja possível de acordo com o número de combinação disponível entre Interfaces e suas respectivas Implementações. No momento em que se tiver dispõe de uma quantidade suficiente e necessária de tal diversidade, pode-se simplesmente configurar o software para que em um dado momento trabalhe com um tipo de implementação específica e em outro instante trabalhe com outra implementação distinta, sem que seja necessário reimplementar sua regra de negócio.

Muitos *frameworks* são construídos utilizando-se os princípios de IoC, pois deste modo é permitido que esqueletos generalistas possam ser elaborados e testados de maneira criteriosa, enquanto que permitam o acoplamento de códigos arbitrários em pontos de interesse, pois estes são específicos ao problema que se deseja resolver. A utilização dos conceitos de IoC não se resume à linguagens cuja característica predominantes seja POO, podendo ser aplicada em soluções conhecidas como por exemplo durante a utilização de *CallBacks*, Agendadores, Mensageria, além de servir como base para vários Padrões de Projeto (*Design Patterns* ou DP).

Uma ressalva importante é que para que seja possível a ligação entre implementações específicas como determinadas Classes e contratos que são estabelecidos por meio de Interfaces, é que as Classes e seus métodos obedeçam exatamente todas as exigências que determinam as Interfaces, seja com relação à existência de métodos explicitamente nominados ou ao recebimento de parâmetros com uma dada tipagem, quando for pertinente, ou mesmo a um tipo de retorno que também pode ser exigido em um método especificado.

2.5.2 Técnicas de Implementação

Em se tratando de OOP, é possível que as definições de IoC sejam atingidas utilizando-se algumas técnicas específicas de implementação e muitas destas são compreendidas por alguns Padrões de Projeto (*Design Patterns*), que de modo geral podem ser classificados como problemas bem conhecidos e recorrentes, portanto catalogados e com uma forma bem estabelecida. Dentre os quais se destacam por sua habilidade de implementar a Inversão de Controle:

- **Localizador de Serviços:** Encapsula o processo de se obter um recurso utilizando uma espécie de catálogo de serviços, onde provê para cada tarefa um recurso distinto. Isso é feito por uma forte camada de abstração entre a requisição do recurso e sua obtenção efetiva. O catálogo de serviços se resume a um registro central, conhecido como

Localizador de Serviços (*Service Locator*), onde para cada requisição realizada, espera-se obter a informação necessária para que determinada tarefa seja cumprida. Pode funcionar como um linker em tempo de execução, permitindo que códigos novos sejam integrados às aplicações sem que sejam necessárias re-inicializações. Porém, pode dificultar o debug em casos de erros, pois atua como uma caixa-preta para a aplicação, que de sua lógica nada conhece, apenas passando parâmetros e recebendo resultados pós-processamento.

- **Método *Template*:** Sua definição se dá em meio à implementação de uma Classe que atende em linhas gerais a resolução de um problema. Observando-se determinados métodos que podem eventualmente receber implementações específicas, denomina-se então estes como Abstratos, para que sejam redefinidos em uma Classe Filha, gerada por meio de Herança. Desse modo, o comportamento geral contido em uma classe Pai pode ser mantido, com um fluxo bem estabelecido, sendo somente o ponto de interesse sobrescrito por sua Classe Filha. Isso ocasiona o benefício da não duplicidade de código, uma vez que especificidades são mantidas somente em classes Filhas. A ideia geral é mantida e organizada na Classe Pai, deixando para suas variantes Filhas implementarem os detalhes que farão destas umas diferentes das outras. Outro aspecto a ser levado em conta é que, em detrimento ao Polimorfismo, esta maneira de sobrescrever código implica em um enxerto exato de código em um ponto bem definido, não ocorrendo a sobrescrita de um método já existente e cujo código é funcional por natureza e atende a um problema específico.
- **Estratégia:** Permite que um determinado algoritmo seja escolhido em tempo de execução de acordo com os parâmetros estabelecidos. Tendo sido selecionado, este será executado em um determinado ponto de interesse no código. É possível descrever todos os diferentes comportamentos a serem adotados em termos de famílias de algoritmos. Cada uma destas famílias pode atuar no ponto de interesse onde se deseja obter o comportamento mutável de acordo com as circunstâncias. Sendo assim, antes de sua execução existe uma avaliação sobre o critério que aponta qual família escolher, sendo que todas são plenamente aplicáveis sem que exista a necessidade da adição de um controle de fluxo para cada situação.

Também são formas de implementar a IoC, algumas técnicas de Injeção de Dependências, onde são mapeadas as relações sistêmicas entre Classes e Recursos aos quais dependem. Estes Recursos são implementados de acordo com um contrato de Interfaces bem estabelecido e, por este motivo, podem ser substituídos de acordo com a situação, flexibilizando as construções internas de uma regra de negócios, não ficando esta com seu

fluxo dependente dos motores de processamentos de dados, substituíveis por meio de técnicas como:

- **Injeção por Construtor:** No momento em que se instancia uma Classe para um Objeto, são passadas suas dependências utilizando seu Construtor. Desta maneira a instância se apresenta utilizável já com todos seus motores de processamento de dados internos configurados de acordo com os recursos recebidos.
- **Injeção por Parâmetro:** Pode-se permitir que Métodos ordinários possam modificar quais são as dependências de uma Instância. Deste modo, passando-se Objetos que atendam às especificações impostas pelas Interfaces às quais suas Classes seguem, é possível se atender tanto à tipagem básica quanto ao que se espera em termos de comportamento.
- **Injeção por Métodos Modificadores:** Utilizando Métodos Modificadores, também nominados como Métodos Set, oferecem uma convenção suficiente para que após o processo de instanciação de uma Classe para um Objeto sejam redefinidos tantos atributos quanto se julgue necessário pelas regras de Injeção de Dependências.

3 METODOLOGIA DE DESENVOLVIMENTO

3.1 *MIDDLEWARE* DISTRIBUÍDO

Diante de um Sistema de Informações Legado onde existam necessidades específicas como reparos estruturais ou modernização em termos de infra-estrutura hospedeira, verifica-se que é imprescindível se realizar um levantamento sobre quais são os principais fatores de impacto que serão ressaltados mediante a execução de mudanças arquiteturais. É preciso que seja determinada a extensão aproximada da base de usuários afetados e o nível de criticidade envolvido na interrupção dos serviços provisionados.

Ao prospectar um LIS como Corpo de Prova, foi constada a existência de particularidades inerentes ao segmento de mercado que faz parte. Em uma ponta da solução, sua base de usuários é representada por mais de uma dezena de milhares de dispositivos móveis. Já na outra ponta, sua base de usuários é composta por centenas de operadores humanos. O tráfego de informações é ocorre em sentido bidirecional entre ambas as pontas; suas operações estão sujeitas a um regime 24x7 (atividades compreendidas por um período de 24 horas ao dia, durante cada um dos 7 dias contidos em uma semana); com consolidação de dados em tempo real para análises e tomada decisão; hospedado em uma infra-estrutura suscetível a falhas devido a um ferramental físico em estágio avançado de depreciação.

Tendo como referencial um LIS de missão crítica como este, medidas compatíveis com o cenário foram adotadas, de modo que não seriam toleradas interrupções de serviço ou quebras de compatibilidade entre versões de serviços envolvidos. Sendo a elaboração de estratégias para a migração de Sistemas de Informações Legados o principal objeto de estudo ao qual se atém este trabalho, a adoção do referido LIS como Corpo de Prova revelou uma oportunidade para que um Modelo de Migração específico fosse criado e validado.

3.1.1 Alta Disponibilidade

Em termos de melhorias executadas, o fator primordial que guiou os demais objetivos: A necessidade de se implementar uma política de contingência em caso de falhas. Essa categoria de problemas nos remete às técnicas de clusterização difundidas entre sistemas que necessitam de Alta-Disponibilidade (*High-Avaiability* ou HA).

Para que um serviço possa se tornar um *cluster* em HA, é necessário que a solução reaja de maneira apropriada mediante eventuais falhas. Essa característica, uma vez cumprida, classifica o sistema a qual faz parte como Tolerante a Falhas [6, 14]. Uma solução assim possui a habilidade de manter-se operante em situações adversas, que normalmente causariam sua indisponibilidade de operações. Para que isso seja possível é necessário entender o conceito de serviços *Stateless* e *Stateful*.

Serviços *Stateless* possuem um fluxo de trabalho completamente orientado às requisições que recebem, ou seja, todos os dados que necessitam para processar uma tarefa lhes é atribuído de uma única vez e necessariamente a troca de mensagens neste contexto acaba quando a tarefa em questão é completamente processada. Isso faz com que o fluxo de seu consumidor lide com tarefas atômicas e, portanto, independentes entre si. Pode ser necessário realizar várias chamadas, armazenar seus resultados temporariamente e, então, realizar um processamento sobre os mesmos, com intuito de completar alguma tarefa-macro que seja definida pela regra de negócio. Não ocorre, por exemplo, o fato de uma tarefa necessitar que múltiplas requisições carregando dados distintos sejam realizadas em uma ordem específica para que o processamento geral obtenha sucesso.

Em Sistemas *Stateless*, o estado de uma requisição existe somente no contexto em que o processamento de uma tarefa está em execução. Após o término de uma tarefa, não ficam pendências, as quais requerem mais dados para então resultarem em um dado processamento e conseqüente retorno de status ou valor. Para exemplificar soluções que atendam à estas características, podemos levar em conta Servidores Web como Apache e Nginx. Estes se comportam deste modo não porque convencionou-se para ambos esse tipo de arquitetura, mas pelo fato de que, por padrão, servem conteúdo HTML utilizando HTTP(S) como protocolo. Tal protocolo foi concebido para funcionar de modo *Stateless* e, assim, o tipo de problema que um serviço se propõe a resolver irá direcionar como sua arquitetura será projetada.

De modo contrário, Sistemas *Stateful* são capazes de guardar os estados em que se encontram suas comunicações até que todo o fluxo de trabalho esteja completo, independente da quantidade de requisições envolvidas, ordem de mensagens adotadas ou dados transmitidos bidirecionalmente. Para executar ações específicas, um cliente pode precisar enviar uma série de informações para um servidor, mas estas podem não ser enviadas de uma vez por conta de uma especificidade da regra de negócio, como quando a consolidação de informações fornecidas influencia no conteúdo de informações que serão

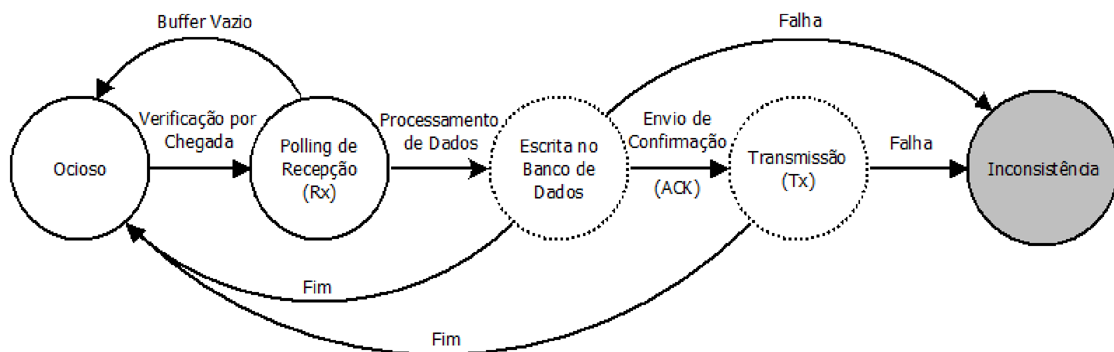
transmitidas em momentos futuros após o processamento e análise de resultados de suas predecessoras.

Existem categorias de Firewall, como o *Packet Filter* (PF), que operam em modo *Stateful*. Nestes, são realizadas inspeções nos pacotes que chegam, ao mesmo tempo em é mantido o controle de todas as conexões ativas, rejeitando pacotes cujo conteúdo não seja autorizado ou que cuja conexão não esteja compreendida entre as conexões ativas, às quais é capaz de gerenciar, mantendo em memória seus estados até que seja terminado o fluxo de troca de informações.

3.1.2 Pontos Críticos

Tendo como partida o LIS assim denominado como Corpo de Prova e partindo-se da premissa de que este devesse integrar uma infra-estrutura mais moderna, sem que deixasse de atender ao cotidiano de suas operações, iniciou-se uma busca por Pontos Críticos. Trata-se de ocorrências na base de código, capazes de inviabilizar o plano de transição, se negligenciados ou se não forem tratados adequadamente. A Figura 3.1 retrata um diagrama de Estados de Máquina onde estão relacionadas as possíveis inconsistências devido a eventuais falhas em fluxos onde são presentes Pontos Críticos [15].

Figura 3.1 – Possíveis inconsistências em caso de falhas no fluxo.



O fluxo básico é compreendido pelo estado Ocioso, onde nenhuma informação chegou. Periodicamente é disparada uma escuta, caracterizada pela Verificação de Entradas, o que nos leva ao estado de Polling Rx (*Reception*), onde caso não existam dados volta-se ao estado anterior e, em caso contrário, os dados recebidos serão processados. O próximo estágio envolve a Escrita em BD e, dependendo da operação, pode terminar o fluxo, fazendo com que se retorne ao estado inicial. Em caso de necessidade de devolução de uma

resposta o próximo estado é o Tx (*Transmission*), onde as informações de retorno são devolvidas para o Dispositivo Móvel.

Se por um acaso as operações que devessem ser escritas no BD não ocorrerem a tempo, mediante uma falha sistêmica, teremos uma inconsistência causada por possíveis mudanças de estados nos dispositivos ou disparo de processamentos de dados que não foram devidamente consolidados em BD. Também ocorrerá inconsistência se uma situação análoga atingir as mensagens que devam ser retornadas para os Dispositivos Móveis mas que não forem entregues devido à quebras inesperadas de fluxo.

Ao se planejar a evolução do LIS em questão estabeleceu-se, de modo consensual, como sendo desejável que este pudesse continuar suas operações em caso de falhas, confiando em meios de HA para que isso fosse possível. Para que a transição fosse possível sem a interrupção das operações, foi adotada uma abordagem gradual, onde a demanda corrente de operações pudesse ser transferida entre um ambiente de produção legado e seu sucessor, com caráter distribuído.

Sabendo que estas decisões são capazes de limitar o alcance das estratégias empregadas na solução, foram levados em conta fatores que poderiam causar a quebra nas operações, em caso de falha. As incompatibilidades que foram encontradas têm relação com características *Stateful* empregadas no serviço. O Corpo de Prova possuía um extenso mapeamento de propriedades encontradas em cada dispositivo com que interagia. Estas eram suscetíveis a mudanças em seus valores durante as etapas envolvidas nos processos de comunicação. Além disso, também foram encontrados buffers responsáveis pelo processamento de consultas SQL (*Structured Query Language*), que eram disparadas contra o Banco de Dados em que o LIS se conectava. Com a interrupção de operações, estas estruturas de dados poderiam sofrer eventuais corrupções e necessitariam de reparos manuais em sua fonte de dados principal.

Para contornar as limitações encontradas, Estruturas de Dados Distribuídas (*Distributed Data Structures* ou DDS) foram utilizadas para lidar com o encargo que os Pontos Críticos se responsabilizavam. Deste modo, bastaria apenas que os dados estivessem replicados em uma quantidade suficiente de nós para que a redundância fosse atingida. Não obstante, para se atingir a escalabilidade horizontal da solução, foi constatado que com o aumento na demanda de operações, haveria a necessidade do processamento de mais requisições. Com isso, seria gerada uma demanda maior sobre a camada de BD, que antes de ser acessada interagia com o buffer de operações, classificado como Ponto Crítico.

Para manter a regra de negócio nos *daemons*, utilizou-se de abstrações quanto às operações e estruturas de dados. Desse modo, o fluxo de operações pôde ser preservado enquanto que ajustes arquiteturais profundos puderam ser realizados, como por exemplo, a distribuição dos dados ao longo de múltiplos nós. As abstrações utilizadas se resumem a estruturas como Filas e Mapas-*Hash*, podendo estes ser aplicados diretamente onde os dados forem compatíveis.

No caso do armazenamento de propriedades contidas em cada dispositivo, Mapas-Hash atendem aos requisitos básicos, pois assim como grupos de elementos de mesmo gênero e que contém uma série de propriedades, poderiam ser utilizados para armazenar uma série de objetos cuja identificação fosse um ID único. Para tornar esta abordagem distribuída, foi então empregado um framework chamado *Hazelcast*, que fornece abstrações suficientes e necessárias para se adaptar o funcionamento do LIS em questão.

No que se refere à utilização da camada de BD, tendo um buffer intermediário de SQLs, este sofreu uma adaptação para que pudesse se tornar distribuído. O modo como era organizado originalmente, fazia com que todas as consultas fossem concatenadas sequencialmente para uma execução mediante a um determinado intervalo de tempo. Para que pudesse escalar conforme a quantidade de nós, foram idealizados múltiplos estágios de processamento distintos, onde cada um deles pudesse se especializar em um determinado tipo de operações.

Assim, foi necessário que se aplicasse duas abordagens distintas, baseadas na natureza distribuída para eliminar os Pontos Críticos na evolução da solução vigente. O armazenamento distribuído de estruturas elementares, com Interfaces padronizadas e bem definidas pode ser feito para o controle global de propriedades do LIS, mas o fluxo de dados envolvidos na mudança do estado durável do BD precisou ser cuidadosamente extraído para uma camada especializada no gerenciamento e execução de tarefas SQL. Este último passo necessitou da elaboração de uma arquitetura para um *Middleware*-Distribuído (MD), utilizado como fator intermediário entre o LIS e o BD, mas capaz de reter em nós distintos os dados antes considerados suscetíveis a falhas. De modo similar ao acesso aos dados sobre dispositivos, a interação com o MD precisou ser abstraída por meio de contratos de Interface bem estabelecidos e abstrações capazes de manter a regra de negócio ao mesmo passo que oferecem a eliminação dos Pontos Críticos.

3.1.3 Arquitetura

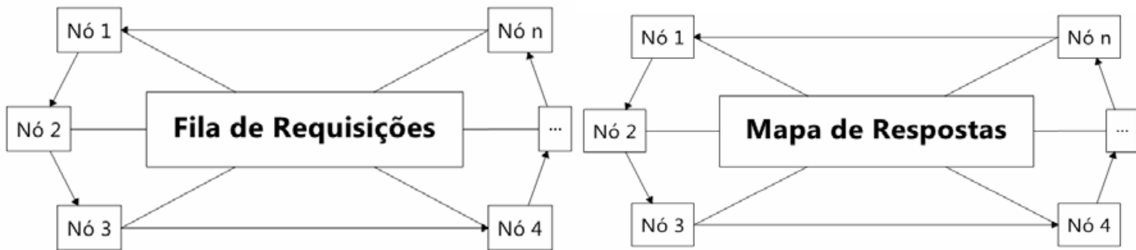
Para que o LIS pudesse evoluir sem problemas de retro-compatibilidade, foram adotadas medidas capazes de manter a regra de negócio em sua totalidade. Uma das principais abordagens foi a criação de um cliente que abstraísse toda a interação que ocorresse entre o LIS e o MD. Desse modo, complementou-se a busca por Pontos Críticos, fazendo com que nos locais identificados em que o acesso aos dados era feito diretamente, nas estruturas mapeadas, fosse então realizado por intermédio de um cliente.

Ter utilizado um cliente intermediário, capaz de abstrair a comunicação distribuída trouxe a vantagem de que dentro do LIS, o fluxo algorítmico se manteve o mesmo, não sendo necessário cuidar de detalhes como conexão entre nós distribuídos e de tarefas auxiliares que frequentemente são requeridas. Sendo assim, foi possível trabalhar em uma arquitetura onde múltiplas instâncias do MD tornaram-se capazes de operar com o mesmo objetivo, de modo paralelo e em diferentes localidades.

Cada nó do MD tornou-se então, responsável por uma porção dos dados totais que a solução armazenava em seus Pontos-Críticos. Estabelecendo-se uma quantidade n de nós, ligados entre si em forma de lista circular no que diz respeito aos dados armazenados, cada um destes nós torna-se responsável de maneira ativa por $1/n$ do montante dos dados presentes na rede. Além disso, cada nó carrega $1/n$ dos dados passivamente, na forma de backup dos exatos $1/n$ cujo seu predecessor é responsável e, por este motivo, os carrega ativamente.

Esta organização torna possível que eventuais falhas na solução sejam toleráveis e, por isso, o LIS consiga dar continuidade às suas operações sem que existam interrupções. Se um dos nós que compõe a rede falhar, os outros nós conseguem assumir seu papel, pois se comunicam em forma de Rede P2P. Sua organização como lista circular ocorre somente com relação aos dados que carregam, como uma estratégia de contingência, mas sendo uma Rede P2P os nós são capazes de alcançar uns aos outros pela topologia que implementa seu overlay de rede. A Figura 3.2 relaciona o esquema circular onde os Nós são uns conectados aos outros, mas com o objetivo de compartilharem as DDSs denominadas Fila de Requisições e Mapa de Respostas [14].

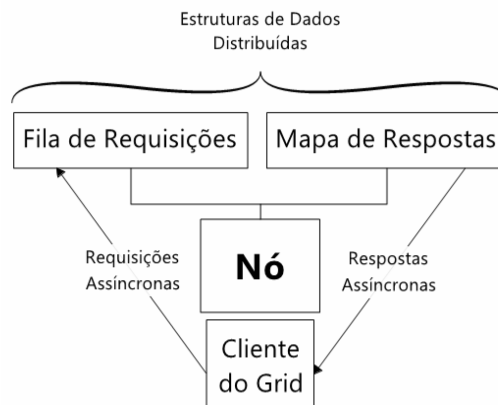
Figura 3.2 – Lista-circular de nós que interconecta DDSs.



Pelas características encontradas em Redes P2P, as DDS que são carregadas por cada nó do MD, têm basicamente o comportamento de Filas e Mapas-*Hash* Distribuídos. Estas DDSs provêem à solução, recursos suficientes e necessários para particionar os dados entre os nós, distribuir estes de maneira adequada, atualizar as informações e replicá-las entre todos os que compõem a rede, realizar buscas dentre os nós ativos sabendo que lá se encontram os dados procurados e não em outros nós espalhados pela rede e etc.

A Figura 3.3 relaciona uma instância *Stand-Alone* denominada como Nó. Cada unidade desta, pode se relacionar com outras demais, que pertencem à mesma rede [14]. É possível que estas se encontrem automaticamente por meio de um *Multi-Cast* disparado sobre o *overlay*. Assim, se existirem mais instâncias na LAN (*Local Area Network*) ou WAN (*Wide Area Network*), estas irão se reconhecer e compartilhar suas estrutudas de dados. Uma Fila de Requisições e um Mapa de Respostas é instanciado em cada nó e a Rede P2P se encarrega de cuidar da conectividade entre cada um dos pontos, lembrando que tais estruturas de dados têm caráter distribuído e, portanto, seus dados são particionados para fins de Alta Disponibilidade. As requisições e respostas ocorrem por padrão de modo assíncrono e, para isso, é necessário que utilizem um Cliente para o Grid como intermediário das operações.

Figura 3.3 – Instância de um nó e relacionamento com DDSs.



Com a abstração às DDS sendo realizada pelo Cliente, toda a complexidade é absorvida pelo mesmo e não pela aplicação que o utiliza, no caso o LIS. Suas operações se resumem a dispararem Requisições que denotam consultas ou escritas em BD e por fim receber o resultado deste processamento em forma de Respostas.

Vale ressaltar neste ponto que o propósito do *Middleware* é garantir que todas as operações de BD em curso sejam retidas, mesmo em caso de falhas na solução. Isso não era garantido antes, pois em caso de problemas que interrompam o serviço, todas as operações enfileiradas para execução seriam perdidas e isso eventualmente acarretava uma série de manutenções para restaurar o estado do BD para pontos onde não existissem inconsistências nos processos envolvidos no negócio da solução como um todo. É natural que em um primeiro momento se confunda o papel desta nova camada com o de uma Fila de Mensagens ou *Driver* para BD. Não obstante, estas duas categorias de solução são utilizadas para compor o MD, adicionando-se o fato de que todas as operações enfileiradas para execução, a exemplo de uma fila de mensagens, continuarão vivas em memória após interrupções de operação abruptas pois estão distribuídas ao longo de múltiplos nós em uma Rede P2P e, no momento certo, serão executadas por intermédio de um *Driver* de BD, como era de se esperar.

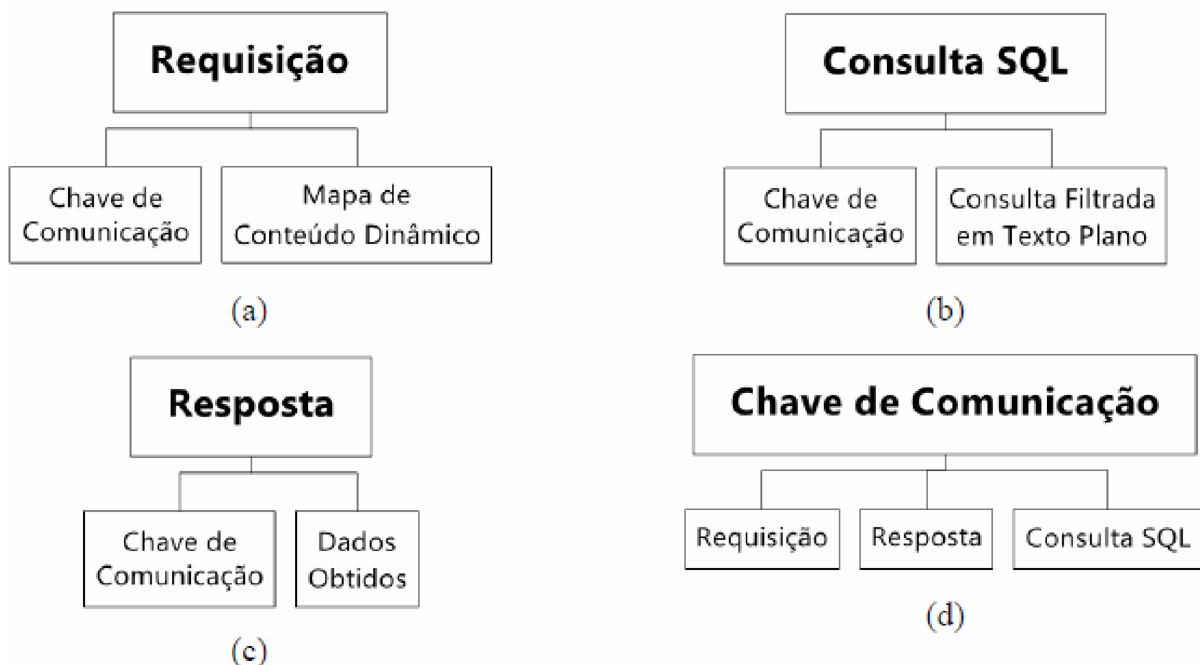
Como Requisições para operações em BD muitas vezes retornam Respostas, uma Estrutura de Dados especial também foi garantida para este fim. Sua natureza não é a de Fila, a exemplo das Requisições, mas sim um Mapa-*Hash* que armazena as Respostas. É possível se estabelecer uma ligação entre uma Requisição e uma Resposta quando o canal de comunicação entre as mesmas permanece aberto de modo bloqueante, ou seja, completamente serial. Mas para ser capaz de operar em larga escala, é necessário que seja implementada uma política de paralelismo para a execução de operações. Neste ponto, é possível que um trabalho seja submetido para execução posterior, liberar o fluxo da aplicação para sua execução normal e em um ponto futuro se recuperar o valor de retorno em forma de Resposta. Para isso é necessário que ambas entidades, Requisição e Resposta, sejam relacionadas a um único ID global.

Tendo estabelecido o ID como propriedade de cada Requisição, após esta ter sido completada, sua resposta é colocada no Mapa-*Hash* relacionando como Chave o ID da Requisição que originou o processamento obtido e sintetizado em forma de Resposta como Valor da Tupla Chave x Valor ao qual é composto o Mapa-*Hash* em questão. Diversas Respostas vão alimentando este Mapa conforme vão sendo executadas as Requisições e na medida em que seus valores são requeridos pela aplicação, as Respostas são recuperadas e

então eliminadas da memória, caracterizando também este fluxo como parte de um problema clássico de Produtores e Consumidores. Para fins de notação, o ID que relaciona Requisição com Resposta foi denominado internamente como Chave de Comunicação, por representar de maneira mais apropriada seu papel perante a ótica de interações entre Clientes e *Grid*.

Esquematicamente, temos a Figura 3.4, que mostra as seguintes estruturas: *(a)* Requisição, com sua Chave de Comunicação e um Mapa de Valores, que são utilizados para preencher sua intenção de execução no BD, como parâmetros; *(b)* a Consulta SQL com a Chave de Comunicações, idêntica à Requisição que lhe deu origem, para fins de relacionamento e sua representação na forma de Consulta em Texto-Plano Filtrado, que contém a instrução SQL que será submetida ao BD, tendo sido preenchida com os Valores advindos do Mapa de Valores da Requisição; *(c)* a Resposta, com a Chave de Comunicação, que relaciona a Requisição de Origem e a Consulta SQL que gerou os Dados Obtidos como resultado de processamento; *(d)* a Chave de Comunicação, sendo uma entidade numérica, comumente referida como um ID e que relaciona as três entidades anteriores, por se apresentar no interior de todas elas como seu atributo [14].

Figura 3.4 – Entidades básicas e seu relacionamento.

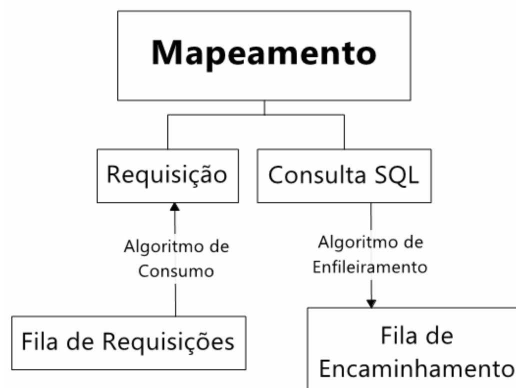


Como existem distintas operações de BD possíveis de se realizar, cada uma conta com particularidades nem sempre abrangidas por chamadas de execução, usualmente disponibilizadas por métodos integrados em *Drivers* de BD. Obviamente os *Drivers* de BD

são importantes para o acesso aos dados, mas as implementações onde estes são invocados podem se tornarem cada vez mais apropriadas para cada tipo de operações executadas quanto se desejar. Por esta razão, o fluxo de execução foi dividido de acordo com o tipo de operação em BD. Tendo um esqueleto em comum a ser seguido, conforme determina um Contrato de Interface, cada fluxo operacional é especializado levando-se em conta o tipo de requisição que irá realizar ao BD, por intermédio de seu *Driver*.

Deste modo, as Requisições passam por uma triagem antes de ser executadas. Esse processo as coloca em filas distintas para execução. O modo como serão executadas depende do tipo de operação e isso pode ser compreendido por execuções seqüenciais, em lote por concatenação ou em lote gerenciadas por *Driver* de BD. Fazendo-se esta triagem, pode-se obter melhores resultados quanto ao desempenho. O benefício dessa prática se reflete em termos de escalabilidade da solução como um todo, aumentando a vazão do processamento de dados. A Figura 3.5 mostra como ocorre o Mapeamento das Requisições, este processo é referente ao primeiro contato que uma Requisição têm com sua Fila [14]. Uma vez presente na Fila de Requisições, cada uma delas é mapeada para um tipo de Operação no BD, sendo então encaminhadas para o próximo estágio.

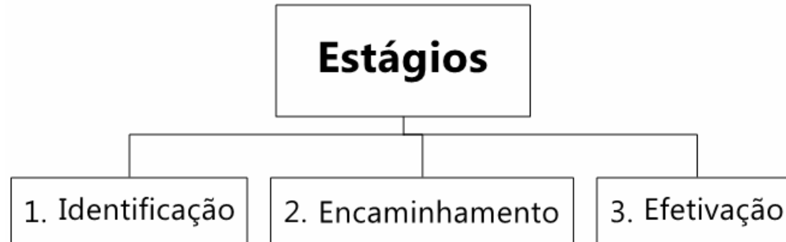
Figura 3.5 – Mapeamento de requisições para consultas SQL.



A Figura 3.6 mostra como são organizados os estágios de processamento-macro. O primeiro deles é chamado de Identificação, onde cada Requisição é mapeada de acordo com sua natureza e então é repassada para o segundo estágio, o Encaminhamento, que cuida de inserir em filas categorizadas as requisições já triadas. Quando for oportuno, as requisições que já foram devidamente encaminhadas e estão na espera para ser consolidadas serão capturadas e processadas pelo Efetivador, que cuidará de processar cada uma de acordo com sua especificidade. Por exemplo, uma Consulta exige um tratamento diferente de um

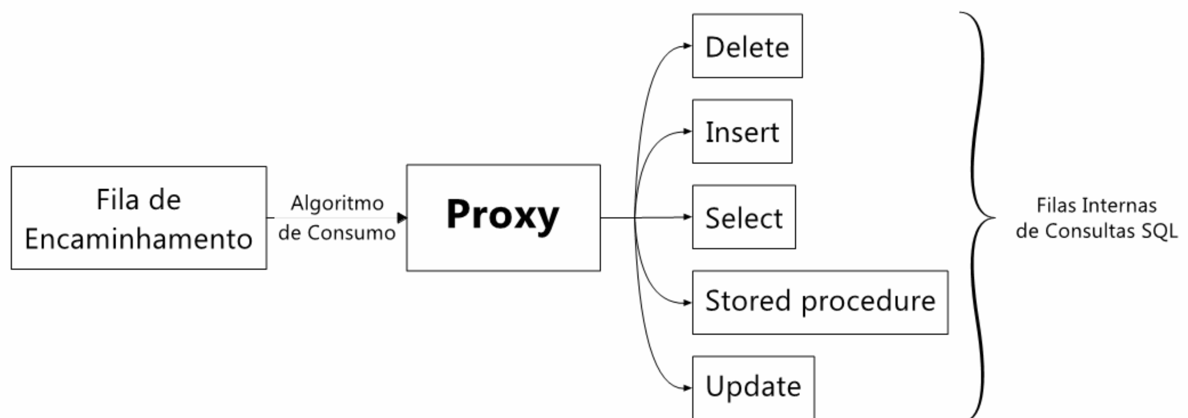
Procedimento Armazenado, que por sua vez possui um fluxo de exceções distinto de Atualizações, que possuem códigos de retorno diferentes de Remoções e Inserções [14].

Figura 3.6 – Os estágios de processamento do *middleware* distribuído.



Diferentes filas internas ao MD cuidam de Requisições de naturezas distintas. Solicitações por Consultas, Inserções, Atualizações, Procedimentos Armazenados e Remoções contém características que tornam distintos os fluxos de tratamento para cada uma destas situações. A figura 3.7 retrata como ocorre o encaminhamento de Consultas SQL para a esfera de fluxos de execução os algoritmos especializados em cada uma destas categorias de operações em BD [14], isso faz com que o processo de tratamento de exceções se torne mais limpo e a manutenção do código menos suscetível a falhas, visto que pode ser feita atomicamente.

Figura 3.7 – Encaminhamento de consultas por especialidade.



Para efetivar as informações no BD, um processo final busca em cada fila a ocorrência de novas Requisições para tratar, sabendo do tipo de fila, escolhe o algoritmo mais apropriado para tratar a requisição e, tendo uma resposta do BD, quando pertinente, trata de devolver a Resposta no Mapa-*Hash* de Respostas, esse processo é descrito pela Figura 3.8 [14]. Uma vez identificadas e classificadas por natureza, as Requisições são efetivadas no BD

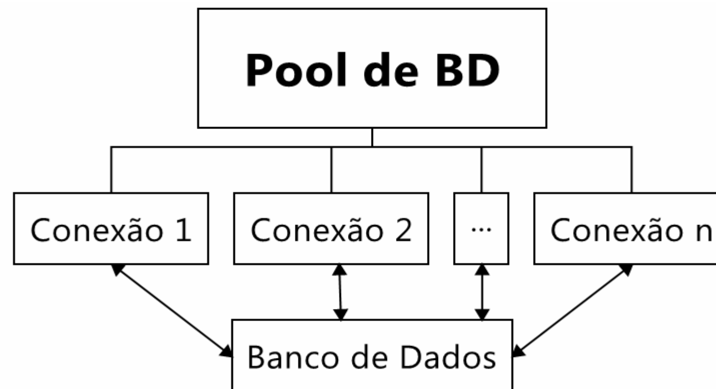
com o algoritmo correto. As conexões com o BD são obtidas por meio de um Pool de Conexões e as Respostas alimentam um Mapa-Hash de Respostas Distribuído. Para obter uma conexão com o BD é utilizada uma técnica de Pool de recursos, onde múltiplas conexões são mantidas ativas e conforme a demanda por execução de operações chega, estas são enfileiradas para ocorrerem mediante a disponibilidade de recursos.

Figura 3.8 – Efetivação de informações no BD e envio de respostas.

Essa abordagem torna o BD imune a sobrecargas causadas por picos de operações. Para isso, estipula-se um valor ideal de conexões que devem ser mantidas com o BD e nenhuma conexão a mais é aberta, utilizando-se somente aquele conjunto de conexões designadas com essa finalidade, daí a definição de *Pool* de conexões: é exatamente este agrupamento destas, que faz do *Pool* uma coleção de recursos que só podem ser utilizados caso estejam livres. No caso de todos os recursos estarem sendo utilizados, é implementada uma lógica de bloqueio no fluxo de operações até que se tenha uma conexão livre capaz de atender a demanda. Também pode se adotar uma lógica não bloqueante, que tenta mais tarde ou devolve uma excessão informando que não há recursos disponíveis, mas esta última não foi utilizada devido à natureza do problema onde é estritamente necessário se esperar para que a execução da Requisição seja feita, uma vez que o fluxo de execução tenha atingido este estágio.

Seja o Pool conectado a um Sistema Gerenciador de Banco de Dados Relacional (*Relational DataBase Management System* ou RDBMS), este gerencia e agrupa suas conexões como se fossem um *Array* destas e, as fornece conforme existe disponibilidade de utilização, esquematicamente está representado pela Figura 3.9 [14] e pode comportar tantas Conexões quantas forem suportadas pelo *Hardware* hospedeiro, devendo este parâmetro sempre estar atrelado a um número obtido por meio de um *Tunning* apropriado.

Figura 3.9 – Pool de conexões com BD.



Tendo em vista este cenário, o cliente que implementa acesso aos Middlewares Distribuídos, ou seja a todos os nós que compõe o *Grid*, oferece opções para a *(i)* submissão de Requisições e *(ii)* recuperação de Respostas, com diferentes versões de algorítimos:

- **Com Espera Bloqueante:** (i) Tenta inserir uma Requisição, bloqueando o fluxo de execução até que surja espaço na Fila de Requisições, caso não exista no momento da chamada. (ii) Tenta recuperar uma Resposta, bloqueando o fluxo de execução até que a mesma apareça no Mapa-*Hash* de Respostas, caso não exista no momento da chamada.
- **Sem Espera Bloqueante:** (i) Tenta inserir uma Requisição, retornando imediatamente o fluxo de execução, informando caso isso não tenha sido possível por falta de espaço na Fila de Requisições. (ii) Tenta recuperar uma Resposta, retornando imediatamente o fluxo de execução, informando caso isso não tenha sido possível encontrá-la no Mapa-*Hash* de Respostas pela sua ausência no momento da chamada.
- **Com Espera Bloqueante Guiada Por *Timeout*:** Faz o mesmo que (i) e (ii) fazem nas versões "Com Espera Bloqueante", desistindo do processo caso o tempo de espera seja atingido por um valor estipulado como *Timeout*.
- **Com Espera Bloqueante Guiada Por *Timeout* e Máximo de Tentativas:** Faz o mesmo que (i) e (ii) fazem nas versões "Com Espera Bloqueante Guiada Por Timeout", repetindo o processo de acordo com um Número Máximo de Tentativas estipulado, antes de desistir do processo (levando em conta o *Timeout* para cada tentativa falha).

É importante notar que as Requisições não são compostas por texto-plano SQL, sendo mapeamentos por intenção de operações. Isso significa que uma inserção, será representada como uma entidade derivada de uma classe *Insert*, contendo apenas os dados que

compõe aquele determinado tipo de *Insert*, como por exemplo, um *InsertDadosX* e seus dados auxiliares sendo injetados para dentro deste como suas dependências.

Uma Resposta para uma Requisição, também não é composta por texto-plano e sim por uma classe análoga, mas de natureza voltada para resposta. Ocorre então um mapeamento 1x1 de Requisição e Resposta. Esse problema é resolvido com um catálogo de meta-dados, capaz de cruzar uma classe Requisição de determinado tipo com outra classe de Resposta de tipo compatível. Essa complexidade é resolvida com auxílio de técnicas de Reflexão (*Reflection*) características de linguagens Orientadas a Objetos, especialmente Java como é o caso abordado por este trabalho.

O cruzamento de algoritmos que devem ser utilizados para resolver determinadas chamadas de BD, retiradas de Filas específicas é realizado com a técnica de Injeção de Dependências, que implementa o Princípio de Inversão de Controle. Esse setup torna as instâncias dos *Middlewares* Distribuídos altamente flexíveis quanto suas construções arquiteturais internas e faz com que sejam capazes de atender a demandas específicas mediante configurações especializadas, sem que seja necessário alterar sua própria regra de negócio.

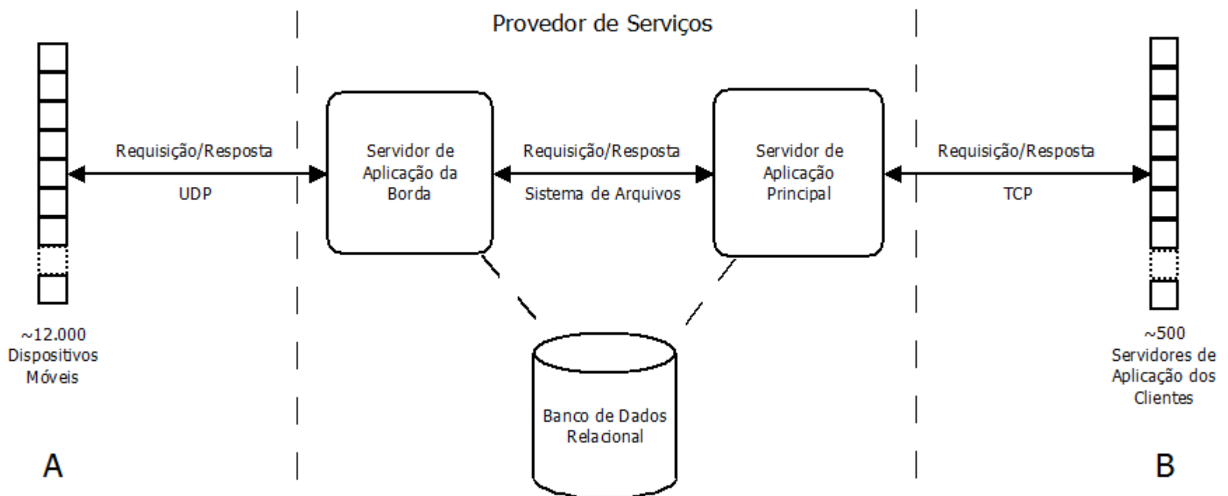
3.2 ESTUDO DE CASO

A exemplo do LIS escolhido como Corpo de Prova, certas particularidades relacionadas a seu ambiente foram encontradas. Tais detalhes guiaram a concepção deste estudo tal qual como ele é e necessitaram muitas vezes de abordagens específicas para que se pudesse concretizar os objetivos básicos de *(i)* Trazer a solução para uma infra-estrutura mais moderna e *(ii)* Tornar a aplicação escolhida tolerante a falhas.

O ambiente onde inicialmente se encontrava o LIS, era composto por um hardware dedicado, localizado em uma única célula física, onde eram hospedados sistemas heterogêneos que se comunicavam por meio de protocolos proprietários. Cerca de 12.000 dispositivos móveis compunham uma ponta da solução, que enviava dados sobre telemetria e geolocalização coletados por sensores destes dispositivos para o LIS em questão. Existia a comunicação com mais sistemas backend e até então não se cogitou a mudança de tal comportamento, nem destes outros serviços. Dados duráveis eram persistidos em Bancos de Dados Relacionais e cada um dos clientes dos serviços, integravam com a solução da Organização para recuperar relatórios e enviar comandos para os atuadores acoplados nos dispositivos móveis.

Cerca de pelo menos uma mensagem por dispositivo era enviada a cada 90 segundos. O protocolo de comunicação era totalmente binário entre os dispositivos e o LIS, pois muitas vezes as informações trocadas referiam-se somente à troca de códigos de acionamento de dispositivos e seu retorno de status. O LIS também era denominado como um Servidor de Aplicação de Borda (*Edge Aplication Server* ou EAS), por ser o primeiro serviço exposto à rede, sendo a camada mais externa da solução. A Figura 3.10 mostra toda a interação ocorrida por toda a solução, verificam-se as integrações existentes entre o Corpo de Prova (LIS ou EAS em questão) [15].

Figura 3.10 – Solução onde é encontrado o corpo de prova.

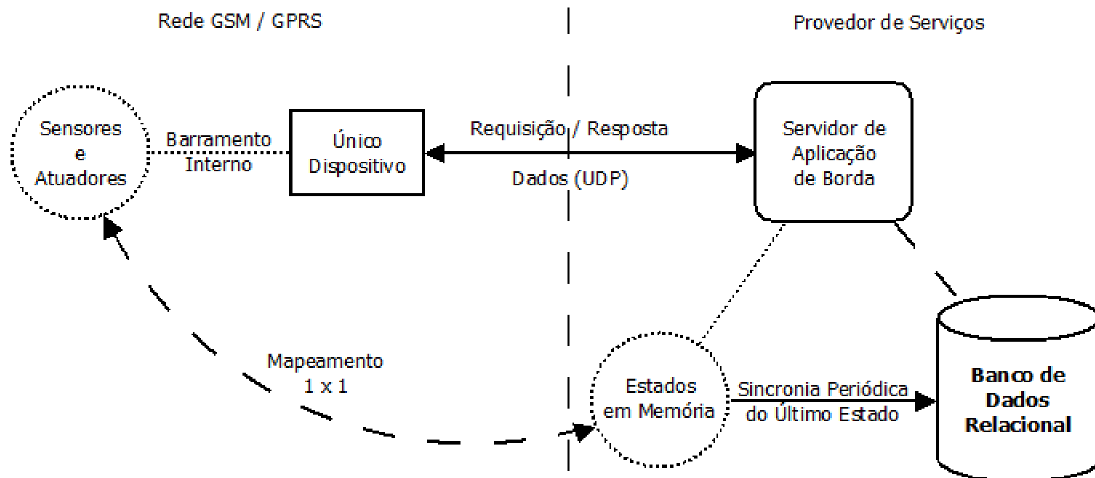


O protocolo binário de comunicação foi implementado sobre a camada de transporte UDP (*User Datagram Protocol*), pois os dispositivos foram projetados para utilizar as redes GSM (*Global Systems for Mobile Communications*) ou GPRS (*General Packet Radio Service*), que não se comportaram bem quanto à manutenção de conectividade quando combinadas com a camada de transporte TCP (*Transmission Control Protocol*).

Um dispositivo móvel, conta com um barramento interno capaz de se comunicar com uma série de sensores, que lhe fornecem dados sobre telemetria e geoposicionamento veicular, e também atuadores, que permitem executar ações diretamente nos veículos, como o acionamento de alarmes, desligamento de motor ou mesmo impor o travamento de baú, quando existir um. As informações capturadas são transmitidas para o EAS e deste, são recebidas ordens de atuação. Existe um mapeamento completo dentro da memória do EAS, referente aos estados de cada dispositivo e, caso não mudem, é persistido em BD num intervalo de 10 minutos.

A Figura 3.11 ilustra como o Provedor de Serviços se relaciona com um único Dispositivo Móvel. É possível se observar a correspondência *1x1* entre os dados coletados de Sensores e os Estados Internos de Memória [15]. Pode-se verificar a eventual sincronia desses dados com o BD, quando necessário e a troca de dados por meio de um protocolo proprietário e binário, em UDP.

Figura 3.11 – Relacionamento entre provedor de serviços e um único dispositivo móvel.

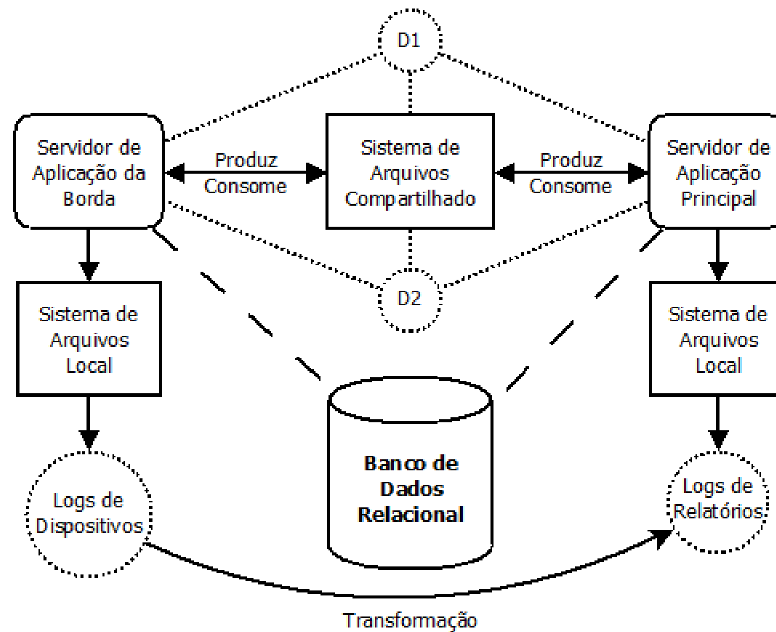


A integração entre o EAS e um Servidor de Aplicação Principal (*Main Application Server* ou MAS), é realizada pela troca de arquivos em texto-plano em Sistemas de Arquivos Compartilhados (*Shared File Systems* ou SFS). Esse comportamento foi mantido ao longo dos anos, por se tratar historicamente da fusão entre sistemas advindos de Organizações distintas. Os estados em memória contidos no EAS podem ser modificados a qualquer momento, se necessário, seja esta requisição vinda de seus clientes finais ou por conta de algum fenômeno físico ocorrido nos dispositivos móveis e capturado por seus sensores.

A Figura 3.12 ilustra esse processo de troca de informações entre sistemas, utilizando os arquivos de integração sobre o SFS. Nota-se uma relação bidirecional de Produtor e Consumidor, que é estabelecida entre ambos os sistemas. D1 e D2 são diretórios de troca, utilizados pelos sistemas, onde BAS produz conteúdo em D1 e MAS consome o conteúdo produzido em D1, ao passo que MAS produz conteúdo em D2 e BAS consome o conteúdo produzido em D2. Os dois sistemas contam com pontos de montagem locais, destinados à estágios de processamento independentes: no diretório local de BAS são produzidos arquivos de *Log*, com todos os registros das atividades e informações sobre estados relevantes nos Dispositivos Móveis [15]. Estes arquivos são processados e suas

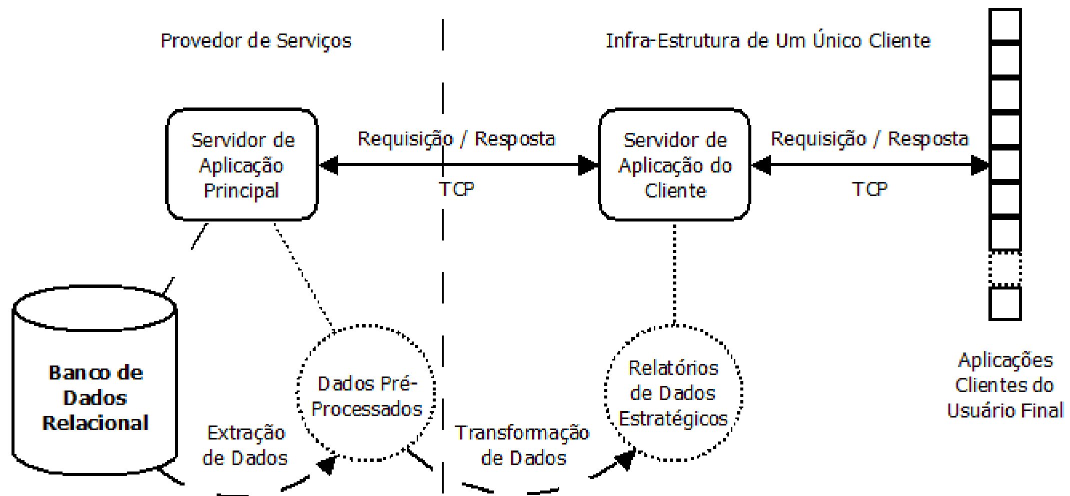
informações geram arquivos de integração que vão para MAS, que os processa e gera Logs de Relatórios em seu diretório local, que serão posteriormente consumidos pelos Servidores de Aplicação localizados nas infra-estruturas dos Clientes.

Figura 3.12 – Relacionamento entre BAS e MAS.



Os clientes finais consomem os recursos de informações do MAS por meio de conexões TCP onde as mensagens trafegam encriptadas. Cada um dos aproximados 500 clientes precisa manter necessariamente em sua infra-estrutura um Servidor de Aplicação para o Cliente (*Customer Aplication Server* ou CAS). Um CAS em questão recebe relatórios estratégicos pré-processados e também é capaz de disparar ordens para seus veículos, por meio desse fluxo de mensagens. A Figura 3.13 exemplifica a interação entre o Provedor de Serviços e um único Cliente, enquanto organização, de modo que em sua infra-estrutura, o MAS se comunique com o CAS e suas inúmeras Aplicações Front-End possam servir seus usuários finais. Esta última interação também é realizada com um protocolo proprietário em texto-plano e encriptado [15].

Figura 3.13 – Relacionamento entre provedor de serviços e um único cliente.



Os clientes finais são capazes de solicitar informações, via Aplicações *Front-End*, sejam elas *Desktop* ou *Web*. A maioria de seu fluxo cotidiano é compreendido pela obtenção de relatórios que auxiliam a tomada de decisões, pois relacionam os dados de telemetria dos veículos monitorados com inúmeros parâmetros, tais como eficiência energética, tempo de execução de tarefas, desgaste mecânico, incidentes nos trajetos. É importante ressaltar que os Dados Pré-Processados e que são dispostos de modo genérico pelo MAS alimentam o CAS, que por sua vez filtra quais destes dados são de interesse. Desta maneira, os usuários finais têm acesso somente a um conjunto de informações consolidadas e que lhes provêem uma visão geral de toda sua frota.

4 RESULTADOS

Para o Corpo de Prova escolhido como objeto de estudo neste trabalho, foi elaborado um plano de migração, que envolveu basicamente a transição deste para uma infra-estrutura mais moderna e se estabeleceu uma configuração de Alta Disponibilidade. Para que a transição ocorresse de maneira suave, sem interrupções no serviço, foi utilizado um componente previamente desenvolvido e aqui denominado como *Middleware* Distribuído (MD).

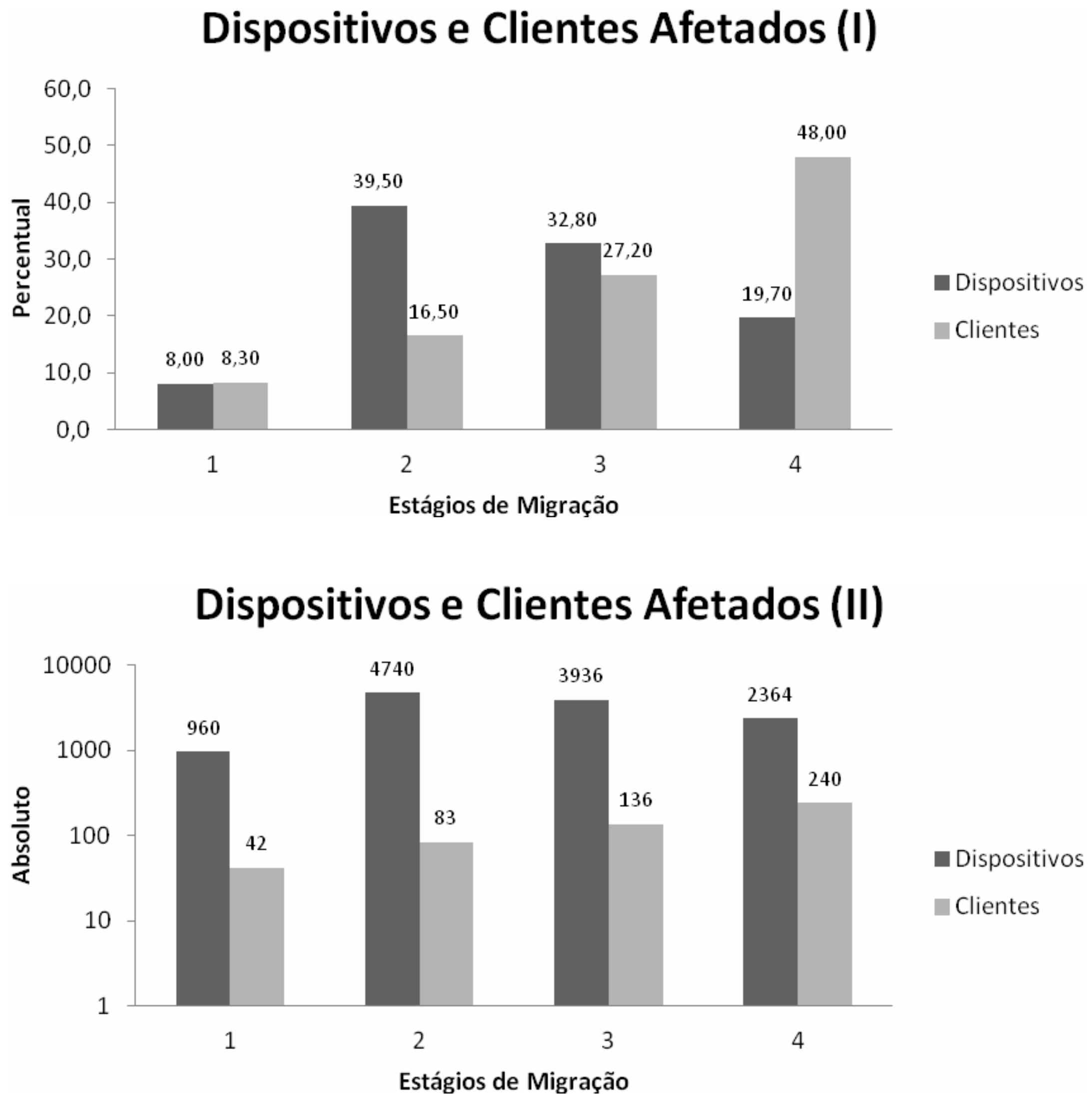
A utilização do MD permitiu que todos os dispositivos que respondiam à antiga infra-estrutura pudessem responder a uma nova infra-estrutura de maneira gradual. Para isso, percentuais do montante de dispositivos foram movidos em meio a diversas fases, até que todo o processo de migração fosse terminado. Para que múltiplas instâncias do MD pudessem operar paralelamente e, ainda sim, compartilharem dados sensíveis às suas operações, denominados como Pontos Críticos, utilizou-se técnicas de computação em *Grid* de memória, distribuindo todas as informações ao longo de diversos nós independentes de uma Rede P2P.

Diferentes fases de migração foram realizadas, cada uma categorizando as características dos dispositivos e o quão dramática seria essa mudança para seus clientes. As fases foram alinhadas da menos crítica para a mais crítica, em ordem de execução. Desse modo, caso existissem falhas durante o processo, estas poderiam ser revertidas e somente um grupo seria afetado durante uma determinada fase transitória. Os grupos de dispositivos foram separados da seguinte maneira:

1. Fornecem somente geo-localização aos clientes.
2. Recebem periódicas reconfigurações em seus relatórios periódicos.
3. Retém mudanças comportamentais em atuadores mediante reconfiguração.
4. Emitem alertas críticos em tempo real, tais como violações nos veículos ou pedidos de socorro.

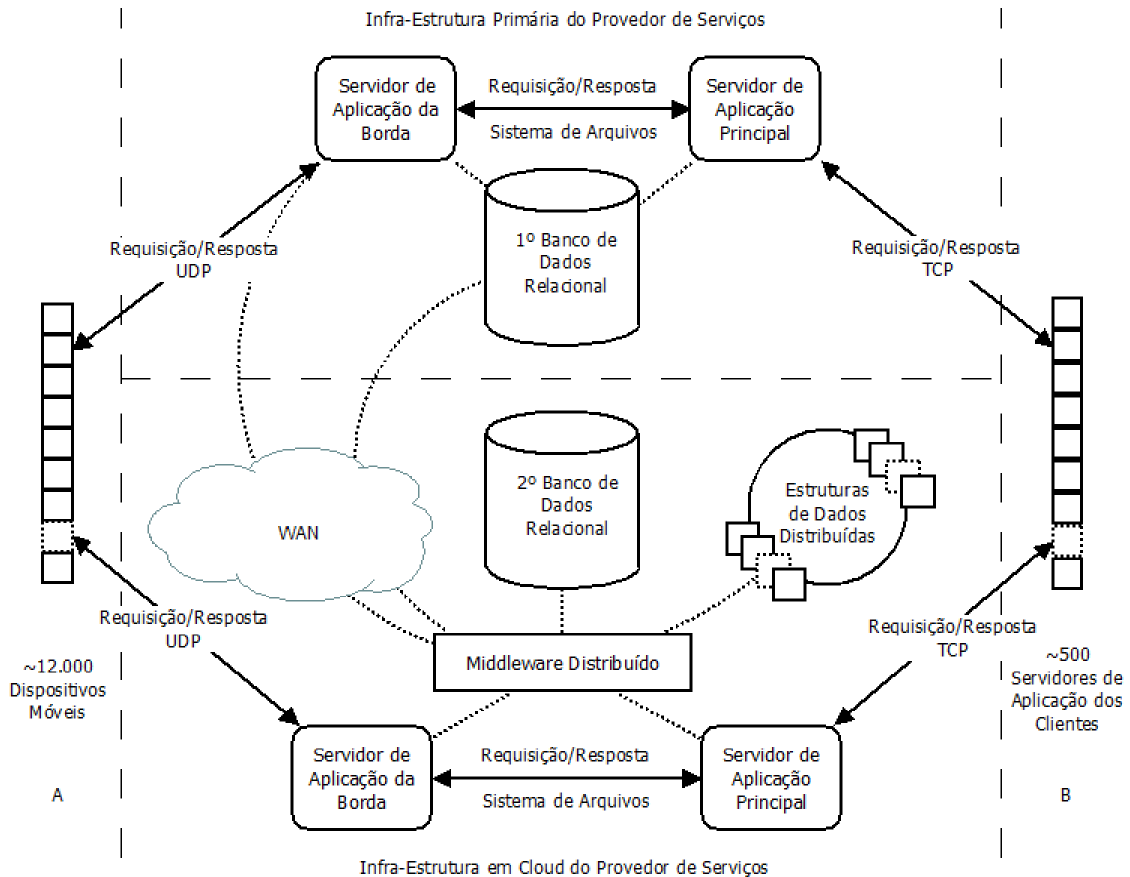
A Figura 4.1 mostra a relação percentual (I) e o número absoluto (II) encontrados em cada estágio da Migração [15]. É possível notar que as quantidades de dispositivos móveis e clientes foram distribuídas de uma maneira proporcional ao longo de cada fase, de modo que em caso de falhas um rollback pudesse ser realizado sem comprometer as demais fases já implantadas, além de não afetar pelas mudanças de uma única vez toda base de dispositivos móveis e clientes.

Figura 4.1 – Clientes e dispositivos afetados por cada estágio de migração.



Neste modelo de solução, representado pela Figura 4.2, podemos verificar que ambos os ecossistemas puderam operar simultaneamente, interconectados pela WAN, onde os dados de estados eram replicados da infra-estrutura antiga para a infra-estrutura nova. Conforme os estágios da migração foram avançando, gradualmente a nova infra-estrutura assumiu por completo o controle das operações [15].

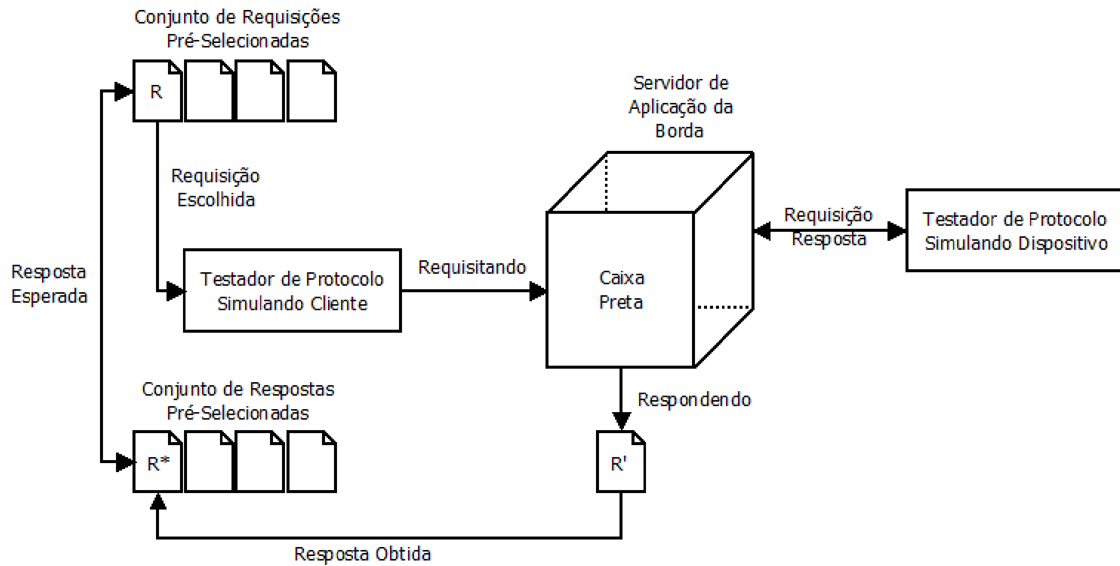
Figura 4.2 – Infra-Estrutura antiga e nova operando simultaneamente.



Para garantir que todos os resultados ocorressem da maneira esperada ou então que eventuais erros fossem minimizados, desenvolveu-se uma estratégia específica de testes, viabilizando a aferição individual de partes internas ao LIS, classificadas como responsáveis pela comunicação com Dispositivos Móveis e MAS. Para isso, um testador de protocolos foi implementado em EAS, de forma que simulações fossem ser realizadas no intuito de cobrir todo o conjunto possível de mensagens conhecidas.

A Figura 4.3 mostra esquematicamente a estratégia de testes adotada. Nesta, pudemos encapsular toda a regra de negócio compreendida pelo EAS como uma Caixa Preta [15]. Dessa maneira, não foi afetado o núcleo de sua inteligência, tendo sido preservado e com a intenção de modificar somente seu comportamento de iterações com outros serviços [31]. Tendo mapeadas, de forma genérica, todas as possíveis trocas de mensagens entre EAS e MAS, se obteve o equivalente à interação entre a infra-estrutura do Provedor de Serviços e seus clientes finais. Com o mapeamento genérico de todas as possíveis trocas de mensagens entre EAS e Dispositivos Móveis, se obteve o equivalente à interação entre o Provedor de Serviços e os Veículos.

Figura 4.3 – Representação do fluxo de testes dos protocolos de comunicação.



Desta maneira, foi organizado um conjunto contendo pares de Requisições *R* e Respostas Esperadas *R**. Para que os testes fossem bem sucedidos, as Respostas *R'* efetivamente obtidas do EAS, tendo sido este tratado como uma Caixa Preta, deveriam corresponder com as Respostas Esperadas *R**.

Caso *R'* e *R** sejam iguais, significa que a funcionalidade que se testou não sofreu com falhas implicadas por mudanças no serviço, sendo assim, estas podem ser integradas à base de código com segurança. Mas, se as respostas diferirem entre si, então as mudanças propostas devem ser rejeitadas. Com resultados diferentes, além da rejeição, podem ser revelados indícios que levem ao traçado completo demonstrado por um comportamento falho. Com esse processo tendo sido integrado à elaboração da solução, foi possível garantir que comportamentos já existentes fossem preservados com segurança e que novas adições à base de código não causariam a quebra das funcionalidades antes consolidadas.

Tendo implementado uma solução Distribuída e integrando o funcionamento de diversos *daemons*, houve a necessidade de tornar ágil o processo de testes. Para isso, foi desenvolvido um conjunto de scripts capazes de realizar o *bootstraping* de todo o ambiente de testes, gerenciando o processo de inicialização e manutenção de estados de todas as aplicações envolvidas. As principais características que tornaram esta uma saída confiável durante a avaliação das mudanças na base de código são as seguintes:

- **Início e Finalização Ordenados:** Por meio de configurações, foi possível definir uma ordem de precedência e requisitos entre os *daemons*, uma vez que a solução como um

todo comporta-se como um Problema de Produtores e Consumidores, existindo tal necessidade para um correto funcionamento de todas as partes envolvidas.

- **Operações em Múltiplas Máquinas:** Possibilita que os *daemons* pudessem ser ativados e desativados de maneira apropriada ao longo de múltiplos nós na rede, de acordo com a natureza do teste.
- **Monitoramento em Tempo Real de Logs:** Falhas podem ser encontradas observando-se os alertas emitidos pelos *daemons* nos arquivos de Log. Assim, quando um comportamento inesperado é encontrado, por não estar em conformidade com padrões pré-estabelecidos, podemos disparar ações, como parâmetros de parada de execução ou então de concessões para continuar com o fluxo.
- **Garantia de que o *Stack* Todo está Rodando:** A condição imposta de que isso seja verdade faz com que a asserção por esta torne-se um parâmetro mandatório para que a solução toda seja avaliada. Os serviços são interdependentes e isso faz com que para uma correta avaliação de comportamentos, todos devem estar funcionando ou nenhum deles deve se manter no ar.

Tendo sido desenvolvido um *Middleware* Distribuído para integrar a solução e viabilizar o processo de migração, um estudo foi realizado sobre o problema que mais poderia inviabilizar as operações em âmbito distribuído. Este se relaciona ao quão escalável uma solução pode ser e está relacionado a um dos Pontos Críticos. Muita demanda por execução de tarefas no BD poderia acarretar em gargalos para as operações e então verificou-se qual era a maior incidência de operações, para que fossem avaliadas.

Foi constatado que o maior número em diversidade de operações ocorria com a Atualização de dados no BD, o que representava aproximadamente 72% do total de possíveis operações no sistema [14]. Por conta desta característica, foi feito um comparativo entre técnicas para a efetivação de dados, utilizando-se o algoritmo mais performático, perante os demais, como padrão para este tipo de operação.

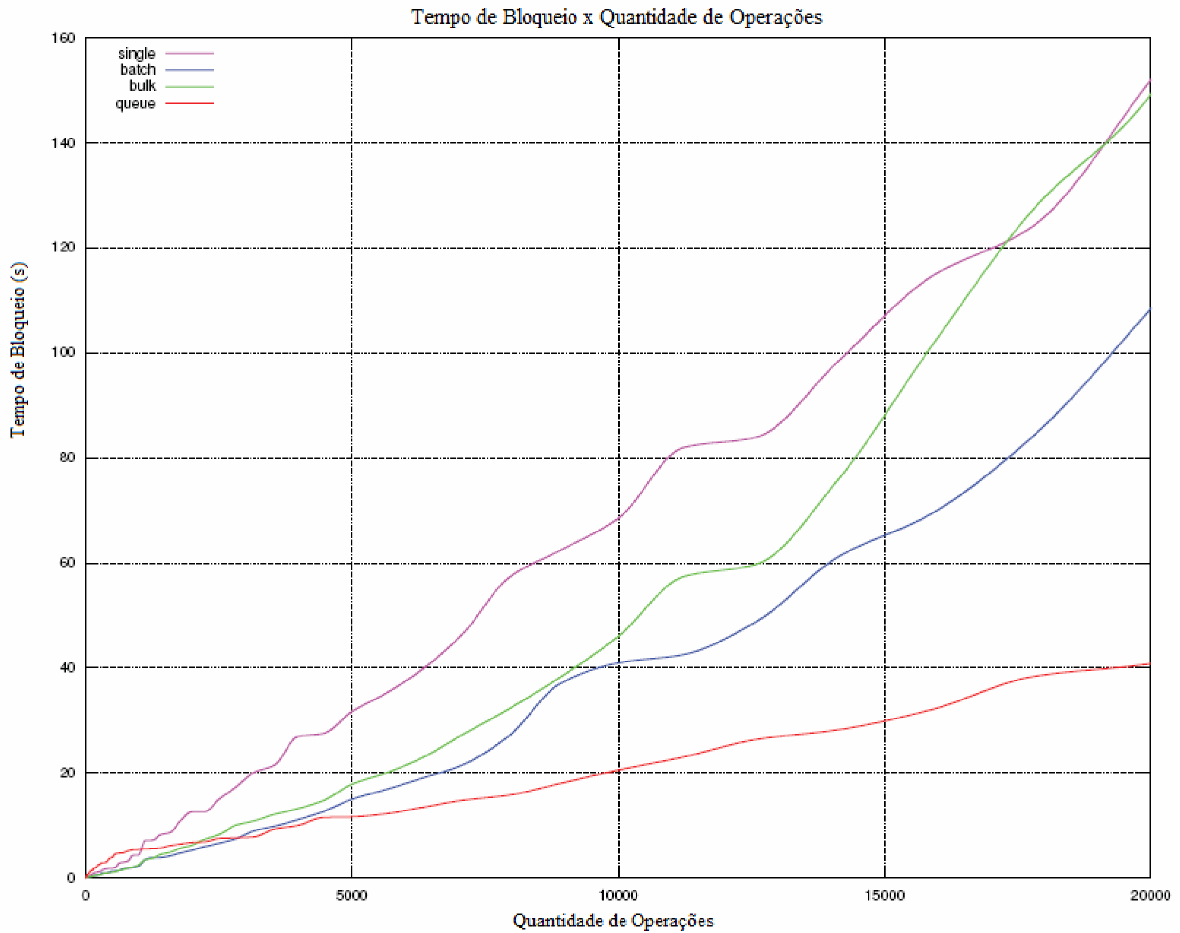
Tendo em vista a expressividade das operações de Atualização, foram avaliados 4 possíveis algoritmos, onde todos se utilizam em um dado momento de mecanismos para consolidação de dados que é compreendido pelo *Driver* de BD:

- ***Single:*** As Consultas são efetivadas uma a uma onde, sequencialmente, é criado um *lock* para execução em BD, a consulta é submetida para o BD, é disparada uma ordem para consolidação dos dados e então o *lock* é destruído.

- ***Batch:*** As Consultas são agrupadas por procedimentos internos ao *Driver*, é criado um *lock* para a execução de todas estas consultas, sendo transmitidas em agrupamentos gerenciados automaticamente pelo *Driver* em forma de *buffers*, consolidadas conforme chegam ao BD e, por fim, o *lock* é destruído.
- ***Bulk:*** As Consultas são agrupadas por concatenação, sem intervenção do *Driver*, é criado um *lock* para a execução de todas as consultas, estas são transmitidas de uma só vez em forma de *streaming*, sendo sua transmissão limitada ao *througput* da rede, consolidadas ao término da chegada e, finalmente, o *lock* é destruído.
- ***Queue:*** As Consultas são inseridas em uma Fila Distribuída, tendo sua replicação garantida e sendo estas replicadas ao longo de todos os nós do MD. Assim, sua execução não ocorre instantaneamente, mas vai sendo atrasada pois as informações já encontram-se consolidadas na rede, podendo assim serem efetivadas em modo *Batch* sem a necessidade de retorno imediato.

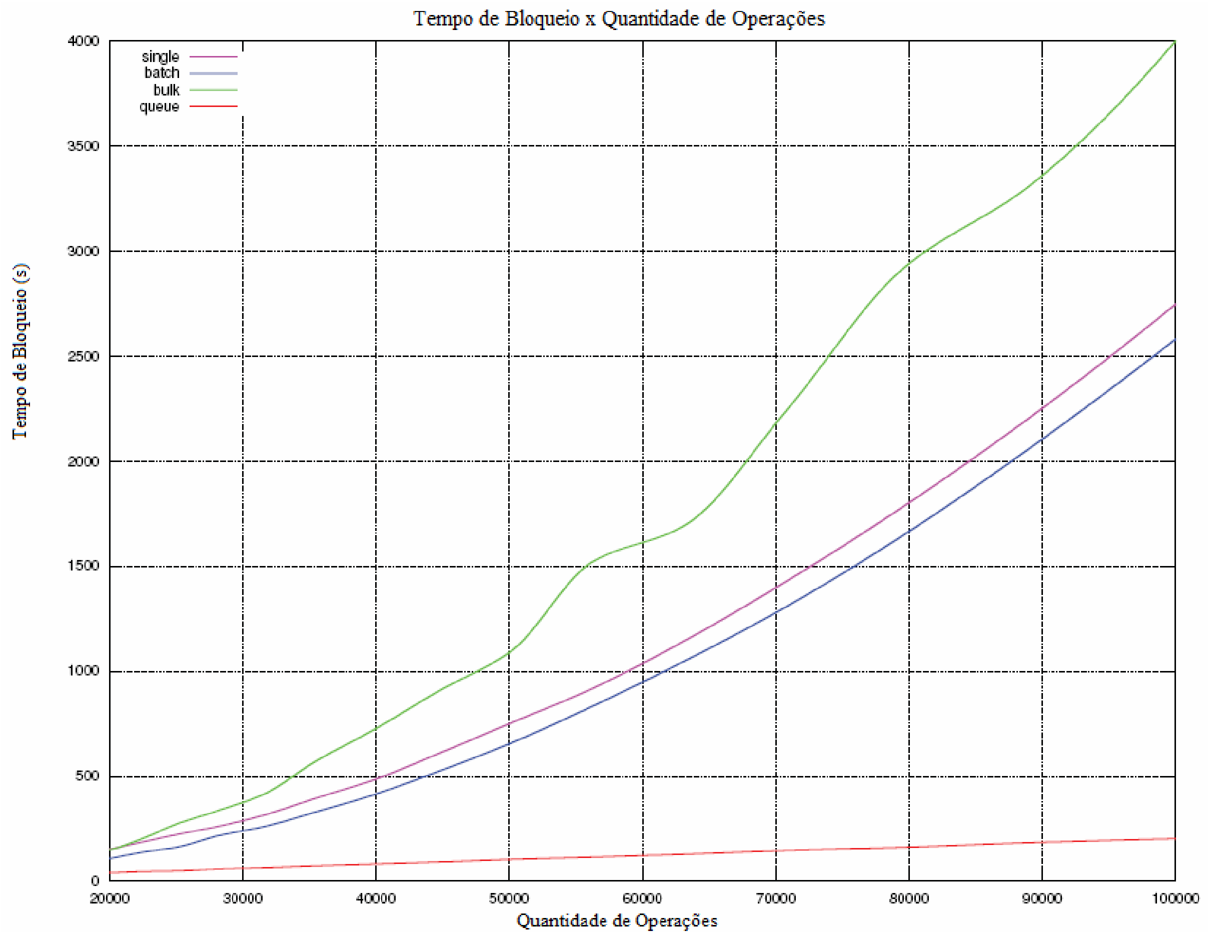
Efetuando-se um Benchmark de desempenho sob estas condições, observou-se que para até 5.000 requisições enfileiradas existe uma competição equilibrada por um melhor desempenho entre os algoritmos. Entre 5.000 operações e 20.000 operações, começa a ficar evidente que a utilização do algoritmo *Queue* torna o tempo de bloqueio menor e, por conseqüência, significa que é mais performático. Tal algoritmo é inerente à arquitetura adotada na concepção do MD, portanto, além de oferecer condições para que as informações a serem efetivadas não se perdessem, mostrou-se melhor que as demais abordagens, conforme aborda a Figura 4.4, comparando o algoritmo *Queue* com os demais, que são amplamente encontradas em modelos tradicionais de acesso a BDs [14].

Figura 4.4 – Tempo de Bloqueio Vs. Quantidade de Operações



Entendendo que o algoritmo empregado como principal efetivador de operações de BD no MD, sobre o maior conjunto de Consultas catalogadas, os testes foram estendidos para se visualizar de maneira mais clara uma situação onde somente uma solução robusta pudesse suportar a carga a carga que lhe fora imposta. Desse modo, foi enfileirado um total de até 100.000 operações em BD, em uma única bateria de testes conforme é ilustrado pela Figura 4.5 [14].

Figura 4.5 – Tempo de Bloqueio Vs. Quantidade de Operações



O resultado foi satisfatório, pois o tempo de bloqueio causado pelo algoritmo *Queue* apresentou-se o menor dentre todos os algoritmos analisados. Com carga máxima, este se manteve na faixa de 250 segundos, enquanto que o segundo melhor algoritmo se manteve na faixa dos 2.500 segundos. Ou seja, a diferença de desempenho observada pode ser compreendida por uma taxa de eficiência até 10 vezes maior para o algoritmo *Queue*, quando comparado com o segundo melhor algoritmo em termos de espera bloqueante, o *Batch*. Em comparação com o pior de todos os algoritmos, o *Bulk*, sua eficiência fica na marca de uma taxa até 16 vezes maior.

5 CONCLUSÃO E TRABALHOS FUTUROS

O principal objetivo deste trabalho foi modernizar a infra-estrutura de um Sistema de Informações Legado por meio de migração. Tendo esta sido dividida em diversas fases, ocorreu de modo suave e de maneira segura. Foi efetivada gradativamente, o que garantiu que as operações não fossem interrompidas em caso de mal-funcionamento.

As características que tornaram o LIS escolhido como um sistema de difícil transposição para um ambiente altamente disponível, se resumiam em estados críticos armazenados em memória. Estes foram mapeados e com o auxílio de um *Middleware* Distribuído e então replicados ao longo de diversos nós, eliminando assim o problema que se tem ao tentar obter um resultado similar em sistemas *Stateful*.

Realizou-se um *benchmark* no *Middleware* Distribuído, com respeito ao seu desempenho perante outros algoritmos utilizados amplamente em modelos clássicos de desenvolvimento quanto ao acesso a Bancos de Dados. O algoritmo empregado no *Middleware* desenvolvido como fruto deste trabalho se mostrou mais performático que os demais, superando o segundo colocado dentre os outros algoritmos em um fator de 10 vezes. Com relação ao menos performático dos algoritmos, se mostrou 16 vezes mais eficiente.

Para garantir que novas funcionalidades pudessem ser implementadas durante o processo de migração e que, ainda, as antigas funcionalidades continuassem operantes, foram desenvolvidas ferramentas auxiliares voltadas ao teste de protocolo de comunicação com as demais soluções que o LIS mantêm contato direto. Deste modo nenhuma funcionalidade foi integrada sem que testes de protocolo a aprovassem. Isso também garantiu que novas implementações não quebrassem comportamentos bem conhecidos e esperados.

Considerando que a solução evoluiu no sentido de ser capaz de operar com múltiplas instâncias de seus serviços simultaneamente, é sugerido que se faça uma medição sobre qual o nível máximo de escalabilidade que pode ser atingido. Este resultado revelará até que ponto mais instâncias clones atendem a demanda sobrevida operacional da solução, com tempos de resposta satisfatórios o bastante para que os serviços não sejam degradados.

Os simuladores auxiliares, desenvolvidos durante a execução do processo de migração, podem ser estendidos de maneira genérica tal que seja possível sua utilização em soluções de propósitos distintos à que foi avaliada durante este trabalho. Isso pode torná-los parte de um ferramental de código-fonte aberto, existindo a possibilidade de se retornar benefícios para a comunidade sem que segredos industriais sejam expostos.

O conjunto de *scripts* que gerenciam o *bootstraping* dos *daemons* se mostrou genérico o suficiente para que seja utilizado com aplicações heterogêneas, sendo passível de integração a mecanismos de *Orchestration* em Plataformas como Serviço, atualmente construídos sob ambientes virtualizados, na modalidade de Computação em Nuvem.

O modelo aqui exemplificado como um estudo de caso foi obtido por meio de um trabalho realizado em campo, com sistemas em ambiente de produção e tendo sido validado em operações reais. Espera-se que este sirva como apoio a campos de estudo tais como Sistemas de Informações Legados, *Middlewares* Distribuídos, Bancos de Dados Relacionais e Arquitetura de *Software*, uma vez que deste se originaram publicações científicas apresentadas nas edições VIII e X respectivamente do SBSI (Simpósio Brasileiro de Sistemas de Informação).

REFERÊNCIAS

[1] Barga, R., Lomet, D., andWeikum, G. (2002). Recovery guarantees for general multitier applications. *Data Engineering, International Conference on*, 0:0543.

[2] Bisbal, J., Lawless, D., Wu, B., and Grimson, J. (1999). Legacy information systems: issues and directions. *IEEE Software*, 16(5):103 –111.

[3] Brinkkemper, S. (1996). Method engineering: Engineering of information systems development methods and tools. *Information and Software Technology,* 38(4):275–280.

[4] Brodie, M. and Stonebraker, M. (1995). Migrating legacy systems: gateways, interfaces & the incremental approach. *Morgan Kaufmann series in data management systems.* Morgan Kaufmann Publishers.

[5] Canfora, G., Fasolino, A. R., Frattolillo, G., and Tramontana, P. (2008). A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. *Journal of Systems and Software*, 81(4):463 – 480. Selected papers from the 10th Conference on Software Maintenance and Reengineering (CSMR 2006).

[6] Cecchet, E., Candea, G., and Ailamaki, A. (2007). Middleware-based database replication: *The gaps between theory and practice*. CoRR, abs/0712.2773.

[7] De Lucia, A., Francese, R., Scanniello, G., and Tortora, G. (2008). Developing legacy system migration methods and tools for technology transfer. *Softw. Pract. Exper.,* 38(13):1333–1364.

[8] D. Barkai, Peer-to-Peer Computing*, Intel Press*, 2002.

[9] Dos Santos, Isaac José Antonio Luquetti. Metodologia para implementação de um programa de manutenibilidade no ciclo de vida de um sistema nuclear. *Revista Produção Online,* Florianópolis, v. 6, n. 3, mar. 2010. ISSN 16761901. Disponível em: <http://producaoonline.org.br/rpo/article/view/629>. Acesso em: 10 Nov. 2013

[10] Erradi, A., Anand, S., and Kulkarni, N. (2006). Evaluation of strategies for integrating legacy applications as services in a service oriented architecture. *IEEE International Conference on Services Computing*, pages 257–260.

[11] Farris, Paul W.; Neil T. Bendle; Phillip E. Pfeifer; David J. Reibstein (2010). Marketing Metrics: The Definitive Guide to Measuring Marketing Performance. *Upper Saddle River*, New Jersey: Pearson Education, Inc. ISBN 0137058292

[12] Fayçal, H., Habiba, D., and Hakima, M. (2010). Integrating legacy systems in a soa using an agent based approach for information system agility. *International Conference on Machine and Web Intelligence (ICMWI)*, pages 338 –343.

[13] Fowler, M. (2004). Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html.

[14] Herrera, R. P. and Felinto, A. S. (2012). A distributed, multi-staged, high-throughput middleware for relational databases. *VIII Simpósio Brasileiro de Sistemas de Informação (SBSI).*

[15] Herrera, R. P. and Felinto, A. S. (2014). A Case Study on Legacy Information Systems Migration: A Fault Tolerant Approach. *X Simpósio Brasileiro de Sistemas de Informação (SBSI).*

[16] Lewis, G. and Smith, D. B. (2007). Developing realistic approaches for the migration of legacy components to service-oriented architecture environments. *2nd International Conference on Trends in Enterprise Application Architecture*, pages 226–240.

[17] Lu, F., Huang, H., Xu, Z., and Yu, H. (2005). A middleware for legacy application wrapper. *First International Conference on Semantics, Knowledge and Grid,* pages 47–.

[18] Luo, Q., Krishnamurthy, S., Mohan, C., Pirahesh, H., Woo, H., Lindsay, B. G., and Naughton, J. F. (2002). Middle-tier database caching for e-business. *In Proceedings of the 2002 ACM SIGMOD international conference on Management of data, SIGMOD '02*, pages 600–611, New York, NY, USA. ACM.

[19] Martin, R. C. (1996). The dependency inversion principle. *C++ Report*, 8:61–66.

[20] Nasr, K. A., Gross, H.-G., and van Deursen, A. (2010). Adopting and evaluating service oriented architecture in industry. 14th *European Conference on Software Maintenance and Reengineering*, pages 11–20.

[21] O'Brien, L., Smith, D., and Lewis, G. (2005). Supporting migration to services using software architecture reconstruction. *13th IEEE International Workshop on Software Technology and Engineering Practice,* pages 81–91.

[22] Parsa, S. and Ghods, L. (2008). *A new* approach to wrap legacy programs into web services. *11th International Conference on Computer and Information Technology* (ICCIT), pages 442 –447.

[23] Rubel, P., Loyall, J. P., Schantz, R. E., and Gillen, M. (2006). Fault tolerance in a multi-layered dre system: A case study. JCP, 1(6):43–52.

[24] Sneed, H. M. (1995*). Planning the reengineering of legacy systems. IEEE Softw*., 12(1):24–34.

[25] Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., and Tantawi, A. (2005). An analytical model for multi-tier internet services and its applications. *SIGMETRICS Perform*. Eval. Rev., 33:291–302.

[26] Vemuri, P. (2008). Modernizing a legacy system to soa - feature analysis approach. *IEEE Region 10 Conference (TENCON*), pages 1 –6.

[27] Warren, I. and Ransom, J. (2002). Renaissance: a method to support software system evolution. *26th Annual International Computer Software and Applications Conference (COMPSAC*), pages 415 – 420.

[28] Welsh, M., Culler, D., and Brewer, E. (2001). Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35:230–243.

[29] Wu, B., Lawless, D., Bisbal, J., Grimson, J.,Wade, V., O'Sullivan, D., and Richardson, R. (1997). Legacy systems migration-a method and its tool-kit framework. *Asia Pacific Software Engineering Conference and International Computer Science Conference* (APSEC / ICSC), pages 312 –320.

[30] Yang, H. Y., Tempero, E., and Melton, H. (2008). An empirical study into use of dependency injection in java. *In Proceedings of the 19th Australian Conference on Software Engineering*, pages 239–247, Washington, DC, USA. IEEE Computer Society.

[31] Zhang, B., Bao, L., Zhou, R., Hu, S., and Chen, P. (2008). A black-box strategy to migrate gui-based legacy systems to web services. *IEEE International Symposium on Service-Oriented System Engineering*, pages 25–31.

APÊNDICES

APÊNDICE A

1º Trabalho publicado pelo Autor (SBSI 2012)

VIII Simpósio Brasileiro de Sistemas de Informação (SBSI 2012)
Trilhas Técnicas

A Distributed, Multi-Staged, High-Throughput Middleware for Relational Databases

Rafael de Paula Herrera[1,2], **Alan Salvany Felinto**[1]

[1]Department of Computing – State University of Londrina (UEL)
Londrina, PR – Brazil

[2]Veltec Technological Solutions
Londrina, PR – Brazil

herrera.rp@gmail.com, alan@uel.br

Abstract. *In this paper we present a distributed middleware for relational databases. Its development was motivated by the need of improvements on a set of legacy systems from the automotive logistics and tracking industry. The solution proved to be effective once it was possible to increase the throughput of handled Requests and offer a minimum level of fault tolerance by employing a pipelined architecture along with distributed data structures. Its deployment ensured that the same legacy applications ecosystem could evolve and operate under growing commercial demand.*

Resumo. *Neste trabalho apresentamos um middleware distribuído para bancos de dados relacionais. Seu desenvolvimento foi motivado pela necessidade de se aprimorar um conjunto de sistemas legados da indústria de logística e rastreamento automotiva. A solução demonstrou ser eficaz uma vez que foi possível aumentar a vazão de requisições tratadas e oferecer um nível mínimo de tolerância a falhas ao se empregar uma arquitetura multi-estágios em conjunto com estruturas de dados distribuídas. Sua implantação garantiu que o mesmo ecossistema de aplicações legadas pudesse evoluir e operar sob crescente demanda comercial.*

1. Introduction

In technology-based automotive industry, one of main trends is that their applications being migrated to Cloud Computing model. It is usual that back-end legacy systems remains in production over years on infrastructures prone to failures at several levels. Technological and architectural restrictions imposed on scenarios like these, make them not supporting smooth transition strategies to a distributed nature. Also, organizations operating growth, caused by fleet and user base increase, makes legacy systems become gradually saturated.

The main factors that prevented the horizontal scaling of service was how they were strongly attached to the application of so-called *Relational Database Management Systems* (RDBMS), coupled with its waiting time caused by blocking execution. The development model used before and not subject to brief architectural changes, was completely thread-centric and suffered from high concurrency overhead on reading and writing information into relational databases.

On the other hand, implementing a fault tolerant mechanism is not a trivial task in applications whose core activities relies exclusively on storing internal states in memory. To ensure operations consistency of offered service, despiting application servers suffering from outages, data must be replicated over a sufficient number of nodes in the network. Being the service dependent on multiple Internet links and having its physical infrastructure subject to periodic maintenance, it becomes ineffective to employ some solution whose performance is limited to LAN context and single data center.

These reasons lead us to create a distributed middleware as an alternative to bring legacy systems to some level of survival, enough, unless they can be completely migrated to an architecture compatible with market needs.

In section 2, we are going to walk over a set of works that addressed performance and fault tolerance issues. They directly influenced the chosen design of middleware, whose higher level architectural details about how distributed data structures and algorithms interact are explained in section 3. A performance measurement is shown in 4, where classic database committing algorithms and proposed approach are confronted, and its results are properly explored.

2. Related Work

In [Bisbal et al. 1999], difficulties found during the process of maintaining legacy information systems was addressed. This can be increased even more, if distributed back-end features introduction is needed on services, in production environment and designed before to working as singular instances. We propose the idea that when using a middleware whose characteristics are comparable to the presented in this work, it is possible to ease much of migration process to a distributed environment. This is depicted by a re-reading performed over the figure 1, which relates the severity of architectural modifications introduced in information systems.

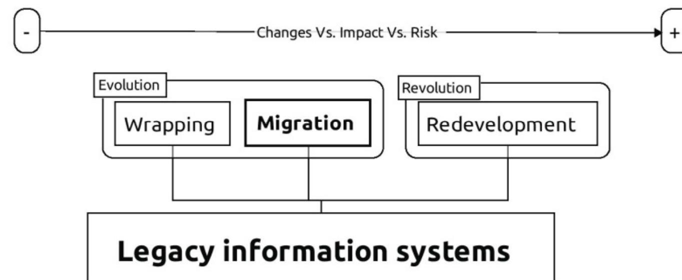


Figure 1. Severity changes on legacy information systems. The migration strategy is eased by use of middleware.

The work [Rubel et al. 2006] shows the importance of employing fail-over on applications whose operation is real-time. In many ways, its problem is similar to that found in the application we have chose to migrate to a distributed environment.

A discussion is promoted in [Barga et al. 2002] on ways to deal with failures occurring in databases, when the access is done by multi-tiered applications. In our

approach, there was linked signals to specific Responses of Requests which caused errors, when their respective SQL statements would be processed by the RDBMS. In addition, we also employed logs for audit.

Back-end services, multi-tiered interconnected by queues are discussed in [Urgaonkar et al. 2005]. Multiple processing stages connected by queues are proposed in [Welsh et al. 2001]. Both works have directly influenced the design of middleware internal processing stages. The way information is retrieved and inserted in the distributed data structures were adjusted for events that would guide all the flow in reactive way.

A prototype was developed, presented in [Luo et al. 2002], whose main motivation was to provide an intermediate caching layer between applications and database. It addressed the possibility of multiple layer instances, which occurs similarly with middleware presented here. [Cecchet et al. 2007] discusses data replication and performance characteristics, both having been taken into consideration during prioritization of distributed data structures which would be replicated across the WAN.

3. Architecture

The middleware reacts to presence of information in their structures of input and output, both replicated across multiple nodes in network. Uses distributed computing resources, provisioned by Hazelcast framework. Its components were developed based on conventions adopted by Google Guice framework.

Its architecture was influenced by the Inversion of Control design pattern [Martin 1996], implemented via Dependency Injection [Fowler 2004, Yang et al. 2008], providing high modularity and loose coupling. This made the input and output structures fully replaceable, since their implementations conforms to the established contract made by their interfaces BlockingQueue ConcurrenMap, respectively.

The data are assigned so that each node, n, is responsible for sharing its active amount along the grid nodes, while it retains a data partition load as backup from its predecessor node, $n - 1$, whose is responsible. Similarly, his successor node, $n + 1$, charges the active data from node n as its backup data partition. This training circular list, followed by the processes of partitioning and backup, guarantees a minimum fail-over level over information stored in memory grid.

A basic API was built, so the developer can interact with the grid in a simple way: inserting, removing and retrieving information over distributed data structures. Thus it was possible to fully decouple all the calls with direct references to RDBMS, performed by legacy systems, while the backward source code has been fully maintained, ensuring operating compliance for native implementations present in numerous locations such as those who are in agreement with contract established by ResultSet interface.

3.1. Stages

The middleware execution flow follows a pipeline composed of multiple stages. As shown in figure 2, stages are being classified as:

1. **Identification** The accumulated Requests in distributed input queue are consumed according to a particular algorithm. Each Request processed must have a corresponding SQL statement. This association is accomplished by means of an

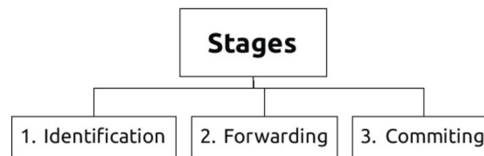


Figure 2. Ordered pipeline stages according to default operating flow.

internal correspondences map so it can cross the possible Requests kinds with the SQL statements related ones. Once obtained, the SQL statements are classified by its semantics.

2. **Forwarding** Each SQL statement is sent to an internal blocking queue and then an consumer algorithm specialist in dealing with that operations family, will then generate its pure SQL statements which will be committed on RDBMS.
3. **Committing** After being consumed, according to a specific policy, the SQL statements are fired against the RDBMS through JDBC (Java TMDatabase Connectivity). Requests that originates Responses are related upon on a communication key, automatically generated by Grid Client, and provided on its distributed map.

3.2. Node

Each grid node corresponds to a stand-alone instance of distributed data structures, stored in RAM memory from host servers. Upon configuration, can perform discovery of other instances, if any, over the LAN/WAN. There are two main data structures, persistent and synchronized across all grid nodes, known respectively as "Requests Queue" and "Responses Map", as shown in figure 3. They are considered well-defined points of data Input/Output to all of middleware instances.

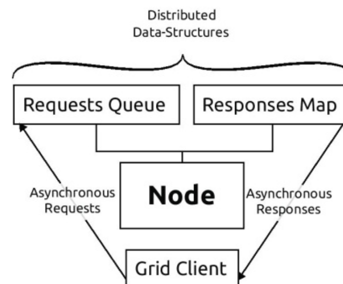


Figure 3. A grid node, having its distributed data-structures accessed by client.

Periodically, every node tries to find other active instances in the LAN, firing UDP multicast messages to the address 224.2.2.3 on port 54327. If it finds another active instance, establishes a TCP connection to it and starts the authentication process, that is validated by a group code identifier and a password, both must be known to all grid connected nodes. The communication is optionally encrypted and is based on JavaTMJCA (Cryptography Architecture). Being the connection successfully

authenticated, newcomers nodes are organized with others in a P2P network and begins partitioning its data among themselves. One grid can have its data-structures distributed over the WAN, as long as at least one of its nodes knows about at least one IP address located on another grid. In last case, the discovery is made solely based on TCP messages.

3.2.1. Grid Client

In order to make the information access transparent, to distributed data structures, there was developed a basic wrapping API that provides different ways to run the same set of actions on the grid. Basically, they are understood by (i) Requests Offer and (ii) Responses Recovery:

Blocking Waiting (i) Tries to insert a Request, blocking the application execution until there is an available slot in Requests Queue buffer. (ii) Attempts to retrieve the answer to a previously performed Request by blocking the application execution until the Response is available on the Responses Map.

Non-Blocking Waiting (i) Tries to insert a Request, stating the case it is not possible. (ii) Attempts to retrieve the answer to a previously performed Request, stating if it have not been made available on the Responses Map.

Timeout Driven Blocking Waiting In both cases, (i) and (ii) are extensions of their "Blocking Waiting" and "Non-Blocking Waiting" versions, respectively. (I) Tries to insert a Request, blocking the application execution until there is an available slot in the Requests Queue buffer, respecting a maximum waiting timeout and stating if it is not possible. (ii) Attempts to retrieve the answer to a previously performed Request by blocking the application execution until the Response is available on the Responses Map, respecting a maximum waiting timeout and stating if it has not been made available.

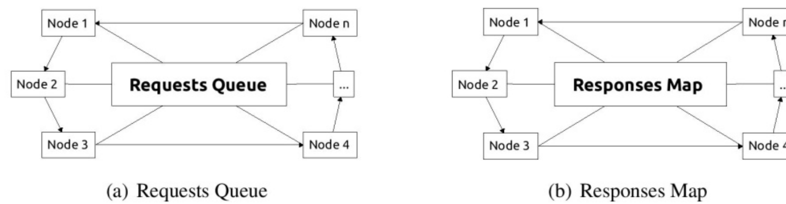
Trials Number and Timeout Driven Blocking Waiting In both cases, (i) and (ii) are extensions of their "Timeout Driven Blocking Waiting" versions. The blocking is driven by a timeout and the process is repeated according to a established number of trials.

3.3. Requests Queue Vs. Responses Map

To take advantage on middleware, applications are encouraged to make use of provided Grid Client API, instead of accessing directly the distributed data-structures. Thus, it is possible to Request and retrieve data in a transparent and simple way. It works on Requests Queue and Responses Map.

When there is need to operate over data, client grid application will input a Request in the distributed queue, exemplified by the figure 4(a). A middleware instance soon will be responsible for consuming the said Request and process it in the pipelined stages.

Once processing have been generated in RDBMS, the answer is input back on distributed map, exemplified by figure 4(b). The application just Requested that operation should be watched, asynchronously, so a Response is signaled by its presence on the Responses Map. To make this possible, one should choose a method of consumption among those implemented by Grid Client.



(a) Requests Queue (b) Responses Map

Figure 4. I/O distributed data-structures over grid nodes.

Distributed data-structures eases the backup process, employing a circular linked list on its self-organizing way, that guarantees a minimum level of fail-over on outages.

3.4. Request Vs. SQL Statement Vs. Response Vs. Communication Key

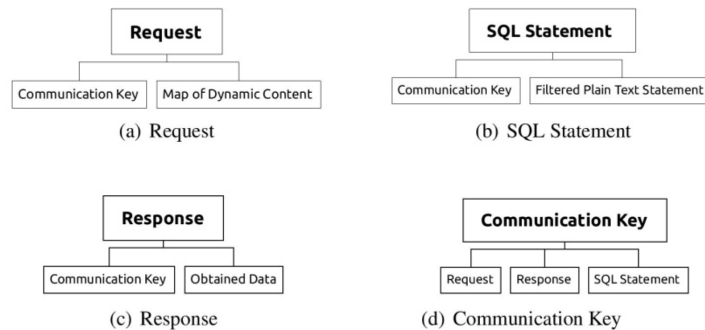
As shown in figure 5(a), every Request must contain both Communication Key and Map of Dynamic Content. The Communication Key is automatically generated by grid client, which can be replaced by a Long value, provided that it guarantees uniqueness along the grid until its related Response be consumed. The Map of Dynamic Content should be implemented based on Map interface contract, so that its key-value must corresponds to "String x String" format.

During SQL Statement mapping to related Request, an iteration is performed over all records from Map of Dynamic Content, in order to find labels inside the plain text SQL stored. If occurrence is found, the label is then replaced by its associated value, being assured that attempted SQL injection attacks will be barred.

A SQL statement should have a 1x1 relationship with a Request. It has in its content, as shown in figure 5(b) statement, a plain text representing SQL statement that should be offered to the RDBMS as an operation after being filtered. When operations are invariant its representation is static. When dynamic data should integrate statement fields, a label is searched exactly like it had been inserted into Map of Dynamic Content at original Request. If it has been found, one would have its textual token representation replaced by its corresponding value.

A Response must contain Communication Key and Retrieved Data. The first must be equals to that found in the Request that originated the Response. The second must be some data-structure that implements the interface specified by ResultSet and also, capable of assuming a disconnected form. This factor is crucial on performing data binary serialization before being persisted throughout distributed data-structures. We used a CachedRowSet interface implementation that includes all of requirements.

Depending on the nature of Requests offering mechanism, it would be necessary the assignment of a unique and pseudo-random Communication Key, able to identify each Request made. Thus, any information which was submitted by Grid Client can be referenced at any stage of processing in middleware internal flow, such as the relationship shown in figure 5(d). In general, this process simply does the linkage as a cross reference between a Request and its corresponding Response among distributed data-structures. Over normal circumstances, the Communication Key is automatically generated by Grid Client and then is attached to the Request.



(a) Request (b) SQL Statement

(c) Response (d) Communication Key

Figure 5. Entities and the relationship between them.

3.5. Mapping

At the time they are consumed, Requests go through a filtering. Its kind is identified according to an internal mapping system, responsible for relating the Requests with their SQL Statements, as shown in figure 6. Thus, it is expected that all operations subject to turn into Requests and subsequent execution in RDBMS are named and linked explicitly. This feature improves security, since Requests go through network without any form of SQL dialect. Thus, the scheme employed in information architecture internal to the RDBMS, is not revealed.

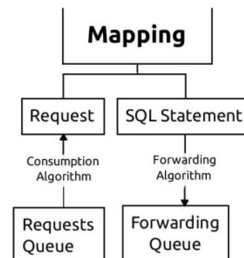


Figure 6. Mapping Requests to their respective SQL Statements.

The set of supported operations by the middleware was selected taking based on those found in the application which had been integrated. Possible operations on the standard RDBMS are understood by Select, Insert, Update, and Removing Stored Procedures.

3.6. Proxy

Having a Request received from client application and crossed with their respective SQL Statement, the routing takes place according to its kind. For each one there is an internal queue that holds their peers, as shown in figure 7. These queues have their data consumed by specialized algorithms, because each kind of SQL Statement generates a specific nature of operations on data stored in RDBMS.

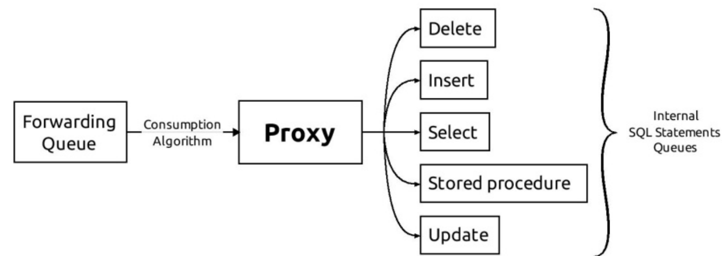


Figure 7. The Proxy forwards SQL Statements to its properly queues.

With the presented distinction, it is possible to identify the frequency with which operations are determined over submitted data. Thus, one can make a fine tunning on employed consumer algorithms, improving committing operation process in RDBMS according to optimal interval timings. Additionally, this information can be used to employ specific improvements in implementation terms.

3.7. Commiter

Once they have been consumed from their internal queues, SQL Statements are submitted to RDBMS with aid of specialized algorithms in the category of operations to be performed. Figure 8 effected shows that a pool manages database connection resources. Once SQL Statements are successfully executed, their answers will be input into the Responses Map. The same Communication Keys from original Requests are attached to the Responses.

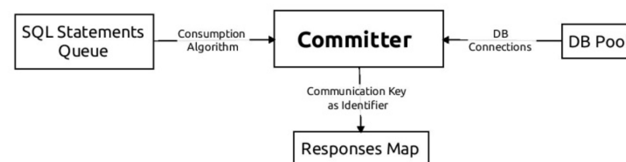


Figure 8. Each SQL Statements queue owns a specialized consumption algorithm that commits operations on RDBMS and returns a Response.

The operations consumption and committing that modifies the durable state of data can be accomplished by use of batch algorithms, since their answers can not be important, resulting in performance gains on RDBMS. It is found under the names (a) Bulk and (b) Batch. In (a) an amount of SQL Statements are concatenated and submitted together to RDBMS as a single SQL Statement. (b) is performed scheduling SQL Statements as they arrive, being validated and submitted to the RDBMS in a second time. Both can be used on the removal, updating and inserting SQL Statements.

Ones that require review about their Responses, are classified under the name of (c) Single, does not go through buffers. They are being treated and subjected directly, in specialized way, one by one. Can be used as any operation kinds and is mandatory to perform read-only queries and stored-procedures invocations.

3.8. Database Pool

In order the middleware can properly use the offered resources by RDBMS, there was employed the Apache DBCP (Database Connection Pool) component. It is a specialized database pooling solution and easily configurable for the following parameters: limiting minimum and maximum amount of active connections used by client applications, validation of connections through standard SQL query, re-establishment of lost connections, unicode support and integration with a wide JDBC drivers variety.

Figure 9 exemplifies pool access to several previously established database connections. As required, the pool checks for resources availability. If so, a fully managed connection is returned. Else, is told to wait until resource is available.

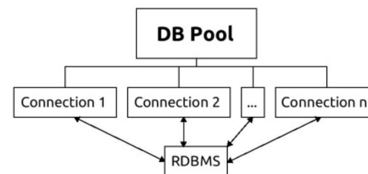


Figure 9. The Pool maintains control over RDBMS connections and provides them, as automatically managed resources.

4. Results

A core business back-end application was chosen as proof of concept on migrating to a distributed environment. The use of middleware played a key role easing the entire process. During the elaboration of data committing mechanism, we conducted a survey over a variety of SQL Statements which was found on direct RDBMS calls. The table 1 lists SQL Statement kinds with its occurrence.

Table 1. SQL Statement kinds found in proof system

SQL Statement kind	Total	$\approx$ Total (%)
Delete	2	1.92
Insert	5	4.80
Select	14	13.46
Stored Procedure	8	7.69
Update	75	72.11

The number of SQL Statements related to updating data, about 72.11%, has motivated the blocking time measurement, which was suffered as result of this kind of application calls.

This comparison relates the number of processed operations with timing that application remains blocked until all being completed. The experiment generated 30 samples after having 1 up to 100,000 Requests processed, consumed by traditional algorithms for direct access to the RDBMS versus the suggested competitor, employed by the use of the Grid Client API, where middleware operates. All iterations, the

set of chosen operations was defined randomly, in order to not favor any caching mechanism. Relevant characteristics of computer hardware that served the experiment are: Intel®Core™i3-350M (3M Cache, 2.26 GHz), 4GB DDR3 RAM, Samsung HM321HI HD and RTL8101E/RTL8102E PCI Express Fast Ethernet controller.

Figure 10 shows significant throughput increase, due to asynchronous model used in detriment to direct access on RDBMS. Up to ≈5 thousand of operations, we observed that there is still a level of competitiveness between different consumption approaches. On ≈5 thousand up to ≈20 thousand of operations, becomes clear that direct access to RDBMS methods are blocking the application at levels much higher than that offered by the presented middleware.

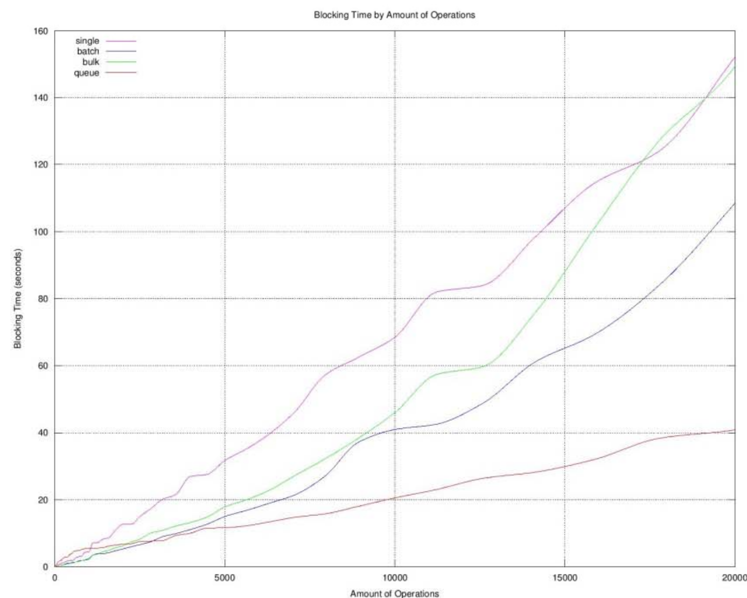


Figure 10. Application blocking time (s) Vs. amount of operations.

Figure 11 shows that when passing ≈20 thousand operations mark, we can see clearly the importance of having been used a queue based data structure as an entry point for Requests, fact that gave support to a much higher magnitude of overload with respect to that provided by traditional methods of blocking waits for RBDMS operations.

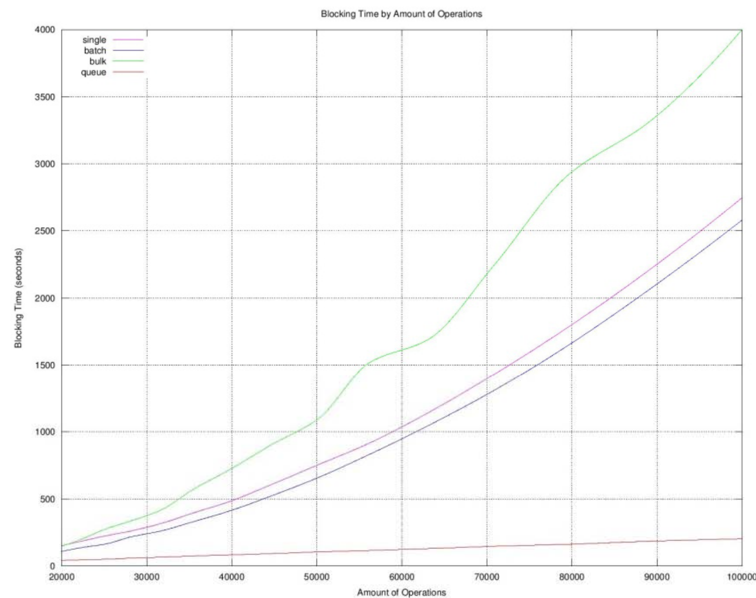


Figure 11. Application blocking time (s) Vs. amount of operations.

5. Conclusion

The middleware was successfully able to assist the process of migrating a back-end systems to a distributed environment. This was achieved because the responsibility to ensure recovery and durable writing data in RDBMS was completely detached from the application in evidence. This process, before comprised by a monolithic flux, was replaced by an event-driven approach, reactive to present data in distributed data-structures.

Information replication has eased the single points of failure elimination across entire chained applications. Being data essential to consistently working of services persisted over several nodes in the grid, it became feasible the process of back-end services instances multiplying on LAN/WAN, the main effect was the risk reduction due to outages. Having been satisfied with information flow processed, its implementation ensures a higher survival course of commercial operations for legacy applications ecosystem.

The developed migration model, able to make back-end applications operating on distributed environments, is being explored as primary means of a smooth applications transition to an infrastructure entirely based on cloud computing and the first stage was solely dependent on the successfully integration and deployment of proposed middleware.

References

Barga, R., Lomet, D., and Weikum, G. (2002). Recovery guarantees for general multi-tier applications. *Data Engineering, International Conference on*, 0:0543.

Bisbal, J., Lawless, D., Wu, B., and Grimson, J. (1999). Legacy information systems: Issues and directions. *IEEE Softw.*, 16:103–111.

Cecchet, E., Candea, G., and Ailamaki, A. (2007). Middleware-based database replication: The gaps between theory and practice. *CoRR*, abs/0712.2773.

Fowler, M. (2004). Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html.

Luo, Q., Krishnamurthy, S., Mohan, C., Pirahesh, H., Woo, H., Lindsay, B. G., and Naughton, J. F. (2002). Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 600–611, New York, NY, USA. ACM.

Martin, R. C. (1996). The dependency inversion principle. *C++ Report*, 8:61–66.

Rubel, P., Loyall, J. P., Schantz, R. E., and Gillen, M. (2006). Fault tolerance in a multi-layered dre system: A case study. *JCP*, 1(6):43–52.

Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., and Tantawi, A. (2005). An analytical model for multi-tier internet services and its applications. *SIGMETRICS Perform. Eval. Rev.*, 33:291–302.

Welsh, M., Culler, D., and Brewer, E. (2001). Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35:230–243.

Yang, H. Y., Tempero, E., and Melton, H. (2008). An empirical study into use of dependency injection in java. In *Proceedings of the 19th Australian Conference on Software Engineering*, pages 239–247, Washington, DC, USA. IEEE Computer Society.

APÊNDICE B

2º Trabalho publicado pelo Autor (SBSI 2014)

A Case Study on Legacy Information Systems Migration: A Fault Tolerant Approach

Rafael de Paula Herrera[1], Alan Salvany Felinto[1]

[1]Department of Computing – State University of Londrina (UEL)
Londrina, PR – Brazil

herrera.rp@gmail.com, alan@uel.br

Abstract. *Legacy Information Systems play key-roles on organizations development and growth. However, they can be considered as risky factor to operations chain whether they do not meet the demanding or become acting as single point of failures. In this work, we propose a migration model which is able to handle systems that depend on Relational Databases and its changes were driven through the use of a distributed middleware. We also pose how this approach was successfully applied while migrating a Legacy Information System to a Cloud Computing based infra-structure, adding fault-tolerance to its architecture as a competitive advantage, enabling the related services to be clustered and then horizontal scaled on demand. All major concerns on how the whole solution and its aggregated tools were conceived are discussed in high-level details, so them can be solely reproduced and integrated to another systems in order to achieve the same goals or improve its level of quality assurance.*

Resumo. *Sistemas de Informações Legados são peças-chave no desenvolvimento e expansão das organizações. No entanto, podem ser considerados fatores de risco para a cadeia operacional se não atenderem a demanda ou se agirem como pontos singulares de falhas. Neste trabalho, propomos um modelo de migrações capaz de lidar com sistemas que dependem de Bancos de Dados Relacionais, tendo suas mudanças sido conduzidas por meio da utilização de um middleware distribuído. Também mostramos como esta abordagem foi aplicada com sucesso durante a migração de um Sistema de Informações Legado para uma infra-estrutura baseada em Computação nas Nuvens, adicionando tolerância à falhas como vantagem competitiva, possibilitando que os serviços relacionados fossem clusterizados e então escalados horizontalmente sob demanda. As principais preocupações sobre como toda a solução e suas ferramentas agregadas foram concebidas são discutidas com um detalhamento em alto-nível, de modo que possam ser individualmente reproduzidas e integradas em outros sistemas, visando atingir os mesmos objetivos ou melhorar os níveis de garantia da qualidade.*

1. Introduction

Information Systems play key-roles on sustainable evolution to practically every industrial segment. Their continuous use represents Return of Investment (ROI) as one of the most tactile results along productive chain [Vemuri 2008].

Over the time, most of Information Systems (IS) tend to be classified as Legacy Information Systems (LIS) due to several reasons, from obsolete Hardware and/or Operating Systems in which they are running on, to social factors as lack of fully understanding about its entire lifecycle or codebase, caused by documentation absence or loss [Brodie and Stonebraker 1995, Bisbal et al. 1999].

In this work we propose a migration model that was developed and successfully applied over production environment in a real time vehicle fleet monitoring LIS. Its application stack ran in a hardware that was highly susceptible to failures and yet could be smoothly moved to a Cloud Computing infrastructure without services disruption.

A conceived distributed middleware [Herrera and Felinto 2012] heavily influenced the design of migration model presented here. We also pose its base architecture details along qualitative results using a set of auxiliary components that was developed in order to assist the systems integration with its Service Provider (SP).

In section 2, we present all related works that influenced us during the architecture designing, which also increased our vision about the development of auxiliary solutions that could allow a secure evolution of the present proposal, while applied to ASP in production.

We introduced the overall environment in section 3, reporting in details the main interdependence relationship found on services provider. Basically, blocks portrays how different systems integrates themselves into a single high-level solution, which end-to-end is taken in details in order to provide us a wider comprehension about the criticality on changing the whole chain.

During the analysis of all systems which composes the solution, we rose up several questions about critical points that could lead the application to fragility and failures. In order to mitigate such behaviors in an improved infrastructure, there were reported the singular points of failure we found, in section 4, being properly cleared.

With an identified opportunity of improvement and knowing the wish on moving the solution from an obsolete infrastructure to another one which could be based on Cloud Computing, we exposed in section 5 the details of developed and applied migration model.

We emphasize critical questions that motivated the designing of strategies and auxiliary tools, so the process could be concluded without outages on services providing.

Finally, we approach on section 6 our ideas about future works, which we believe to be the next potential contributions to distributed systems architecture area, that once well understood in details, could be applied to innumerous LIS from several kinds.

2. Related Works

Maintaining a LIS requires concerns on data interchangeability between related systems, so several approaches were described by [Bisbal et al. 1999]. Its main contribution to our research was the properly discretization and analysis of migration strategies, exposing qualitative up/downsides from each one of them. Inspired in a set of techniques and best practices stated in, we developed a distributed middleware which main focus is to take care of performance issues on LISes that strictly depends on relational databases, providing a codebase foundation to its evolution [Herrera and Felinto 2012].

Assuming that information systems evolution is directly related on how hard is its decomposability [Brodie and Stonebraker 1995], a good migration strategy should be focused on turning a big and monolithic solution into smaller pieces of software, which could be easily managed, tested and atomically changed over the time, however the qualitative features that determine such level of success on this approach are not explicit and needs a specific study on the complexity carried by legacy solution [Sneed 1995].

In order to lead a data migration, there was developed a methodology concerned on it and the need for parallel operation of the legacy and target systems during migration was discussed in [Wu et al. 1997]. We advocate that a gradual migration model that should retains both systems running until the whole process is done, as a secure and reasonable choice. System evolution is covered as re-engineering and continuously improvements by a stream of incremental changes [Warren and Ransom 2002].

There was developed a middleware framework that links a new Web-based user interface with a wrapped Legacy Information System, as an incremental migration strategy based on re-engineering of user interface [De Lucia et al. 2008].

Reusing existing concepts of old methods allowed us to re-engineering them in new methods while some server regions was adapted for WAN data interchanging through distributed middleware adoption [Brinkkemper 1996, Herrera and Felinto 2012]. In a similar approach, a resource access middleware was developed considering a 3-layer architecture to wrap legacy scientific application to grid service [Lu et al. 2005].

There was presented a survey of key approaches to integrate and/or transform legacy applications into services to participate in an enterprise-wide SOA [Erradi et al. 2006]. Also, a two phases approach integrates legacy systems into Web Services [Parsa and Ghods 2008] was developed nearby the period of a case study on modernizing large scale systems to SOA was made publicly [Vemuri 2008].

A descriptive case study from the same industry segment as our work was fits in, was presented and covers aspects on re-engineering and evolution when Service Oriented Architecture (SOA) is adopted [Nasr et al. 2010]. Migrating application features to SOA by black-box approach based on multi-agent system was presented [Fayçal et al. 2010] and we developed auxiliary tools which essentially has the same influences.

A management decision model was presented for migration versus integration applied to SOA, taking care of combined strategic factors and cost-benefit analysis for key-decisions [Umar and Zordan 2009]. Architecture reconstruction to support system modernization through identification and reuse of legacy components as services in a SOA is shown along an industry case study [O'Brien et al. 2005].

Our testing workbench evolved in a sense that affected server was handled as a blackbox, in order to ensure cohesion over architectural codebase changes. A similar approach was used in [Zhang et al. 2008] while migrating another application class. Testing on SOA is discussed by means of complexity they could eventually impose, as its location is no longer tiered to a single organization domain [Lewis and Smith 2007, Canfora et al. 2008].

3. Overall Environment

The Service Provider domain was composed by dedicated hardware under a single physical cell and hosted heterogeneous systems communicating themselves by means of open and proprietary protocols.

Overall environment is show by Figure 1, where mobile devices send real time telemetry and geolocation data collected by sensors, from A endpoint. Customer Application Servers were located in apparted infrastructures, performing monitoring and control jobs over the vehicle fleet, so we must consider B as a multi-endpoint.

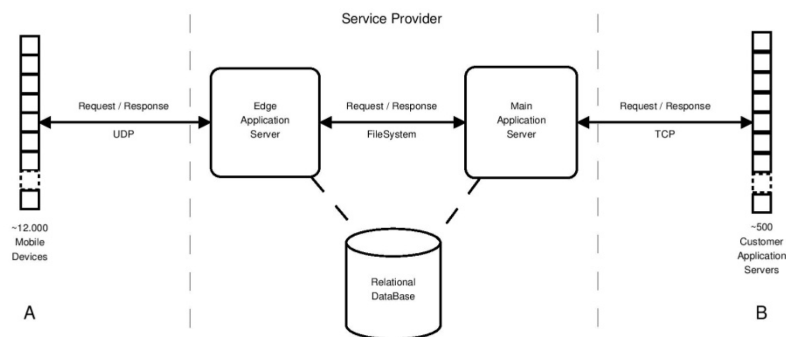


Figure 1. Main structures and its joint points.

There was an in-house message exchanging system between $\approx 12,000$ devices and Edge Application Server (EAS). At least one message per device is delivered within a 90 seconds interval. Using Global System for Mobile Communications (GSM) or General Packet Radio Service (GPRS) means over Transport Control Protocol (TCP) has lead to eventual communication breakdowns, motivating the development of a protocol featuring checksum control and ordered delivery, over User Datagram Protocol (UDP). This strategy was an effective and lightweight approach when compared to previous one, with a low rate of packet loss and communication failures (both bellow $\approx 0.05\%$).

The relationship between the EAS and a single device is shown by the Figure 2, where each device has an internal bus, able to communicates with sensors and actuators which is connected to. EAS retains a complete in-memory states mapping of sensors and actuators and if it does not get changed, in about 10 minutes, it should be persisted into Relational Database (RDB).

Based on the nature of interchanged messages between Mobile Devices, Service Provider and Customer Application Servers (CAS), integration data is generated in plain text files, which would be consumed later and, if needed, key-states of entire solution could be modified into EAS reserved memory and then persisted to RDB.

The Main Application Server (MAS), consumes the available information through a Shared File System (SFS), as shown in Figure 3, sending a bunch of pre-processed files to CAS, which would be used later on building operational reports to its end customers. All the communications between MAS and about 500 CAS was given by means of a plain text protocol, authenticated, cryptographed and TCP based.

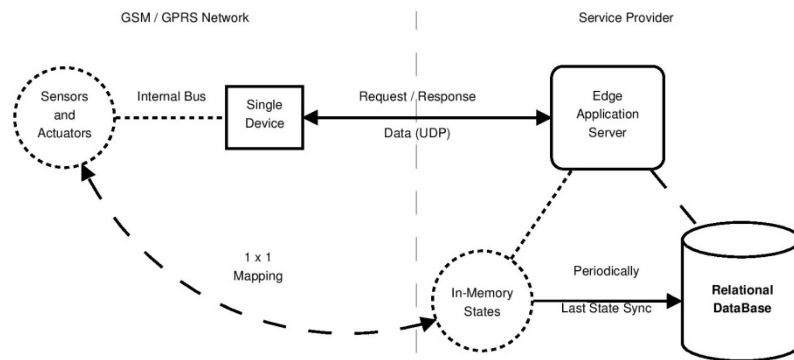


Figure 2. Relationship between the Service Provider and a Single Device.

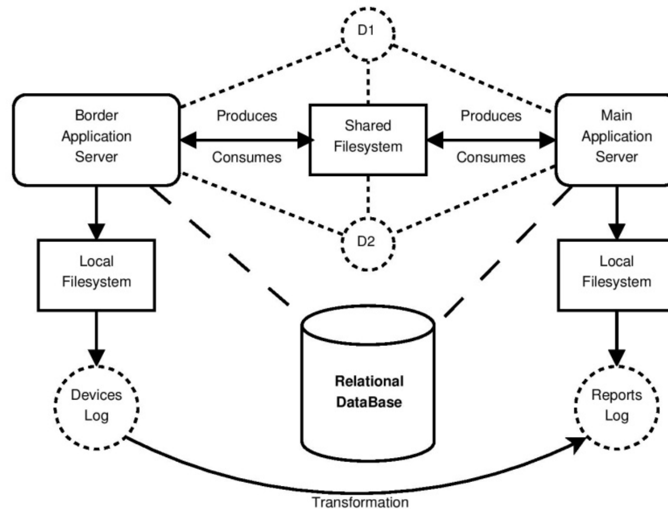


Figure 3. Relationship beteween EAS and MAS, both located at the Services Provider infrastructure.

In order to get integrated to whole solution, each customer should maintain an Application Server under its own infrastructure. It was charged on receiving and aggregating all vehicle fleet data, which was previously designated as its responsibility. The Figure 4 shows how this approach could lead us to an distributed data storage of pre-processed informations, by distinct business centers, once the raw informations are stored on Services Provider side, waiting for data recovery needs that could eventually be requested by some customers, which are limited up to 3 years period.

We can classify all the communications as bi-directional, in other words, the end customer can interact with its vehicle fleet, intermediated by these mechanisms, sending commands that goes from requesting specific data and then reconfiguring specific devices,

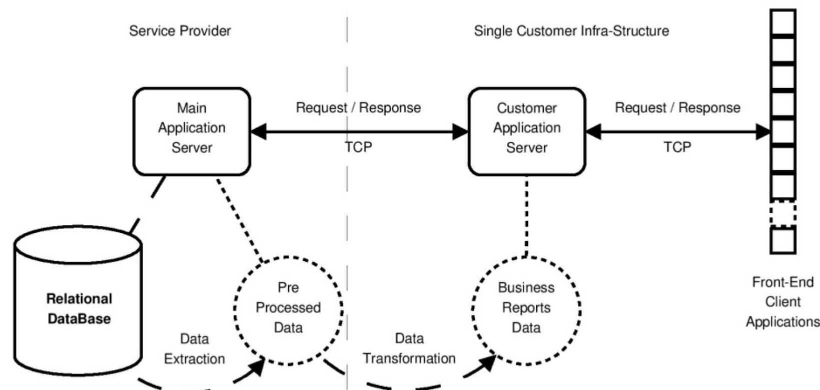


Figure 4. Relationship between the Services Provider and a Single Customer Infra-Structure.

to start actuators which are physically attached to the monitored vehicles. The SFS and the RDB could be changed during this process, and therefore, the associated data flow is variable according to the actions taken by its end users and received data from devices that would be later being processed by BAS.

4. Singular Point of Failure

The main characteristics that lead us to develop a distributed middleware in the past [Herrera and Felinto 2012], was the presence of a data-structure which is essential to EAS working: an operations buffer that should be taken into RDB. Once the service, by any reason, was abruptly broken, such buffer was lost and, consequently, the related operations to missed SQL queries could cause data inconsistencies.

There were threads made for receiving and dealing with arrived data from both ends of BAS, either from mobile devices or customers. We have found 2 failures at the non-transactional employed flow which, together with the stateful nature of crucial data being temporary stored in memory, could cause inconsistencies in durable-state of whole solution in case outages being occurring within key-steps. The flow can be described by the following main states:

1. The first state, Idle, means a waiting period before any checking could be done over EAS received data. It can be configured and by default is set up to 100 ms.
2. Having the Idle period passed by, a Polling is done in order to catch new arrived messages, despite where did they came from. If there are no messages received, the buffer is empty and then the thread returns to Idle state.
3. If a message was received, it will be processed and then got its results written into RDB. Whether the Acknowledge (ACK) is unnecessary, then the messaging process is finished and the thread returns to Idle state.
4. If a received message needs ACK after its results were written into RDB, then a message is sent with a list of actions made over some request and the messaging process is finished, so the thread returns to Idle state.

5. In case of failures within the system fragility period, an inconsistency will be generated in the whole durable-state. This occurrence demands maintenances and could be sporadically found, ever with outages due abruptly crashes.

It is possible to abstract the described flow of messages handling, as shown by Figure 5. Basically, when an end point does not receives the right acknowledge of some written data, a singular point of failure is reached. So on, a set of discrepancies might be found on virtual data mapping, which corresponds to real world states at several levels of the whole solution, depending solely on the criticality of the lost data and what system modules such consistent data affects.

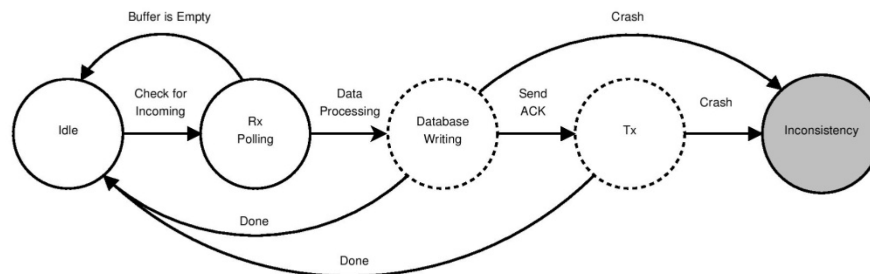


Figure 5. States Machine illustrating an inconsistency upon EAS outages.

Suppose that devices reconfiguration were received by EAS, it would be forwarded through the network to devices, after a properly handling. Once the targets have stated that they properly received and applied the configurations, their related statuses should be persisted to RDB. If, from this time, an outage occurs, the non-transactional nature of the process at both ends, would certainly lead us to an inconsistency.

5. Developed Migration Model

Aiming uninterrupted service providing, we conceived a migration strategy able to turn the Service Provider current architecture into a fault tolerant one. In order to overcome problems such as Hardware provisioning and solution continuous evolving, we used an environment that is solely based on Cloud Computing technologies, understood as Infrastructure as a Service (IaaS) and provided by Amazon Web Services™(AWS).

In order to accomplish such goal with cohesion and security, a smooth migration was planned over the monitored devices. Consequently, the end users base were also gradually migrated. The chosen devices groups, for each migration step, were taken according to complexity level of operations. Each numbered migration step is related to Mobile Devices features and is ordered from less to most critical ones:

1. Only provides geolocation queries.
2. Receives remote reconfigurations on its periodic reports.
3. Retains behavioral changes on its actuators, by remotes reconfigurations.
4. Generates real time critical alerts such as vehicles violations or requested helps.

The Figure 6 relates the percentage of devices and customers that were affected in each migration step. All devices and end users base were migrated from an environment

which was susceptible to failures and ran over obsolete Hardware, to another one that is fully elastic and fault tolerant. However, the fault tolerance itself, could not be achieved only with the full stack of applications being moved to such environment, but it was needed that this Software could be deployed to multiple instances and being able to finish some operations that were running into another machine which suffered from outage.

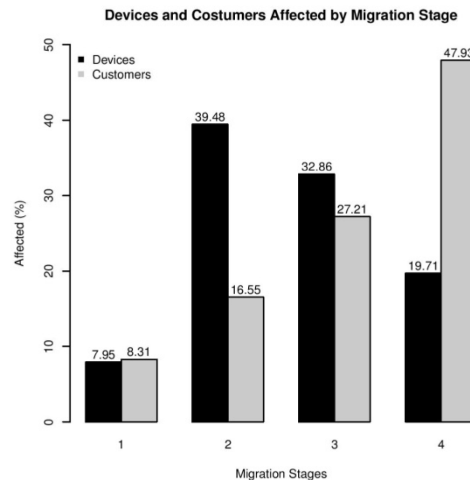


Figure 6. Impact of each migration step on devices and customers.

In order to build the idealized migration model and reach the fault tolerance, it was developed a middleware (Herrera and Felinto 2012) that was able to intermediate the communication process with RDB, storing all the stateful data from EAS into an in-memory grid. It enabled that multiple instances of the same daemon being able to continue running operations from another instances that for any reason were abruptly interrupted.

An overall solution overview, counting on proposed improvements, is portrayed by the Figure 7, where the distributed architecture is shown, while still be able to interchange data with legacy infrastructure. The main idea consists on allowing all the migration steps could occur such way that the services providing would not be shutdown.

The Cloud infrastructure received a modified version of BAS, where it is possible to do a linkage between multiple instances of the same kind, so eventual crashes would not lead to unavailability. While the migrations were occurring, the responsibility over devices was transferred from the 1st RDB to 2nd RDB, once all devices should be reporting all informations with Cloud infrastructure as its main reference.

During the migrations steps, there were several updates made on EAS codebase. To ensure that no feature gets broken, there were developed two simulators, able to reproduce the behavior of end customers and mobile devices in everyday operations. The Figure 8 shows the simulated environment working, where EAS is handled as a blackbox

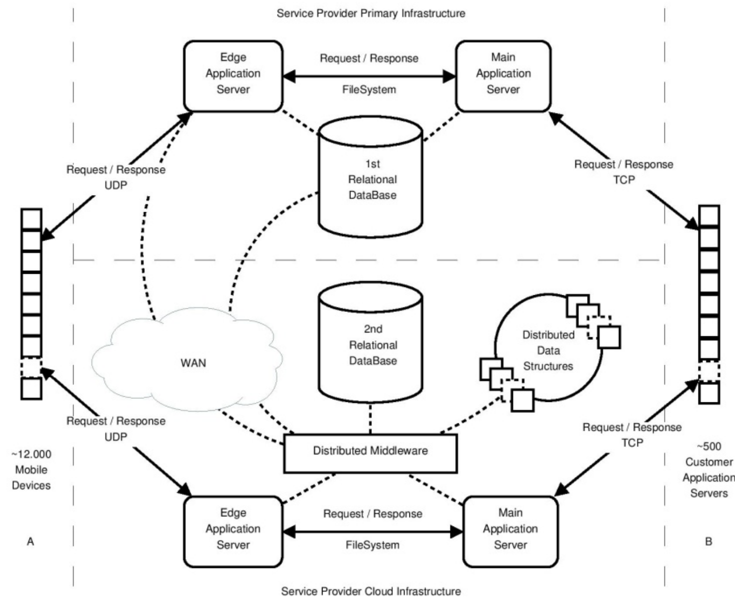


Figure 7. The improved environment, communicating with LIS over the WAN and making use of distributed middleware.

that is able to communicates with both customers and devices simulator. The requests set was chosen such that they fully cover the messages variety, which can occur in production.

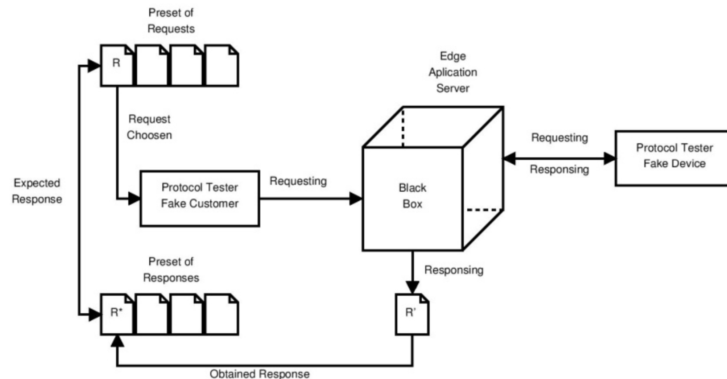


Figure 8. Devices and Customers simulation, handling EAS as a blackbox. The received responses set should match to the expected responses in order the codebase changes be accepted.

Each chosen request R, owns an expected $R*$ response, previously selected as a mean of acceptance. Once the protocol tester has selected any R message, then it is being fired against the blackboxed server, generating a response R' that is received and

compared with its related expected response $R*$.

If both R' and $R*$ are equals, it means that the developed feature does not affects negatively the server behavior and then, we can accept that codebase changes. However, whether the the comparison lead us to different results, the proposed changes must me rejected and an accurate analysis of R' could provide us traces on what is wrong with the evaluated implementation. Once these improvements were aggregated to the protocol testing process, we could ensure that all known behaviors would be retained while new features could be continuously added with security.

Since we developed a distributed solution with several daemons, we needed to ensure the tests process during the architectural changes conceiving could be agile. Thus, we developed a set of scripts that are able to setup a test environment with the applications full stack deployed into and ready to use. The main characteristics found in this script set which turned it into a reliable tool during codebase changing are:

- Ordered Startup and Shutdown: By means of configuration, we could set precedence between daemons, once there are productors and consumers in entire solution. Special parameters passing were allowed for bootstraping, such as activation of remotely monitoring on internal application states.
- Multiple hosts operations: Daemons can be activated and deactivated properly in different nodes along the network, according to test nature.
- Real-time log watching: Failures could be found by daemon alerts within log files, so when some miss behavior is detected upon previously stated patterns, we could setup stop conditions or else we can monitor for some expected behavior, in order to signal that whole operations are fully cohese and then could be continued.
- Ensure the full stack is properly running: It imposes the condition that no daemon could be solely ran. So, all the data flow is ensured and no end-point will fail to communicates with another ones due secondary reasons, as not being properly started.

6. Future Works

During the development of this work, we could state some points we believe to deserve more in depth attention. Thus, according to our past experiences it lead us to list them as potential contributions on the legacy and distributed systems fields of knowledge.

Considering the solution evolved in the sense of being able to operate with multiple instances of the fullstack, simultaneously, we suggest a measurement on how horizontal the solution could be scaled. We expect such rate reveal us to what extend the load applied over the solution could be relieved, simply by starting clone instances, so they can co-operate while doing load balancing themselves.

The auxiliary simulators that were developed, especially to migration process, could be extended to become generic enough, allowing its complete customization for another LIS testing purposes. This way, in the future they can be known as open tools for testing and protocols validation, enabling a wider range of blackbox tests on general purposes servers.

The set of scripts that handles startup and shutdown over the daemons, are found in a flexibility level enough such as it can be used in heterogeneous applications of several

kinds and, therefore, maybe it could be widely used by Platform as a Service, Cloud Computing based industry, because it allows the management of different services located in apparted hosts, building in a completely distributed way the requested setup.

7. Conclusion

The importance of this Case Study, was primarily concerned on addressing how a viable transition to a modern infrastructure could be made smoothly, while providing means for secure evolving of the whole solution, with its behavior fully validated on each iteration of migration steps, where brand new features could be developed and integrated to main codebase of the project.

The imposed fragility over the full operations chain, was essentially related to the stateful nature of critical data, that were in-memory stored. This phenomenon was successfully overcame after the aggregation of distributed middleware, which uses distributed data structures that resembles queues to handle the operations demand that made changes on RDB and memory states.

Conceptually, such data structures describe a query buffer such that they are fully compatible themselves, eliminating problems on storing batch operations into the memory from a single server, affecting them end-to-end in a non-transactional flow. Thus, even there are running operations, it could be immediately resumed by any EAS active instance, no matter what reason lead other service nodes to outages.

To ensure that EAS behavior has not been changed during the development and migration process, protocols testers were made featuring output validation by means of known and expected responses, attached to both ends of servers, which were able to be handled as different blackboxes on each planned migration step, securing that the solution could evolve without services disruption neither inclusion of bugs on its codebase.

References

Bisbal, J., Lawless, D., Wu, B., and Grimson, J. (1999). Legacy information systems: issues and directions. *IEEE Software*, 16(5):103 –111.

Brinkkemper, S. (1996). Method engineering: Engineering of information systems development methods and tools. *Information and Software Technology*, 38(4):275–280.

Brodie, M. and Stonebraker, M. (1995). *Migrating legacy systems: gateways, interfaces & the incremental approach*. Morgan Kaufmann series in data management systems. Morgan Kaufmann Publishers.

Canfora, G., Fasolino, A. R., Frattolillo, G., and Tramontana, P. (2008). A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. *Journal of Systems and Software*, 81(4):463 – 480. Selected papers from the 10th Conference on Software Maintenance and Reengineering (CSMR 2006).

De Lucia, A., Francese, R., Scanniello, G., and Tortora, G. (2008). Developing legacy system migration methods and tools for technology transfer. *Softw. Pract. Exper.*, 38(13):1333–1364.

Erradi, A., Anand, S., and Kulkarni, N. (2006). Evaluation of strategies for integrating legacy applications as services in a service oriented architecture. *IEEE International Conference on Services Computing*, pages 257–260.

Fayçal, H., Habiba, D., and Hakima, M. (2010). Integrating legacy systems in a soa using an agent based approach for information system agility. *International Conference on Machine and Web Intelligence (ICMWI)*, pages 338 –343.

Herrera, R. and Felinto, A. (2012). A distributed, multi-staged, high-throughput middleware for relational databases. *IIX Simpósio Brasileiro de Sistemas de Informação (SBSI)*.

Lewis, G. and Smith, D. B. (2007). Developing realistic approaches for the migration of legacy components to service-oriented architecture environments. *2nd International Conference on Trends in Enterprise Application Architecture*, pages 226–240.

Lu, F., Huang, H., Xu, Z., and Yu, H. (2005). A middleware for legacy application wrapper. *First International Conference on Semantics, Knowledge and Grid*, pages 47–.

Nasr, K. A., Gross, H.-G., and van Deursen, A. (2010). Adopting and evaluating service oriented architecture in industry. *14th European Conference on Software Maintenance and Reengineering*, pages 11–20.

O'Brien, L., Smith, D., and Lewis, G. (2005). Supporting migration to services using software architecture reconstruction. *13th IEEE International Workshop on Software Technology and Engineering Practice*, pages 81–91.

Parsa, S. and Ghods, L. (2008). A new approach to wrap legacy programs into web services. *11th International Conference on Computer and Information Technology (ICCIT)*, pages 442 –447.

Sneed, H. M. (1995). Planning the reengineering of legacy systems. *IEEE Softw.*, 12(1):24–34.

Umar, A. and Zordan, A. (2009). Reengineering for service oriented architectures: A strategic decision model for integration versus migration. *J. Syst. Softw.*, 82(3):448–462.

Vemuri, P. (2008). Modernizing a legacy system to soa - feature analysis approach. *IEEE Region 10 Conference (TENCON)*, pages 1 –6.

Warren, I. and Ransom, J. (2002). Renaissance: a method to support software system evolution. *26th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 415 – 420.

Wu, B., Lawless, D., Bisbal, J., Grimson, J., Wade, V., O'Sullivan, D., and Richardson, R. (1997). Legacy systems migration-a method and its tool-kit framework. *Asia Pacific Software Engineering Conference and International Computer Science Conference (APSEC / ICSC)*, pages 312 –320.

Zhang, B., Bao, L., Zhou, R., Hu, S., and Chen, P. (2008). A black-box strategy to migrate gui-based legacy systems to web services. *IEEE International Symposium on Service-Oriented System Engineering*, pages 25–31.

TRABALHOS PUBLICADOS PELO AUTOR

1. Herrera, R. P. and Felinto, A. S. (2012). **A distributed, multi-staged, high-throughput middleware for relational databases**. VIII Simpósio Brasileiro de Sistemas de Informação (SBSI).

2. Herrera, R. P. and Felinto, A. S. (2014). **A Case Study on Legacy Information Systems Migration: A Fault Tolerant Approach**. X Simpósio Brasileiro de Sistemas de Informação (SBSI).