

UNIVERSIDADE DA CALIFÓRNIA,  
IRVINA

Estilos de arquitetura e o design de arquiteturas de software baseadas em rede

DISSERTAÇÃO

apresentados em satisfação parcial dos requisitos para o grau de

DOUTOR DE FILOSOFIA

em Ciência da Informação e Computação

por

Roy Thomas Fielding

Comissão de Dissertação:

Professor Richard N. Taylor, Presidente  
Professor Mark S. Ackerman  
Professor David S. Rosenblum

2000

© Roy Thomas Fielding, 2000.

Todos os direitos reservados.

A dissertação de Roy Thomas Fielding foi aprovada e é  
aceitável em qualidade e forma para publicação  
em microfilme:

---

---

---

Presidente do Comitê

Universidade da Califórnia, Irvine  
2000

## DEDICAÇÃO

Para

meus pais,

Pete e Kathleen Fielding,

que tornaram tudo isso possível,  
por seu inesgotável incentivo e paciência.

E também para

Tim Berners-Lee,

por tornar a World Wide Web um projeto aberto e colaborativo.

*O que é a vida?*

*É o clarão de um vaga-lume na noite.*

*É a respiração de um búfalo no inverno.*

*É a pequena sombra que corre pela grama e se perde no  
pôr do sol.*

— As últimas palavras de Crowfoot (1890), guerreiro e orador Blackfoot.

*Quase todo mundo se sente em paz com a natureza: ouvindo as ondas do mar  
contra a costa, em um lago tranquilo, em um campo de grama, em uma  
charneca soprada pelo vento. Um dia, quando tivermos aprendido de novo o  
caminho atemporal, sentiremos o mesmo por nossas cidades, e sentiremos  
tanta paz nelas quanto hoje, caminhando à beira-mar, ou estendidos na grama  
alta de um Prado.*

— Christopher Alexander, The Timeless Way of Building (1979)

## ÍNDICE

	Página
LISTA DE FIGURAS .....	vi
LISTA DE MESAS.....	vii
AGRADECIMENTOS .....	viii
CURRICULUM VITAE.....	x
RESUMO DA DISSERTAÇÃO .....	xvi
INTRODUÇÃO .....	1
CAPÍTULO 1: Arquitetura de Software .....	5
1.1 Abstração em tempo de execução .....	.....
5 1.2 Elementos .....	7
1.3 Configurações .....	12 1.4
Propriedades .....	12 1.5
Estilos.....	13 1.6
Padrões e Linguagens de Padrões .....	16 1.7
Visualizações .....	17
1.8 Trabalho Relacionado .....	.....
18 1.9 Resumo .....	23
CAPÍTULO 2: Arquiteturas de aplicativos baseados em rede.....	24 2.1
Escopo.....	.....
2.2 Avaliando o Design de Arquiteturas de Aplicativos .....	.....
2.3 Propriedades arquitetônicas de interesse chave .....	28
2.4 Resumo .....	37

CAPÍTULO 3: Estilos de arquitetura baseados em rede .....	38	3.1
Metodologia de classificação .....	38	3.2
Estilos de fluxo de dados .....	41	3.3
Estilos de replicação .....	43	3.4 Estilos
Hierárquicos.....	45	3.5 Estilos de
código móvel .....	50	3.6 Estilos ponto
a ponto.....	55	3.7
Limitações .....	59	3.8 Trabalho
Relacionado .....	60	3.9
Resumo.....		64
 CAPÍTULO 4: Projetando a Arquitetura da Web: Problemas e Insights.....	66	
4.1 Requisitos do Domínio do Aplicativo WWW .....	66	4.2
Problema .....	71	4.3
Abordagem.....	72	4.4
Resumo.....		75
 CAPÍTULO 5: Transferência de Estado Representacional (REST) .....	76	5.1
Derivando REST .....	76	5.2
Elementos de Arquitetura REST .....	86	5.3
Vistas Arquiteturais REST .....	97	5.4
Trabalho Relacionado .....	103	
5.5 Resumo .....		105
 CAPÍTULO 6: Experiência e Avaliação .....	107	
6.1 Padronizando a Web.....	107	6.2 DESCANSO Aplicado
ao URI.....	109	6.3 REST Aplicado a
HTTP.....	116	6.4 Transferência de
Tecnologia.....	134	6.5 Lições de
Arquitetura .....	138	6.6
Resumo .....		147
 CONCLUSÕES .....		148
 REFERÊNCIAS.....		152

## LISTA DE FIGURAS

	Página
Figura 5-1. Estilo nulo	77
Figura 5-2. Servidor cliente	78
Figura 5-3. Cliente-Stateless-Servidor	78
Figura 5-4. Client-Cache-Stateless-Server	80
Figura 5-5. Diagrama de arquitetura inicial da WWW	81
Figura 5-6. Uniform-Client-Cache-Stateless-Server	82
Figura 5-7. Uniform-Layered-Client-Cache-Stateless-Server	83
Figura 5-8. DESCANSO	84
Figura 5-9. Derivação REST por restrições de estilo	85
Figura 5-10. Visualização de processo de uma arquitetura baseada em REST	98

# LISTA DE MESAS

	Página
Tabela 3-1. Avaliação de estilos de fluxo de dados para hipermídia baseada em rede	41
Tabela 3-2. Avaliação de estilos de replicação para hipermídia baseada em rede	43
Tabela 3-3. Avaliação de estilos hierárquicos para hipermídia baseada em rede	45
Tabela 3-4. Avaliação de estilos de código móvel para hipermídia baseada em rede	51
Tabela 3-5. Avaliação de estilos peer-to-peer para hipermídia baseada em rede	55
Tabela 3-6. Resumo da avaliação	65
Tabela 5-1. Elementos de dados REST	88
Tabela 5-2. Conectores REST	93
Tabela 5-3. Componentes REST	96

## AGRADECIMENTOS

Foi um grande prazer trabalhar com o corpo docente, funcionários e alunos da Universidade da Califórnia, Irvine, durante meu período como estudante de doutorado. Este trabalho nunca teria sido possível se não fosse pela liberdade que me foi dada para perseguir meus próprios interesses de pesquisa, em grande parte graças à gentileza e orientação considerável fornecida por Dick Taylor, meu conselheiro de longa data e presidente do comitê. Mark Ackerman também merece muitos agradecimentos, pois foi sua aula sobre serviços de informação distribuídos em 1993 que me apresentou à comunidade de desenvolvedores da Web e levou a todo o trabalho de design descrito nesta dissertação. Da mesma forma, foi o trabalho de David Rosenblum em arquiteturas de software em escala de Internet que me convenceu a pensar em minha própria pesquisa em termos de arquitetura, em vez de simplesmente hipermídia ou design de protocolo de camada de aplicação.

O estilo de arquitetura da Web foi desenvolvido iterativamente durante um período de seis anos, mas principalmente durante os primeiros seis meses de 1995. Ele foi influenciado por inúmeras discussões com pesquisadores da UCI, funcionários do World Wide Web Consortium (W3C) e engenheiros dos Grupos de trabalho HTTP e URI da Internet Engineering Taskforce (IETF). Gostaria de agradecer particularmente a Tim Berners-Lee, Henrik Frystyk Nielsen, Dan Connolly, Dave Raggett, Rohit Khare, Jim Whitehead, Larry Masinter e Dan LaLiberte por muitas conversas ponderadas sobre a natureza e os objetivos da arquitetura WWW. Também gostaria de agradecer a Ken Anderson por sua visão sobre a comunidade de hipertexto aberto e por abrir caminho para a pesquisa em hipermídia na UCI. Agradeço também aos meus colegas pesquisadores de arquitetura da UCI, que terminaram antes de mim, incluindo Peyman Oreizy, Neno Medvidovic, Jason Robbins e David Hilbert.

A arquitetura da Web é baseada no trabalho colaborativo de dezenas de desenvolvedores de software voluntários, muitos dos quais raramente recebem o crédito que merecem por serem pioneiros da Web antes de se tornar um fenômeno comercial. Além do pessoal do W3C acima, o reconhecimento deve ir para os desenvolvedores de servidores que permitiram grande parte do rápido crescimento da Web em 1993-1994 (mais ainda, acredito, do que os navegadores). Isso inclui Rob McCool (NCSA httpd), Ari Luotonen (CERN httpd/proxy) e Tony Sanders (Plexus). Obrigado também ao "Sr. Content", Kevin Hughes, por ser o primeiro a implementar a maioria das formas interessantes de mostrar informações na Web além do hipertexto. Os primeiros desenvolvedores de clientes também merecem agradecimentos: Nicola Pellow (modo de linha), Pei Wei (Viola), Tony Johnson (Midas), Lou Montulli (Lynx), Bill Perry (W3) e Marc Andreessen e Eric Bina (Mosaic for X). Finalmente, meus agradecimentos pessoais vão para meus colaboradores do libwww-perl, Oscar Nierstrasz, Martijn Koster e Gisle Aas. Felicidades!

A arquitetura moderna da Web ainda é definida mais pelo trabalho de voluntários individuais do que por qualquer empresa. O principal deles são os membros da Apache Software Foundation. Agradecimentos especiais a Robert S. Thau pelo design incrivelmente robusto do Shambhala que levou ao Apache 1.0, bem como por muitas discussões sobre extensões da Web desejáveis (e indesejáveis), a Dean Gaudet por me ensinar mais sobre avaliação detalhada do desempenho do sistema do que eu pensava Eu precisava saber, e para Alexei Kosut por ser o primeiro a implementar a maior parte do HTTP/1.1 no Apache. Agradecimentos adicionais ao restante dos fundadores do Grupo Apache, incluindo Brian Behlendorf, Rob Hartill, David Robinson, Cliff Skolnick, Randy Terbush e Andrew Wilson, por construir uma comunidade da qual todos podemos nos orgulhar e mudar o mundo mais uma vez.

Também gostaria de agradecer a todas as pessoas da eBuilt que fizeram dela um ótimo lugar para se trabalhar. Agradecimentos especiais aos quatro fundadores técnicos – Joe Lindsay, Phil Lindsay, Jim Hayes e Joe Manna – por criar (e defender) uma cultura que torna a engenharia divertida. Obrigado também a Mike Dewey, Jeff Lenardson, Charlie Bunten e Ted Lavoie, por tornarem possível ganhar dinheiro enquanto se divertem. E um agradecimento especial a Linda Dailing, por ser a cola que nos mantém unidos.

Obrigado e boa sorte para a equipe da Endeavors Technology, incluindo Greg Bolcer, Clay Cover, Art Hitomi e Peter Kammer. Por fim, gostaria de agradecer às minhas três musas – Laura, Nikki e Ling – por sua inspiração ao escrever esta dissertação.

Em grande parte, minha pesquisa de dissertação foi patrocinada pela Agência de Projetos de Pesquisa Avançada de Defesa e pelo Laboratório de Pesquisa da Força Aérea, Comando de Material da Força Aérea, USAF, sob o contrato número F30602-97-2-0021. O governo dos EUA está autorizado a reproduzir e distribuir reimpressões para fins governamentais, independentemente de qualquer anotação de direitos autorais. As opiniões e conclusões aqui contidas são de responsabilidade dos autores e não devem ser interpretadas como necessariamente representando as políticas ou endossos oficiais, expressos ou implícitos, da Agência de Projetos de Pesquisa Avançada de Defesa, do Laboratório de Pesquisa da Força Aérea ou dos EUA Governo.

# CURRICULUM VITAE

Roy Thomas Fielding

## Educação

### Doutor em Filosofia (2000)

University of California, Irvine  
Information and Computer Science  
Institute of Software Research Orientador:  
Dr. Richard N. Taylor *Dissertação: Estilos de Arquitetura e o Projeto de Arquiteturas de Software Baseadas em Rede*

### Mestrado em Ciências (1993)

Universidade da Califórnia, Irvine  
Ciência da Informação e da Computação  
Ênfase Principal: Software

### Bacharel em Ciências (1988)

Universidade da Califórnia, Irvine  
Ciência da Informação e da Computação

## Experiência profissional

- 12/99 - Cientista-chefe, eBuilt, Inc., Irvine, Califórnia
- 3/99 - Presidente, The Apache Software Foundation
- 4/92 - 12/99 Estudante de Pós-Graduação Pesquisador, Instituto de Pesquisa de Software  
Universidade da Califórnia, Irvine
- 6/95 - 9/95 Visiting Scholar, World Wide Web Consortium (W3C)  
Laboratório de Ciência da Computação do MIT, Cambridge, Massachusetts
- 9/91 - 3/92 Assistente de Ensino  
ICS 121 - Introdução à Engenharia de Software  
ICS 125A - Projeto em Engenharia de Software  
Universidade da Califórnia, Irvine
- 11/89 - 6/91 Engenheiro de software  
ADC Kentrox, Inc., Portland, Oregon
- 7/88 - 8/89 Equipe Profissional (Engenheiro de Software)  
PRC Public Management Services, Inc., São Francisco, Califórnia
- 10/86 - 6/88 Programador/Analista  
Megadyne Information Systems, Inc., Santa Ana, Califórnia
- 6/84 - 9/86 Programador/Analista  
TRANSMAX, Inc., Santa Ana, Califórnia

## Publicações

### Artigos de periódicos referenciados

- [1] RT Fielding, EJ Whitehead, Jr., KM Anderson, GA Bolcer, P. Oreizy e RN Taylor. Desenvolvimento baseado na Web de Produtos de Informação Complexos. *Comunicações da ACM*, 41(8), agosto de 1998, pp. 84-92.
- [2] RT Fielding. Mantendo Infoestruturas de Hipertexto Distribuídas: Bem-vindo ao Web do MOMspider. *Computer Networks and ISDN Systems*, 27(2), novembro de 1994, pp. 193-204. (Revisão de [7] após seleção especial pelos árbitros.)

### Publicações da Conferência Referenciada

- [3] RT Fielding e RN Taylor. Princípios de Design da Arquitetura Web Moderna. In *Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000)*, Limerick, Irlanda, junho de 2000, pp. 407-416.
- [4] A. Mockus, RT Fielding e J. Herbsleb. Um estudo de caso de desenvolvimento de software de código aberto: o servidor Apache. In *Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000)*, Limerick, Irlanda, junho de 2000, pp. 263-272.
- [5] EJ Whitehead, Jr., RT Fielding e KM Anderson. Fusão da tecnologia WWW e Link Server: uma abordagem. In *Proceedings of the 2nd Workshop on Open Hypermedia Systems, Hypertext'96*, Washington, DC, março de 1996, pp. 81-86.
- [6] MS Ackerman e RT Fielding. Manutenção de acervo na Biblioteca Digital. In *Proceedings of Digital Libraries '95*, Austin, Texas, junho de 1995, pp. 39-48.
- [7] RT Fielding. Mantendo Infoestruturas de Hipertexto Distribuídas: Bem-vindo à Web do MOMspider. In *Proceedings of the First International World Wide Web Conference*, Genebra, Suíça, maio de 1994, pp. 147-156.

### Padrões industriais

- [8] RT Fielding, J. Gettys, JC Mogul, HF Nielsen, L. Masinter, P. Leach e T. Berners-Lee. Protocolo de transferência de hipertexto — HTTP/1.1. *Internet Draft Standard RFC 2616*, junho de 1999. [Obsoletes RFC 2068, janeiro de 1997.]
- [9] T. Berners-Lee, RT Fielding e L. Masinter. Identificadores Uniformes de Recursos (URI): Sintaxe Genérica. *Rascunho da Internet RFC 2396*, agosto de 1998.
- [10] J. Mogul, RT Fielding, J. Gettys e HF Frystyk. Uso e interpretação de números de versão HTTP. *Internet Informativa RFC 2145*, maio de 1997.
- [11] T. Berners-Lee, RT Fielding e HF Nielsen. Protocolo de Transferência de Hipertexto - HTTP/1.0. *Internet Informativa RFC 1945*, maio de 1996.
- [12] RT Fielding. Localizadores de recursos uniformes relativos. *Padrão proposto para Internet RFC 1808*, junho de 1995.

### **Artigos da indústria**

- [13] RT Fielding. Os segredos para o sucesso do Apache. *Linux Magazine*, 1(2), junho de 1999, págs. 29-71.
- [14] RT Fielding. Liderança Compartilhada no Projeto Apache. *Comunicações da ACM*, 42(4), abril de 1999, pp. 42-43.
- [15] RT Fielding e GE Kaiser. O Projeto Apache HTTP Server. *Internet IEEE Computing*, 1(4), julho-agosto de 1997, pp. 88-90.

### **Publicações Não Referenciadas**

- [16] RT Fielding. Estilos de arquitetura para aplicativos baseados em rede. Fase II Survey Paper, Departamento de Informação e Ciência da Computação, Universidade da Califórnia, Irvine, julho de 1999.
- [17] J. Grudin e RT Fielding. Grupo de Trabalho sobre Métodos e Processos de Design. In *Proceedings of the ICSE'94 Workshop on SE-HCI: Joint Research Issues*, Sorrento, Itália, maio de 1994. Publicado em “Engenharia de Software e Interação Humano-Computador”, Springer-Verlag LNCS, vol. 896, 1995, pp. 4-8.
- [18] RT Fielding. Proposta GET Condisional para Cache HTTP. *Publicado na WWW*, em janeiro de 1994.

### **Pacotes de software publicados**

- [19] *Apache httpd*. O servidor Apache HTTP é o software de servidor Web mais popular do mundo, usado por mais de 65% de todos os sites públicos da Internet em julho de 2000.
- [20] *libwww-perl*. Uma biblioteca de pacotes Perl4 que fornece uma interface de programação simples e consistente para a World Wide Web.
- [21] *Cebolas*. Uma biblioteca de pacotes Ada95 que fornece fluxos empilháveis eficientes capacidade para E/S de rede e sistema de arquivos.
- [22] *MOMaranha*. O MOMspider é um robô da web para fornecer manutenção multi-proprietário de infoestruturas de hipertexto distribuídas.
- [23] *wwwstat*. Um conjunto de utilitários para pesquisar e resumir o acesso ao servidor httpd WWW logs e auxiliando outras tarefas do webmaster.

### **Apresentações formais**

- [1] *Estado de Apache*. Convenção de Software de Código Aberto O'Reilly, Monterey, CA, julho 2000.
- [2] *Design de Princípios da Arquitetura Web Moderna*. Conferência Internacional de 2000 em Engenharia de Software, Limerick, Irlanda, junho de 2000.
- [3] *HTTP e Apache*. ApacheCon 2000, Orlando, Flórida, março de 2000.

- [4] *A Comunicação Humana e o Desenho da Arquitetura Web Moderna.* WebNet World Conference on the WWW and the Internet (WebNet 99), Honolulu, HI, outubro de 1999.
- [5] *A Apache Software Foundation.* Computer & Communications Industry Association, Autumn Members Meeting, Dallas, TX, setembro de 1999.
- [6] *Identificadores Uniformes de Recursos.* O Workshop sobre Tecnologia em Escala da Internet (TWIST 99), Irvine, CA, agosto de 1999.
- [7] *Apache: Passado, Presente e Futuro.* Web Design World, Seattle, WA, julho de 1999.
- [8] *Relatório de progresso sobre o Apache.* ZD Open Source Forum, Austin, TX, junho de 1999.
- [9] *Código aberto, estilo Apache: lições aprendidas com software colaborativo Desenvolvimento.* Second Open Source and Community Licensing Summit, San Jose, CA, março de 1999.
- [10] *O Projeto Apache HTTP Server: Lições aprendidas com o software colaborativo.* AT&T Labs — Pesquisa, Folsom Park, NJ, outubro de 1998.
- [11] *Desenvolvimento Colaborativo de Software: Juntando-se ao Projeto Apache.* ApacheCon '98, San Francisco, CA, outubro de 1998.
- [12] *Transferência de Estado Representacional: Um Estilo Arquitetônico para Interação Hipermídia Distribuída.* Microsoft Research, Redmond, WA, maio de 1998.
- [13] *O Grupo Apache: Um Estudo de Caso de Colaboração na Internet e Comunidades.* Seminário WWW de Ciências Sociais da UC Irvine, Irvine, CA, maio de 1997.
- [14] *WebSoft: Construindo um Ambiente Global de Engenharia de Software.* Workshop sobre Engenharia de Software (on) the World Wide Web, 1997 International Conference on Software Engineering (ICSE 97), Boston, MA, maio de 1997.
- [15] *Evolução do Protocolo de Transferência de Hipertexto.* Simpósio de Pesquisa ICS, Irvine, CA, janeiro de 1997.
- [16] *Infraestrutura e evolução da World Wide Web.* Simpósio IRUS SETT sobre WIRED: World Wide Web and the Internet, Irvine, CA, maio de 1996.
- [17] *Cache HTTP.* Quinta Conferência Internacional da World Wide Web (WW5), Paris, França, maio de 1996.
- [18] *A Importância da Infraestrutura da World Wide Web.* Simpósio de Software da Califórnia (CSS '96), Los Angeles, CA, abril de 1996.
- [19] *Software da World Wide Web: Uma visão privilegiada.* Mesa Redonda da Área da Baía da IRUS (BART), Palo Alto, CA, janeiro de 1996.
- [20] *libwww-Perl4 e libwww-Ada95.* Quarta World Wide Web Internacional Conferência, Boston, MA, dezembro de 1995.
- [21] *Protocolo de transferência de hipertexto — HTTP/1.x.* Quarta World Wide Web Internacional Conferência, Boston, MA, dezembro de 1995.

- [22] Protocolo de *transferência de hipertexto — HTTP/1.x*. HTTP Working Group, 34th Internet Engineering Taskforce Meeting, Dallas, TX, dezembro de 1995.
- [23] Protocolo de *transferência de hipertexto — HTTP/1.0 e HTTP/1.1*. Grupo de Trabalho HTTP, 32ª Reunião da Força-Tarefa de Engenharia da Internet, Danvers, MA, abril de 1995.
- [24] *WWW Developer Starter Kits para Perl*. WebWorld Conference, Orlando, FL, janeiro de 1995, e Santa Clara, CA, abril de 1995.
- [25] *Localizadores de recursos uniformes relativos*. URI Working Group, 31st Internet Engineering Taskforce Meeting, San Jose, CA, dezembro de 1994.
- [26] Protocolo de *transferência de hipertexto — HTTP/1.0*. HTTP BOF, 31st Internet Engineering Taskforce Meeting, San Jose, CA, dezembro de 1994.
- [27] *Atrás das Cortinas: Como a Web foi/é/será criada*. Seminário World Wide Web de Ciências Sociais da UC Irvine, Irvine, CA, outubro de 1995.
- [28] *Mantendo Infoestruturas de Hipertexto Distribuídas: Bem-vindo à Web do MOMspider*. Primeira Conferência Internacional da World Wide Web, Genebra, Suíça, maio de 1994.

### **Atividades profissionais**

- Webmaster, 1997 Conferência Internacional sobre Engenharia de Software (ICSE'97), Boston, maio de 1997.
- Presidente da Sessão HTTP, Quinta Conferência Internacional da World Wide Web (WWW5), Paris, França, maio de 1996.
- Presidente Birds-of-a-Feather e Presidente da Sessão, Quarta World Wide Web Internacional Conferência (WW4), Boston, dezembro de 1995.
- Estudante Voluntário, 17th International Conference on Software Engineering (ICSE 17), Seattle, junho de 1995.
- Estudante Voluntário, Segunda Conferência Internacional da World Wide Web (WW2), Chicago, outubro de 1994.
- Estudante Voluntário, 16th International Conference on Software Engineering (ICSE 16), Sorrento, Itália, abril de 1994.
- Co-fundador e membro, The Apache Group, 1995-presente.
- Fundador e arquiteto-chefe, projeto colaborativo libwww-perl, 1994-95.
- Representante do ICS, Associated Graduate Students Council, 1994-95.

### **Associações profissionais**

- A Apache Software Foundation
- Associação de Máquinas de Computação (ACM)
- Grupos de Interesse Especial da ACM em Engenharia de Software (SIGSOFT), Comunicações de Dados (SIGCOMM) e Groupware (SIGGROUP)

## **Honras, prêmios, bolsas**

2000	Appaloosa Award for Vision, O'Reilly Open Source 2000
2000	Melhor Aluno de Pós-Graduação, UCI Alumni Association
1999	Prêmio ACM Software System
1999	TR100: Top 100 jovens inovadores, MIT Technology Review
1991	Regent's Fellowship, Universidade da Califórnia
1988	Golden Key National Honor Society
1987	Lista de Honra do Reitor

## RESUMO DA DISSERTAÇÃO

Estilos de arquitetura e o design de arquiteturas de software baseadas em rede

por

Roy Thomas Fielding

Doutor em Filosofia em Ciência da Informação e Computação

Universidade da Califórnia, Irvine, 2000

Professor Richard N. Taylor, Presidente

A World Wide Web teve sucesso em grande parte porque sua arquitetura de software foi projetado para atender às necessidades de um sistema de hipermídia distribuído em escala da Internet. A Web foi desenvolvida iterativamente nos últimos dez anos por meio de uma série de modificações nos padrões que definem sua arquitetura. Para identificar esses aspectos da Web que precisava de melhorias e evitar modificações indesejáveis, um modelo para o era necessária uma arquitetura moderna da Web para orientar seu projeto, definição e implantação.

A pesquisa em arquitetura de software investiga métodos para determinar a melhor forma de partitionar um sistema, como os componentes se identificam e se comunicam entre si, como informação é comunicada, como os elementos de um sistema podem evoluir independentemente e como todos os itens acima podem ser descritos usando notações formais e informais. Meu trabalho é motivado pelo desejo de entender e avaliar o projeto arquitetônico de redes software aplicativo baseado em princípios de uso de restrições arquitetônicas, obter as propriedades funcionais, de desempenho e sociais desejadas de uma arquitetura. Um estilo arquitetônico é um conjunto nomeado e coordenado de restrições arquitetônicas.

Esta dissertação define um framework para entender a arquitetura de software via estilos arquitetônicos e demonstra como os estilos podem ser usados para guiar a arquitetura projeto de software aplicativo baseado em rede. Um levantamento de estilos arquitetônicos para aplicações baseadas em rede são usadas para classificar estilos de acordo com a arquitetura propriedades que eles induzem em uma arquitetura para hipermídia distribuída. Em seguida, apresento o O estilo arquitetônico Representational State Transfer (REST) e descreve como o REST tem sido usado para orientar o design e o desenvolvimento da arquitetura para a Web moderna.

REST enfatiza a escalabilidade de interações de componentes, generalidade de interfaces, implantação independente de componentes e componentes intermediários para reduzir latência de interação, reforçar a segurança e encapsular sistemas legados. eu descrevo o princípios de engenharia de software que orientam o REST e as restrições de interação escolhidas para reter esses princípios, contrastando-os com as restrições de outros estilos arquitetônicos.

Por fim, descrevo as lições aprendidas com a aplicação do REST ao design do Hypertext Padrões de Protocolo de Transferência e Identificador Uniforme de Recursos, e de seus implantação em software cliente e servidor Web.

## INTRODUÇÃO

*Com licença... você disse 'facas'?*

— City Gent #1 (Michael Palin), The Architects Sketch [111]

Como previsto por Perry e Wolf [105], a arquitetura de software tem sido um ponto focal para pesquisa em engenharia de software na década de 1990. A complexidade dos sistemas de software modernos exigiram uma ênfase maior em sistemas com componentes, onde a implementação é particionada em componentes independentes que se comunicam para realizar uma tarefa desejada. A pesquisa em arquitetura de software investiga métodos para determinar como melhor partitionar um sistema, como os componentes se identificam e se comunicam, como a informação é comunicada, como os elementos de um sistema podem evoluir de forma independente, e como todos os itens acima podem ser descritos usando notações formais e informais.

Uma boa arquitetura não é criada no vácuo. Todas as decisões de projeto no nível arquitetural deve ser feito dentro do contexto funcional, comportamental e requisitos sociais do sistema que está sendo projetado, que é um princípio que se aplica igualmente tanto para a arquitetura de software quanto para o campo tradicional da arquitetura predial. A diretriz de que “a forma segue a função” vem de centenas de anos de experiência com projetos de construção fracassados, mas muitas vezes é ignorado pelos profissionais de software. A parte engraçada dentro do esboço do Monty Python, citado acima, é a noção absurda de que um arquiteto, quando diante do objetivo de projetar um bloco urbano de apartamentos (apartamentos), apresentaria uma projeto de construção com todos os componentes de um matadouro moderno. Pode muito bem ser o melhor projeto de matadouro já concebido, mas isso seria de pouco conforto para os inquilinos em potencial enquanto são arrastados por corredores contendo facas rotativas.

A hipérbole de *The Architects Sketch* pode parecer ridícula, mas considere com que frequência vemos projetos de software começarem com a adoção da última moda em design arquitetônico e só depois descobrirá se os requisitos do sistema exigem tal arquitetura.

Design-by-buzzword é uma ocorrência comum. Pelo menos parte desse comportamento dentro do indústria de software é devido a uma falta de compreensão de *por que* um determinado conjunto de arquitetura restrições é útil. Em outras palavras, o raciocínio por trás de boas arquiteturas de software é não é aparente para os designers quando essas arquiteturas são selecionadas para reutilização.

Esta dissertação explora uma junção nas fronteiras de duas disciplinas de pesquisa em ciência da computação: software e redes. A pesquisa de software há muito se preocupa com a categorização de projetos de software e o desenvolvimento de metodologias de projeto, mas raramente foi capaz de avaliar objetivamente o impacto de várias escolhas de design sobre comportamento do sistema. A pesquisa em rede, em contraste, é focada nos detalhes de comportamento de comunicação entre sistemas e melhorar o desempenho de técnicas de comunicação, muitas vezes ignorando o fato de que mudar o estilo de interação de um aplicação pode ter mais impacto no desempenho do que os protocolos de comunicação usados para essa interação. Meu trabalho é motivado pelo desejo de entender e avaliar a projeto arquitetônico de software de aplicativo baseado em rede por meio do uso de princípios restrições arquitetônicas, obtendo assim as características funcionais, de desempenho e sociais propriedades desejadas de uma arquitetura. Quando recebe um nome, um conjunto coordenado de restrições arquitetônicas torna-se um estilo arquitetônico.

Os três primeiros capítulos desta dissertação definem uma estrutura para entender arquitetura de software por meio de estilos arquitetônicos, revelando como os estilos podem ser usados para guiar o projeto arquitetônico de software aplicativo baseado em rede. Estilos arquitetônicos comuns

são pesquisados e classificados de acordo com as propriedades arquitetônicas que eles induzem quando aplicada a uma arquitetura para hipermídia baseada em rede. Essa classificação é usada para identificar um conjunto de restrições arquitetônicas que poderiam ser usadas para melhorar a arquitetura de o início da World Wide Web.

Arquitetar a Web requer uma compreensão de seus requisitos, pois discuta no Capítulo 4. A Web pretende ser uma hipermídia distribuída *em escala da Internet* sistema, o que significa muito mais do que apenas dispersão geográfica. A Internet é sobre a interconexão de redes de informação através das fronteiras organizacionais. Fornecedores serviços de informação devem ser capazes de lidar com as demandas de escalabilidade anárquica e a implantação independente de componentes de software. A hipermídia distribuída fornece uma meios uniformes de acesso aos serviços através da incorporação de controles de ação dentro do apresentação de informações recuperadas de sites remotos. Uma arquitetura para a Web deve portanto, ser projetado com o contexto de comunicação de objetos de dados de grande granularidade entre redes de alta latência e vários limites de confiança.

O Capítulo 5 apresenta e elabora a Transferência de Estado Representacional (REST) estilo arquitetural para sistemas hipermídia distribuídos. REST fornece um conjunto de restrições arquitetônicas que, quando aplicadas como um todo, enfatizam a escalabilidade de interações de componentes, generalidade de interfaces, implantação independente de componentes, e componentes intermediários para reduzir a latência de interação, reforçar a segurança e encapsular sistemas legados. Eu descrevo os princípios de engenharia de software que orientam o REST e as restrições de interação escolhidas para reter esses princípios, contrastando-os com o restrições de outros estilos arquitetônicos.

Nos últimos seis anos, o estilo arquitetônico REST tem sido usado para guiar o projeto e desenvolvimento da arquitetura para a Web moderna, conforme apresentado no Capítulo 6. trabalho foi feito em conjunto com minha autoria dos padrões da Internet para o Hypertext Transfer Protocol (HTTP) e Uniform Resource Identifiers (URI), os dois especificações que definem a interface genérica usada por todas as interações de componentes no Rede.

Como a maioria dos sistemas do mundo real, nem todos os componentes da arquitetura da Web implantada obedecer a todas as restrições presentes em seu projeto arquitetônico. REST tem sido usado tanto como significa definir melhorias arquiteturais e identificar incompatibilidades arquiteturais.

As incompatibilidades ocorrem quando, por desconhecimento ou descuido, uma implementação de software é implantado que viola as restrições de arquitetura. Enquanto as incompatibilidades não podem ser evitadas em geral, é possível identificá-los antes que se tornem padronizados. Diversos incompatibilidades dentro da arquitetura moderna da Web são resumidas no Capítulo 6, juntamente com análises de por que eles surgiram e como eles se desviam do REST.

Em resumo, esta dissertação faz as seguintes contribuições para a pesquisa de software na área de Ciência da Informação e da Computação:

- uma estrutura para entender a arquitetura de software por meio de estilos de arquitetura, incluindo um conjunto consistente de terminologia para descrever a arquitetura de software;
- uma classificação de estilos de arquitetura para software de aplicativo baseado em rede por as propriedades arquitetônicas que induziriam quando aplicadas à arquitetura de um sistema hiper mídia distribuído;
- REST, um novo estilo de arquitetura para sistemas hiper mídia distribuídos; e,
- aplicação e avaliação do estilo arquitetural REST no projeto e implantação da arquitetura para a World Wide Web moderna.

# CAPÍTULO 1

## Arquitetura de software

Apesar do interesse na arquitetura de software como campo de pesquisa, há pouca acordo entre os pesquisadores sobre o que exatamente deve ser incluído na definição de arquitetura. Em muitos casos, isso levou a que aspectos importantes do projeto arquitetônico fossem ignorado por pesquisas anteriores. Este capítulo define uma terminologia auto-consistente para arquitetura de software baseada em um exame das definições existentes na literatura e minha própria visão em relação às arquiteturas de aplicativos baseados em rede. Cada definição, destacada dentro de uma caixa para facilitar a referência, é seguida por uma discussão sobre como ele é derivado ou se compara a pesquisas relacionadas.

### 1.1 Abstração em tempo de execução

Uma **arquitetura de software** é uma abstração dos elementos de tempo de execução de um sistema de software durante alguma fase de sua operação. Um sistema pode ser composto de muitos níveis de abstração e muitas fases de operação, cada uma com sua própria arquitetura de software.

No coração da arquitetura de software está o princípio da abstração: esconder alguns dos detalhes de um sistema através do encapsulamento para melhor identificar e sustentar suas propriedades [117]. Um sistema complexo conterá muitos níveis de abstração, cada um com seu arquitetura própria. Uma arquitetura representa uma abstração do comportamento do sistema naquele nível, de tal forma que os elementos arquitetônicos são delineados pelas interfaces abstratas que eles fornecer a outros elementos desse nível [9]. Dentro de cada elemento pode ser encontrado outro arquitetura, definindo o sistema de subelementos que implementam o comportamento representado

pela interface abstrata do elemento pai. Esta recursão de arquiteturas continua para baixo aos elementos mais básicos do sistema: aqueles que não podem ser decompostos em elementos.

Além dos níveis de arquitetura, um sistema de software geralmente terá vários fases operacionais, como inicialização, inicialização, processamento normal, reinicialização e desligar. Cada fase operacional tem sua própria arquitetura. Por exemplo, uma configuração arquivo será tratado como um elemento de dados durante a fase de inicialização, mas não será considerado um elemento arquitetônico durante o processamento normal, pois nesse ponto a informação contidos já terão sido distribuídos por todo o sistema. Pode, de fato, ter definido a arquitetura de processamento normal. Uma descrição geral de uma arquitetura de sistema deve ser capaz de descrever não apenas o comportamento operacional do sistema arquitetura durante cada fase, mas também a arquitetura das transições entre as fases.

Perry e Wolf [105] definem elementos de processamento como “transformadores de dados”, enquanto Shaw et al. [118] descrevem componentes como “o locus de computação e estado”. Isto é esclarecido em Shaw e Clements [122]: “Um componente é uma unidade de software que executa alguma função em tempo de execução. Exemplos incluem programas, objetos, processos e filtros.” Isso levanta uma importante distinção entre arquitetura de software e o que é normalmente referido como estrutura de software: o primeiro é uma abstração do tempo de execução comportamento de um sistema de software, enquanto o último é uma propriedade da fonte de software estática código. Embora haja vantagens em ter a estrutura modular do código-fonte corresponder à decomposição do comportamento dentro de um sistema em execução, também há vantagens em ter componentes de software independentes implementados usando partes do mesmo código (por exemplo, bibliotecas compartilhadas). Separamos a visão da arquitetura de software daquela do

código-fonte para focar nas características de tempo de execução do software independente de um implementação de determinado componente. Portanto, o projeto arquitetônico e o código-fonte projeto estrutural, embora intimamente relacionado, são atividades de projeto separadas. Infelizmente, algumas descrições de arquitetura de software não fazem essa distinção (por exemplo, [9]).

## 1.2 Elementos

Uma **arquitetura de software** é definida por uma configuração de elementos arquitetônicos – componentes, conectores e dados – restritos em seus relacionamentos para alcançar um conjunto desejado de propriedades arquiteturais.

Um exame abrangente do escopo e da base intelectual da arquitetura de software pode ser encontrado em Perry e Wolf [105]. Apresentam um modelo que define um software arquitetura como um conjunto de *elementos* arquitetônicos que têm uma *forma particular*, explicada por um conjunto de *razões*. Os elementos arquitetônicos incluem processamento, dados e elementos de conexão. A forma é definida pelas propriedades dos elementos e as relações entre os elementos — isto é, as restrições sobre os elementos. A justificativa fornece a base para a arquitetura, capturando a motivação para a escolha do estilo arquitetônico, a escolha dos elementos e a forma.

Minhas definições para arquitetura de software são uma versão elaborada daquelas dentro do Perry e Wolf [105], exceto que eu excluo a lógica. Embora a razão seja aspecto importante da pesquisa de arquitetura de software e da descrição da arquitetura em particular, incluí-lo na definição de arquitetura de software implicaria que a documentação do projeto faz parte do sistema de tempo de execução. A presença ou ausência de razão pode influenciar a evolução de uma arquitetura, mas, uma vez constituída, a arquitetura é independente de suas razões de ser. Sistemas refletivos [80] podem usar as características de

desempenho passado para mudar o comportamento futuro, mas ao fazê-lo eles estão substituindo um arquitetura de nível com outra arquitetura de nível inferior, em vez de abranger racional dentro dessas arquiteturas.

Como ilustração, considere o que acontece com um edifício se suas plantas e design planos são queimados. O edifício desmorona imediatamente? Não, uma vez que as propriedades por que as paredes sustentam o peso do telhado permanecem intactas. Uma arquitetura tem, por design, um conjunto de propriedades que permitem atender ou exceder os requisitos do sistema. O desconhecimento dessas propriedades pode levar a alterações posteriores que violam a arquitetura, apenas uma vez que a substituição de uma parede de suporte por um grande caixilho de janela pode violar o estabilidade estrutural de um edifício. Assim, em vez de lógica, nossa definição de software arquitetura inclui propriedades arquitetônicas. A razão explica essas propriedades, e a falta de lógica pode resultar em decadência gradual ou degradação da arquitetura ao longo do tempo, mas a lógica em si não faz parte da arquitetura.

Uma característica chave do modelo em Perry e Wolf [105] é a distinção dos vários tipos de elementos. *Elementos de processamento* são aqueles que realizam transformações em dados, *dados elementos* são aqueles que contêm a informação que é usada e transformada, e *elementos de conexão* são a cola que mantém as diferentes peças da arquitetura juntos. Eu uso os termos mais comuns de *componentes* e *conectores* para me referir a elementos de processamento e de conexão, respectivamente.

Garlan e Shaw [53] descrevem uma arquitetura de um sistema como uma coleção de componentes computacionais juntamente com uma descrição das interações entre esses componentes - os conectores. Este modelo é expandido em Shaw et al. [118]: O arquitetura de um sistema de software define esse sistema em termos de componentes e de

interações entre esses componentes. Além de especificar a estrutura e a topologia do sistema, a arquitetura mostra a correspondência pretendida entre o sistema requisitos e elementos do sistema construído. Elaboração adicional desta definição pode ser encontrado em Shaw e Garlan [121].

O que é surpreendente sobre o Shaw et al. [118] é que, em vez de definir o arquitetura do software como existente dentro do software, está definindo uma descrição do arquitetura do software como se essa fosse a arquitetura. No processo, softwares a arquitetura como um todo é reduzida ao que é comumente encontrado em ambientes mais informais. diagramas de arquitetura: caixas (componentes) e linhas (conectores). Elementos de dados, juntamente com muitos dos aspectos dinâmicos das arquiteturas de software reais, são ignorados. Tal modelo é incapaz de descrever adequadamente arquiteturas de software baseadas em rede, uma vez que a natureza, localização e movimento dos elementos de dados dentro do sistema é muitas vezes o único determinante mais significativo do comportamento do sistema.

### 1.2.1 Componentes

Um **componente** é uma unidade abstrata de instruções de software e estado interno que fornece uma transformação de dados por meio de sua interface.

Os componentes são o aspecto mais facilmente reconhecido da arquitetura de software. Perry e Os elementos de processamento de Wolf [105] são definidos como aqueles componentes que transformação nos elementos de dados. Garlan e Shaw [53] descrevem componentes simplesmente como os elementos que executam a computação. Nossa definição tenta ser mais precisa em fazendo a distinção entre componentes e o software dentro dos conectores.

Um componente é uma unidade abstrata de instruções de software e estado interno que fornece uma transformação de dados através de sua interface. Exemplos de transformações incluem

carregando na memória a partir do armazenamento secundário, realizando alguns cálculos, traduzindo para um formato diferente, encapsulamento com outros dados, etc. O comportamento de cada componente é parte da arquitetura na medida em que esse comportamento pode ser observado ou discernido a partir do ponto de vista de outro componente [9]. Em outras palavras, um componente é definido por sua interface e os serviços que ela fornece a outros componentes, e não por sua implementação por trás da interface. Parnas [101] definiria isso como o conjunto de suposições que outros elementos arquitetônicos podem fazer sobre o componente.

### 1.2.2 Conectores

Um **conector** é um mecanismo abstrato que medeia a comunicação, coordenação ou cooperação entre os componentes.

Perry e Wolf [105] descrevem os elementos de conexão vagamente como a cola que mantém o várias peças da arquitetura juntas. Uma definição mais precisa é fornecida por Shaw e Clements [122]: Um conector é um mecanismo abstrato que medeia a comunicação, coordenação ou cooperação entre os componentes. Os exemplos incluem compartilhado representações, chamadas de procedimento remoto, protocolos de passagem de mensagens e fluxos de dados. Talvez a melhor maneira de pensar sobre conectores seja contrastá-los com componentes.

Os conectores permitem a comunicação entre os componentes transferindo elementos de dados de uma interface para outra sem alterar os dados. Internamente, um conector pode consistem em um subsistema de componentes que transformam os dados para transferência, realizam a transferir e, em seguida, reverter a transformação para entrega. No entanto, o exterior a abstração comportamental capturada pela arquitetura ignora esses detalhes. Em contrapartida, um componente pode, mas nem sempre irá, transformar dados da perspectiva externa.

### 1.2.3 Dados

Um **datum** é um elemento de informação que é transferido de um componente, ou recebido por um componente, por meio de um conector.

Como observado acima, a presença de elementos de dados é a distinção mais significativa entre o modelo de arquitetura de software definido por Perry e Wolf [105] e o modelo usado por grande parte da pesquisa rotulou arquitetura de software [1, 5, 9, 53, 56, 117-122, 128]. Boasson [24] critica a pesquisa atual de arquitetura de software por sua ênfase em estruturas de componentes e ferramentas de desenvolvimento de arquitetura, sugerindo que mais foco deve ser colocado na modelagem arquitetural centrada em dados. Comentários semelhantes são feitos por Jackson [67].

Um dado é um elemento de informação que é transferido de um componente, ou recebido por um componente, através de um conector. Exemplos incluem sequências de bytes, mensagens, parâmetros e objetos serializados, mas não incluem informações que são permanentemente residente ou oculto dentro de um componente. Do ponto de vista arquitetônico, um “arquivo” é uma transformação que um componente do sistema de arquivos pode fazer a partir de um dado “nome do arquivo” recebido em sua interface para uma sequência de bytes gravada em um sistema de armazenamento. Componentes também podem gerar dados, como no caso de um software encapsulamento de um relógio ou sensor.

A natureza dos elementos de dados dentro de uma arquitetura de aplicação baseada em rede irá muitas vezes determinar se um determinado estilo arquitetônico é apropriado ou não. Isto é particularmente evidente na comparação de paradigmas de design de código móvel [50], onde a escolha deve ser feita entre interagir diretamente com um componente ou transformar o componente em um elemento de dados, transferindo-o através de uma rede e, em seguida, transformando-o

de volta para um componente que pode interagir localmente. É impossível avaliar tal uma arquitetura sem considerar elementos de dados no nível arquitetural.

### 1.3 Configurações

Uma **configuração** é a estrutura de relacionamentos de arquitetura entre componentes, conectores e dados durante um período de tempo de execução do sistema.

Abowd et al. [1] definem a descrição arquitetônica como suporte à descrição de sistemas em termos de três classes sintáticas básicas: componentes, que são o locus da computação; conectores, que definem as interações entre os componentes; e configurações, que são coleções de componentes e conectores interativos. Vários concretos específicos de estilo notações podem ser usadas para representá-los visualmente, facilitar a descrição de computações e interações, e restringir o conjunto de sistemas desejáveis.

Estritamente falando, pode-se pensar em uma configuração como sendo equivalente a um conjunto de restrições específicas na interação do componente. Por exemplo, Perry e Wolf [105] incluem topologia em sua definição de relacionamentos de forma arquitetônica. No entanto, separando o topologia ativa de restrições mais gerais permite que um arquiteto distinguir a configuração ativa do domínio potencial de todos os legítimos configurações. Racionalidade adicional para distinguir configurações dentro de arquitetura linguagens de descrição é apresentada em Medvidovic e Taylor [86].

### 1.4 Propriedades

O conjunto de propriedades arquitetônicas de uma arquitetura de software inclui todas as propriedades que derivam da seleção e arranjo de componentes, conectores e dados dentro do sistema. Exemplos incluem as propriedades funcionais alcançadas pelo sistema e não

propriedades funcionais, como relativa facilidade de evolução, reutilização de componentes,

eficiência e extensibilidade dinâmica, muitas vezes referidos como atributos de qualidade [9].

As propriedades são induzidas pelo conjunto de restrições dentro de uma arquitetura. As restrições são muitas vezes motivadas pela aplicação de um princípio de engenharia de software [58] a um aspecto de os elementos arquitetônicos. Por exemplo, o estilo *de tubo e filtro uniforme* obtém a qualidades de reutilização de componentes e configurabilidade da aplicação aplicando generalidade para suas interfaces de componentes — restringindo os componentes a uma única interface modelo. Portanto, a restrição arquitetônica é “interface de componente uniforme”, motivada por o princípio da generalidade, a fim de obter duas qualidades desejáveis que se tornarão as propriedades arquitetônicas de componentes reutilizáveis e configuráveis quando esse estilo é instanciado dentro de uma arquitetura.

O objetivo do projeto arquitetônico é criar uma arquitetura com um conjunto de propriedades que formam um superconjunto dos requisitos do sistema. A importância relativa do várias propriedades arquitetônicas dependem da natureza do sistema pretendido. Seção 2.3 examina as propriedades que são de interesse particular para aplicativos baseados em rede arquiteturas.

## 1.5 Estilos

Um **estilo de arquitetura** é um conjunto coordenado de restrições de arquitetura que restringem as funções/recursos de elementos de arquitetura e os relacionamentos permitidos entre esses elementos em qualquer arquitetura que esteja em conformidade com esse estilo.

Uma vez que uma arquitetura incorpora propriedades funcionais e não funcionais, ela pode ser

difícil comparar diretamente arquiteturas para diferentes tipos de sistemas, ou mesmo para

mesmo tipo de sistema definido em ambientes diferentes. Os estilos são um mecanismo para categorizar arquiteturas e definir suas características comuns [38]. Cada estilo fornece uma abstração para as interações dos componentes, capturando a essência de um padrão de interação ignorando os detalhes incidentais do resto da arquitetura [117].

Perry e Wolf [105] definem o estilo arquitetônico como uma abstração de tipos de elementos e aspectos formais de várias arquiteturas específicas, talvez concentrando-se apenas em certas aspectos de uma arquitetura. Um estilo de arquitetura encapsula decisões importantes sobre os elementos arquitetônicos e enfatiza importantes restrições sobre os elementos e suas relacionamentos. Esta definição permite estilos que se concentram apenas nos conectores de um arquitetura, ou em aspectos específicos das interfaces dos componentes.

Em contraste, Garlan e Shaw [53], Garlan et al. [56], e Shaw e Clements [122] todos definir estilo em termos de um padrão de interações entre componentes tipados. Especificamente, um estilo arquitetônico determina o vocabulário de componentes e conectores que podem ser usados em instâncias desse estilo, juntamente com um conjunto de restrições sobre como eles podem ser combinado [53]. Essa visão restrita dos estilos arquitetônicos é resultado direto de sua definição de arquitetura de software – pensando na arquitetura como uma descrição formal, ao invés de um sistema em execução, leva a abstrações baseadas apenas nos padrões compartilhados de diagramas de caixas e linhas. Abowd et ai. [1] vá mais longe e defina isso explicitamente como visualizar o coleção de convenções que são usadas para interpretar uma classe de descrições arquitetônicas como definindo um estilo arquitetônico.

Novas arquiteturas podem ser definidas como instâncias de estilos específicos [38]. Desde estilos arquitetônicos podem abordar diferentes aspectos da arquitetura de software,

arquitetura pode ser composta de vários estilos. Da mesma forma, um estilo híbrido pode ser formado combinando vários estilos básicos em um único estilo coordenado.

Alguns estilos arquitetônicos são frequentemente retratados como soluções “bala de prata” para todas as formas de software. No entanto, um bom designer deve selecionar um estilo que corresponda às necessidades do problema particular que está sendo resolvido [119]. Escolhendo o estilo arquitetônico certo para um aplicação baseada em rede requer uma compreensão do domínio do problema [67] e assim as necessidades de comunicação da aplicação, uma consciência da variedade de estilos arquitetônicos e as preocupações particulares que eles abordam, e a capacidade de antecipar a sensibilidade de cada estilo de interação para as características da rede comunicação [133].

Infelizmente, usar o termo estilo para se referir a um conjunto coordenado de restrições geralmente leva à confusão. Este uso difere substancialmente da etimologia do *estilo*, que enfatizaria a personalização do processo de design. Loerke [76] dedica um capítulo à denegrindo a noção de que as preocupações estilísticas pessoais têm algum lugar no trabalho de um arquiteto profissional. Em vez disso, ele descreve os estilos como a visão dos críticos da arquitetura passada, onde a escolha disponível de materiais, a cultura da comunidade ou o ego do local governante foram responsáveis pelo estilo arquitetônico, não o designer. Em outras palavras, Loerke vê a verdadeira fonte de estilo na arquitetura de construção tradicional como o conjunto de restrições aplicado ao design, e obter ou copiar um estilo específico deve ser o menor dos objetivos do projetista. Uma vez que referir-se a um conjunto nomeado de restrições como um estilo torna mais fácil comunicar as características das restrições comuns, usamos estilos arquitetônicos como método de abstração, em vez de um indicador de design personalizado.

## 1.6 Padrões e Linguagens de Padrões

Em paralelo com a pesquisa de engenharia de software em estilos arquitetônicos, o objeto comunidade de programação orientada vem explorando o uso de padrões de projeto e linguagens de padrões para descrever abstrações recorrentes em software baseado em objetos desenvolvimento. Um padrão de projeto é definido como uma construção de sistema importante e recorrente. Uma linguagem de padrões é um sistema de padrões organizado em uma estrutura que orienta os padrões aplicação [70]. Ambos os conceitos são baseados nos escritos de Alexander et al. [3, 4] com no que diz respeito à arquitetura de edifícios.

O espaço de design de padrões inclui preocupações de implementação específicas para o técnicas de programação orientada a objetos, como herança de classe e interface composição, bem como as questões de design de nível superior abordadas pelos estilos arquitetônicos [51]. Em alguns casos, as descrições de estilo arquitetônico foram reformuladas como padrões [120]. No entanto, um benefício primário dos padrões é que eles podem descrever protocolos complexos de interações entre objetos como uma única abstração [91], assim incluindo tanto as restrições de comportamento quanto as especificidades da implementação. Em geral, um padrão, ou linguagem de padrões no caso de múltiplos padrões integrados, pode ser pensado como uma receita para implementar um conjunto desejado de interações entre objetos. Em outras palavras, um padrão define um processo para resolver um problema seguindo um caminho de design e escolhas de implementação [34].

Assim como os estilos de arquitetura de software, a pesquisa de padrões de software se desviou um pouco de sua origem na arquitetura de edifícios. De fato, a noção de padrões de Alexander centra-se não em arranjos recorrentes de elementos arquitetônicos, mas sim na padrão de eventos – atividade humana e emoção – que ocorrem dentro de um espaço, com o

compreensão de que um padrão de eventos não pode ser separado do espaço onde ocorre [3]. A filosofia de design de Alexander é identificar padrões de vida que são comuns ao cultura-alvo e determinar quais restrições arquitetônicas são necessárias para diferenciar um dado espaço de modo que permita que os padrões desejados ocorram naturalmente. Esses padrões existem em vários níveis de abstração e em todas as escalas.

*Como um elemento no mundo, cada padrão é uma relação entre um determinado contexto, um certo sistema de forças que ocorre repetidamente nesse contexto e uma certa configuração espacial que permite que essas forças se resolvam.*

*Como um elemento da linguagem, um padrão é uma instrução, que mostra como essa configuração espacial pode ser usada, repetidamente, para resolver um determinado sistema de forças, onde quer que o contexto o torne relevante.*

*O padrão é, em suma, ao mesmo tempo uma coisa que acontece no mundo e a regra que nos diz como criar essa coisa e quando devemos criá-la. É tanto um processo quanto uma coisa; tanto uma descrição de uma coisa que está viva quanto uma descrição do processo que irá gerar essa coisa. [3]*

De muitas maneiras, os padrões de Alexander têm mais em comum com arquitetura de software estilos do que os padrões de projeto da pesquisa OOPL. Um estilo arquitetônico, como uma conjunto de restrições, é aplicado a um espaço de projeto para induzir a propriedades desejadas do sistema. Ao aplicar um estilo, um arquiteto está se diferenciando o espaço de design de software na esperança de que o resultado corresponda melhor às forças inerentes à aplicação, levando assim a um comportamento do sistema que aprimora o padrão natural em vez de do que entrar em conflito com ela.

### 1.7 Visualizações

*Um ponto de vista de arquitetura geralmente é específico do aplicativo e varia amplamente com base no domínio do aplicativo. ... vimos pontos de vista de arquitetura que abordam uma variedade de questões, incluindo: questões temporais, abordagens de estado e controle, representação de dados, ciclo de vida da transação, salvaguardas de segurança e demanda de pico e degradação graciosa. Sem dúvida, há muitos outros pontos de vista possíveis. [70]*

Além das muitas arquiteturas dentro de um sistema e dos muitos estilos arquitetônicos a partir da qual as arquiteturas são compostas, também é possível visualizar uma arquitetura de muitas perspectivas diferentes. Perry e Wolf [105] descrevem três visões importantes em arquitetura de software: processamento, dados e visualizações de conexão. Uma visão de processo enfatiza o fluxo de dados através dos componentes e alguns aspectos das conexões entre os componentes em relação aos dados. Uma visualização de dados enfatiza o fluxo de processamento, com menos ênfase nos conectores. Uma visão de conexão enfatiza a relação entre componentes e o estado da comunicação.

Múltiplas visões arquitetônicas são comuns em estudos de caso de arquiteturas específicas [9]. Uma metodologia de projeto arquitetônico, o 4+1 View Model [74], organiza a descrição de uma arquitetura de software usando cinco visualizações simultâneas, cada uma das quais aborda um conjunto específico de preocupações.

## **1.8 Trabalho Relacionado**

Incluo aqui apenas as áreas de pesquisa que definem a arquitetura de software ou descrevem estilos de arquitetura de software. Outras áreas para pesquisa de arquitetura de software incluem técnicas de análise arquitetônica, recuperação e reengenharia de arquitetura, ferramentas e ambientes para projeto arquitetônico, refinamento da arquitetura desde a especificação até a implementação e estudos de caso de arquiteturas de software implantadas [55]. Trabalho relacionado em as áreas de classificação de estilo, paradigmas de processos distribuídos e middleware são discutido no Capítulo 3.

### **1.8.1 Metodologias de Projeto**

A maioria das primeiras pesquisas sobre arquitetura de software concentrava-se em metodologias de projeto.

Por exemplo, o design orientado a objetos [25] defende uma maneira de estruturar problemas que naturalmente a uma arquitetura baseada em objetos (ou, mais precisamente, não leva naturalmente a qualquer outra forma de arquitetura). Uma das primeiras metodologias de design a enfatizar o design no nível arquitetural é Jackson System Development [30]. JSD intencionalmente estrutura a análise de um problema de modo que conduza a um estilo de arquitetura que combina pipe-and-filter (fluxo de dados) e restrições de controle de processo. Essas metodologias de projeto tendem a produzir apenas um estilo de arquitetura.

Houve alguns trabalhos iniciais investigando metodologias para a análise e desenvolvimento de arquiteturas. Kazman et al. descreveram métodos de projeto para obter os aspectos arquitetônicos de um projeto por meio de análise baseada em cenário com SAAM [68] e análise de trade-off arquitetural via ATAM [69]. Shaw [119] compara uma variedade de caixas projetos de seta e seta para um sistema de controle de cruzeiro automotivo, cada um feito usando um metodologia de projeto e abrangendo diversos estilos arquitetônicos.

### **1.8.2 Manuais de Design, Padrões de Design e Linguagens de Padrões**

Shaw [117] defende o desenvolvimento de manuais de arquitetura nos mesmos moldes disciplinas tradicionais de engenharia. A comunidade de programação orientada a objetos assumiu a liderança na produção de catálogos de padrões de design, como exemplificado pela "Gang of Four" [51] e os ensaios editados por Coplien e Schmidt [33].

Padrões de projeto de software tendem a ser mais orientados a problemas do que estilos de arquitetura. Shaw [120] apresenta oito exemplos de padrões arquitetônicos baseados nos estilos arquitetônicos

descrito em [53], incluindo informações sobre os tipos de problemas mais adequados para cada arquitetura. Buschmann et al. [28] fornecem um exame abrangente do padrões arquiteturais comuns ao desenvolvimento baseado em objetos. Ambas as referências são puramente descritivo e não faça nenhuma tentativa de comparar ou ilustrar as diferenças entre padrões arquitetônicos.

Tepfenhart e Cusick [129] usam um mapa bidimensional para diferenciar entre taxonomias de domínio, modelos de domínio, estilos arquitetônicos, frameworks, kits, design padrões e aplicações. Na topologia, os padrões de projeto são projetos predefinidos estruturas usadas como blocos de construção para uma arquitetura de software, enquanto estilos arquitetônicos são conjuntos de características operacionais que identificam uma família arquitetônica independente de domínio do aplicativo. No entanto, eles não definem a arquitetura em si.

### **1.8.3 Modelos de referência e arquiteturas de software específicas de domínio (DSSA)**

Os modelos de referência são desenvolvidos para fornecer estruturas conceituais para descrever arquiteturas e mostrando como os componentes estão relacionados entre si [117]. O objeto Management Architecture (OMA), desenvolvida pela OMG [96] como modelo de referência para arquiteturas de objetos distribuídos intermediados, especifica como os objetos são definidos e criados, como os aplicativos cliente invocam objetos e como os objetos podem ser compartilhados e reutilizados. o ênfase está no gerenciamento de objetos distribuídos, em vez de aplicação eficiente interação.

Hayes-Roth et al. [62] definem a arquitetura de software específica de domínio (DSSA) como compreendendo: a) uma arquitetura de referência, que descreve uma framework para um domínio significativo de aplicações, b) uma biblioteca de componentes, que

contém pedaços reutilizáveis de conhecimento de domínio e c) um método de configuração de aplicativo para selecionar e configurar componentes dentro da arquitetura para atender requisitos de aplicação. Tracz [130] fornece uma visão geral do DSSA.

Os projetos DSSA foram bem-sucedidos na transferência de decisões de arquitetura para a execução sistemas, restringindo o espaço de desenvolvimento de software a um estilo de arquitetura específico que corresponde aos requisitos de domínio [88]. Exemplos incluem ADAGE [10] para aviônicos, AIS [62] para sistemas inteligentes adaptativos, e MetaH [132] para orientação de mísseis, navegação, e sistemas de controle. DSSA enfatiza a reutilização de componentes dentro de um domínio arquitetônico, em vez de selecionar um estilo arquitetônico específico para cada sistema.

#### **1.8.4 Linguagens de Descrição de Arquitetura (ADL)**

A maioria dos trabalhos publicados recentemente sobre arquiteturas de software está na área de linguagens de descrição de arquitetura (ADL). Uma AVD é, de acordo com Medvidovic e Taylor [86], uma linguagem que fornece recursos para a especificação e modelagem explícita da arquitetura conceitual de um sistema de software, incluindo no mínimo: componentes, interfaces de componentes, conectores e configurações de arquitetura.

Darwin é uma linguagem declarativa que se destina a ser uma notação de propósito geral para especificar a estrutura de sistemas compostos por diversos componentes usando diversos mecanismos de interação [81]. As qualidades interessantes de Darwin são que ele permite que o especificação de arquiteturas distribuídas e arquiteturas compostas dinamicamente [82].

UniCon [118] é uma linguagem e conjunto de ferramentas associado para compor uma arquitetura de um conjunto restrito de exemplos de componentes e conectores. Wright [5] fornece uma base formal

para especificar as interações entre componentes arquiteturais especificando o conector tipos por seus protocolos de interação.

Assim como as metodologias de design, as ADLs geralmente introduzem suposições arquiteturais específicas que podem afetar sua capacidade de descrever alguns estilos arquitetônicos e podem entrar em conflito com as suposições no middleware existente [38]. Em alguns casos, uma ADL é projetada especificamente para um único estilo arquitetônico, melhorando assim sua capacidade de descrição e análise ao custo da generalidade. Por exemplo, C2SADEL [88] é um ADL projetado especificamente para descrever arquiteturas desenvolvidas no estilo C2 [128]. Em contraste, ACME [57] é uma ADL que tenta ser o mais genérica possível, mas com o trade-off sendo que não suporta análise específica de estilo e a construção de aplicações reais; em vez disso, seu foco está no intercâmbio entre as ferramentas de análise.

### **1.8.5 Modelos Arquitetônicos Formais**

Abowd et al. [1] afirmam que os estilos arquitetônicos podem ser descritos formalmente em termos de um pequeno conjunto de mapeamentos do domínio sintático de descrições arquitetônicas (caixa e diagramas de linha) para o domínio semântico do significado arquitetônico. No entanto, isso pressupõe que a arquitetura é a descrição, em vez de uma abstração de um sistema em execução.

Inverardi e Wolf [65] usam o formalismo Chemical Abstract Machine (CHAM) para modelar elementos de arquitetura de software como produtos químicos cujas reações são controladas por regras explicitamente declaradas. Especifica o comportamento dos componentes de acordo com como eles transforma os elementos de dados disponíveis e usa regras de composição para propagar o transformações em um resultado geral do sistema. Embora este seja um modelo interessante, é

claro como o CHAM poderia ser usado para descrever qualquer forma de arquitetura cuja propósito vai além de transformar um fluxo de dados.

Rapide [78] é uma linguagem de simulação concorrente, baseada em eventos, projetada especificamente para definir e simular arquiteturas de sistemas. O simulador produz uma ordem parcialmente conjunto de eventos que podem ser analisados quanto à conformidade com as restrições arquitetônicas interconexão. Le Métayer [75] apresenta um formalismo para a definição de arquiteturas em termos de grafos e gramáticas de grafos.

## 1.9 Resumo

Este capítulo examinou o pano de fundo para esta dissertação. Apresentando e formalizando um conjunto consistente de terminologia para conceitos de arquitetura de software é necessário para evitar a confusão entre arquitetura e descrição de arquitetura que é comum na literatura, particularmente porque grande parte da pesquisa anterior sobre arquitetura exclui os dados como importante elemento arquitetônico. Conclui com um levantamento de outras pesquisas relacionadas arquitetura de software e estilos de arquitetura.

Os próximos dois capítulos continuam nossa discussão do material de base, concentrando-se em arquiteturas de aplicativos baseados em rede e descrevendo como os estilos podem ser usados para orientar seu projeto arquitetônico, seguido por um levantamento de estilos arquitetônicos comuns usando uma metodologia de classificação que destaca as propriedades arquitetônicas induzidas quando os estilos são aplicados a uma arquitetura para hipermídia baseada em rede.

## CAPÍTULO 2

### Arquiteturas de aplicativos baseados em rede

Este capítulo continua nossa discussão sobre o material de referência, concentrando-se em redes arquiteturas de aplicativos baseados e descrevendo como os estilos podem ser usados para guiar suas projeto arquitetônico.

#### 2.1 Escopo

A arquitetura é encontrada em vários níveis dentro dos sistemas de software. Esta dissertação examina o mais alto nível de abstração na arquitetura de software, onde as interações entre os componentes são capazes de serem realizados na comunicação em rede. Limitamos nossa discussão para estilos de arquiteturas de aplicativos baseados em rede, a fim de reduzir o dimensões de variância entre os estilos estudados.

##### 2.1.1 Baseado em rede versus distribuído

A principal distinção entre arquiteturas baseadas em rede e arquiteturas de software em geral é que a comunicação entre os componentes é restrita à passagem de mensagens [6], ou o equivalente de passagem de mensagens se um mecanismo mais eficiente pode ser selecionado em execução tempo com base na localização dos componentes [128].

Tanenbaum e van Renesse [127] fazem uma distinção entre sistemas distribuídos e sistemas baseados em rede: um sistema distribuído é aquele que parece para seus usuários um sistema centralizado comum, mas roda em várias CPUs independentes. Em contraste, sistemas baseados em rede são aqueles capazes de operar em uma rede, mas não

necessariamente de uma forma que seja transparente para o usuário. Em alguns casos é desejável que o usuário esteja ciente da diferença entre uma ação que requer uma solicitação de rede e um que seja satisfatório em seu sistema local, particularmente quando o uso da rede implica em um custo extra de transação [133]. Esta dissertação abrange sistemas baseados em rede por não limitando os estilos candidatos àqueles que preservam a transparência para o usuário.

### **2.1.2 Software de aplicativo versus software de rede**

Outra restrição ao escopo desta dissertação é que limitamos nossa discussão a arquiteturas de aplicativos, excluindo o sistema operacional, software de rede e alguns estilos de arquitetura que usariam apenas uma rede para suporte ao sistema (por exemplo, controle de processo estilos [53]). As aplicações representam a funcionalidade “consciente do negócio” de um sistema [131].

A arquitetura de software aplicativo é um nível de abstração de um sistema geral, no qual os objetivos de uma ação do usuário são representáveis como propriedades arquitetônicas funcionais. Por exemplo, uma aplicação hipermídia deve se preocupar com a localização da informação páginas, realizando solicitações e renderizando fluxos de dados. Isso contrasta com uma rede abstração, onde o objetivo é mover bits de um local para outro sem levar em consideração por que esses bits estão sendo movidos. É apenas ao nível da aplicação que podemos avaliar design trade-offs com base no número de interações por ação do usuário, a localização de estado do aplicativo, a taxa de transferência efetiva de todos os fluxos de dados (em oposição ao potencial taxa de transferência de um único fluxo de dados), a extensão da comunicação sendo realizada por usuário ação, etc

## 2.2 Avaliando o Design de Arquiteturas de Aplicativos

Um dos objetivos desta dissertação é fornecer orientação de design para a tarefa de selecionar ou criando a arquitetura mais adequada para um determinado domínio de aplicação, mantendo em mente que uma arquitetura é a realização de um projeto arquitetônico e não o projeto em si. Uma arquitetura pode ser avaliada por suas características de tempo de execução, mas obviamente preferem um mecanismo de avaliação que possa ser aplicado ao candidato projetos arquitetônicos antes de ter que implementar todos eles. Infelizmente, a arquitetura os projetos são notoriamente difíceis de avaliar e comparar de maneira objetiva. Como a maioria artefatos de design criativo, as arquiteturas são normalmente apresentadas como um trabalho concluído, como se o projeto simplesmente surgiu totalmente formado da mente do arquiteto. Para avaliar uma projeto arquitetônico, precisamos examinar a lógica do projeto por trás das restrições que ele coloca em um sistema e compara as propriedades derivadas dessas restrições com o destino objetivos do aplicativo.

O primeiro nível de avaliação é definido pelos requisitos funcionais do aplicativo. Por exemplo, não faz sentido avaliar o projeto de uma arquitetura de controle de processo contra os requisitos de um sistema de hipermídia distribuído, uma vez que a comparação é discutível se o arquitetura não funcionaria. Embora isso elimine alguns candidatos, na maioria casos, restarão muitos outros projetos arquitetônicos capazes de atender às necessidades funcionais do aplicativo. O restante difere por sua ênfase relativa no não requisitos funcionais – o grau em que cada arquitetura suportaria os vários propriedades arquitetônicas não funcionais que foram identificadas como necessárias para o sistema. Como as propriedades são criadas pela aplicação de restrições arquitetônicas, é possível avaliar e comparar diferentes projetos arquitetônicos, identificando os

restrições dentro de cada arquitetura, avaliando o conjunto de propriedades induzidas por cada restrição e comparando as propriedades cumulativas do projeto com essas propriedades exigido do aplicativo.

Conforme descrito no capítulo anterior, um estilo arquitetônico é um conjunto coordenado de restrições arquitetônicas que receberam um nome para facilitar a referência. Cada decisão de projeto arquitetônico pode ser vista como uma aplicação de um estilo. Desde a adição de uma restrição pode derivar um novo estilo, podemos pensar no espaço de todas as possibilidades arquitetônicas estilos como uma árvore de derivação, com sua raiz sendo o estilo nulo (conjunto vazio de restrições).

Quando suas restrições não entram em conflito, os estilos podem ser combinados para formar estilos híbridos, eventualmente culminando em um estilo híbrido que representa uma abstração completa do projeto arquitetônico. Um projeto arquitetônico pode, portanto, ser analisado por seu conjunto de restrições em uma árvore de derivação e avaliar o efeito cumulativo das restrições representadas por essa árvore. Se entendermos as propriedades induzidas por cada estilo, então percorrer a árvore de derivação nos dá uma compreensão do design geral das propriedades arquitetônicas. As necessidades específicas de uma aplicação podem então ser comparadas com as propriedades do desenho. A comparação torna-se uma questão relativamente simples de identificar qual projeto arquitetônico satisfaz as propriedades mais desejadas para aquela aplicação.

Deve-se tomar cuidado para reconhecer quando os efeitos de uma restrição podem neutralizar a benefícios de alguma outra restrição. No entanto, é possível que um software experiente arquiteto para construir tal árvore de derivação de restrições arquitetônicas para um determinado aplicativo domínio e, em seguida, use a árvore para avaliar muitos projetos arquitetônicos diferentes para aplicativos dentro desse domínio. Assim, construir uma árvore de derivação fornece um mecanismo para orientação de projeto arquitetônico.

A avaliação das propriedades arquitetônicas dentro de uma árvore de estilos é específica para as necessidades de um domínio de aplicação particular porque o impacto de uma determinada restrição é muitas vezes depende das características da aplicação. Por exemplo, o estilo pipe-and-filter permite várias propriedades arquitetônicas positivas quando usadas em um sistema que requer dados transformações entre componentes, enquanto isso não acrescentaria nada além de sobrecarga a um sistema que consiste apenas em mensagens de controle. Uma vez que raramente é útil comparar projetos arquitetônicos em diferentes domínios de aplicativos, o meio mais simples de garantir consistência é tornar a árvore específica do domínio.

A avaliação do projeto é frequentemente uma questão de escolha entre trade-offs. Perry e Wolf [105] descrevem um método de reconhecimento de trade-offs explicitamente colocando um valor numérico peso contra cada propriedade para indicar sua importância relativa para a arquitetura, fornecer uma métrica normalizada para comparar projetos candidatos. No entanto, para ser um métrica significativa, cada peso teria que ser cuidadosamente escolhido usando um objetivo escala consistente em todas as propriedades. Na prática, essa escala não existe. Ao invés de fazendo o arquiteto mexer nos valores de peso até que o resultado corresponda à sua intuição, eu preferem apresentar todas as informações ao arquiteto em um formato facilmente visível, e a intuição do arquiteto seja guiada pelo padrão visual. Isso será demonstrado no Próximo Capítulo.

## 2.3 Propriedades arquitetônicas de interesse chave

Esta seção descreve as propriedades arquiteturais usadas para diferenciar e classificar estilos arquitetônicos nesta dissertação. Não pretende ser uma lista abrangente. Eu tenho incluiu apenas as propriedades que são claramente influenciadas pelo conjunto restrito de estilos

pesquisado. Propriedades adicionais, às vezes chamadas de qualidades de software, são cobertas pela maioria dos livros de engenharia de software (por exemplo, [58]). Bass et al. [9] examinou qualidades em no que diz respeito à arquitetura de software.

### **2.3.1 Desempenho**

Uma das principais razões para focar em estilos para aplicativos baseados em rede é porque interações de componentes podem ser o fator dominante na determinação da percepção do usuário desempenho e eficiência da rede. Uma vez que o estilo arquitetônico influencia a natureza das dessas interações, a seleção de um estilo arquitetônico apropriado pode fazer a diferença entre o sucesso e o fracasso na implantação de um aplicativo baseado em rede.

O desempenho de um aplicativo baseado em rede é limitado primeiramente pelo aplicativo requisitos, depois pelo estilo de interação escolhido, seguido pela arquitetura realizada, e finalmente pela implementação de cada componente. Em outras palavras, o software não pode evitar o custo básico de atender às necessidades de aplicação; por exemplo, se o aplicativo exigir que dados estejam localizados no sistema A e processados no sistema B, então o software não pode evitar movendo esses dados de A para B. Da mesma forma, uma arquitetura não pode ser mais eficiente do que seu estilo de interação permite; por exemplo, o custo de múltiplas interações para mover os dados de A para B não pode ser menor do que a de uma única interação de A para B. Finalmente, independentemente da qualidade de uma arquitetura, nenhuma interação pode ocorrer mais rápido do que um componente implementação pode produzir dados e seu destinatário pode consumir dados.

#### **2.3.1.1 Desempenho da Rede**

As medidas de desempenho da rede são usadas para descrever alguns atributos de comunicação. A taxa de *transferência* é a taxa na qual as informações, incluindo dados de aplicativos e

sobrecarga de comunicação, é transferido entre os componentes. A sobrecarga pode ser separada em overhead de configuração inicial e overhead por interação, uma distinção que é útil para identificando conectores que podem compartilhar a sobrecarga de configuração em várias interações (*amortização*). A largura de banda é uma medida da taxa de transferência máxima disponível em um determinado link de rede. A largura de banda utilizável refere-se à parte da largura de banda que é realmente disponível para o aplicativo.

Os estilos afetam o desempenho da rede por sua influência no número de interações por ação do usuário e a granularidade dos elementos de dados. Um estilo que incentiva os pequenos, interações fortemente tipadas serão eficientes em uma aplicação envolvendo pequenas transferências de dados entre componentes conhecidos, mas causará sobrecarga excessiva em aplicativos que envolvem grandes transferências de dados ou interfaces negociadas. Da mesma forma, um estilo que envolve a coordenação de vários componentes organizados para filtrar um grande fluxo de dados estará fora de lugar em um aplicativo que requer principalmente pequenas mensagens de controle.

#### 2.3.1.2 Desempenho percebido pelo usuário

O desempenho percebido pelo usuário difere do desempenho da rede, pois o desempenho do uma ação é medida em termos de seu impacto no usuário na frente de um aplicativo, em vez de do que a taxa na qual a rede move as informações. As principais medidas para o usuário desempenho percebido são a latência e o tempo de conclusão.

A latência é o período de tempo entre o estímulo inicial e a primeira indicação de um resposta. A latência ocorre em vários pontos no processamento de um ação do aplicativo: 1) o tempo necessário para o aplicativo reconhecer o evento que iniciou a ação; 2) o tempo necessário para configurar as interações entre os componentes; 3) o tempo necessário para transmitir cada interação aos componentes; 4) o tempo necessário para

processar cada interação nesses componentes; e, 5) o tempo necessário para completar transferência e processamento suficientes do resultado das interações antes que o aplicativo seja capaz de começar a renderizar um resultado utilitário. É importante notar que, embora apenas (3) e (5) representam a comunicação de rede real, todos os cinco pontos podem ser afetados pelo estilo arquitetônico. Além disso, várias interações de componentes por ação do usuário são aditivo à latência, a menos que ocorram em paralelo.

A conclusão é a quantidade de tempo necessária para concluir uma ação do aplicativo. Conclusão tempo depende de todas as medidas acima mencionadas. A diferença entre um o tempo de conclusão da ação e sua latência representam o grau em que o aplicativo é processando incrementalmente os dados que estão sendo recebidos. Por exemplo, um navegador da Web que pode renderizar uma imagem grande enquanto ela está sendo recebida fornece uma percepção do usuário significativamente melhor desempenho do que aquele que espera até que a imagem inteira seja completamente recebida antes de renderização, mesmo que ambos tenham o mesmo desempenho de rede.

É importante observar que as considerações de design para otimizar a latência geralmente terão o efeito colateral da degradação do tempo de conclusão e vice-versa. Por exemplo, compressão de um fluxo de dados pode produzir uma codificação mais eficiente se o algoritmo amostrar um parte dos dados antes de produzir a transformação codificada, resultando em um tempo de conclusão para transferir os dados codificados pela rede. No entanto, se isso a compactação está sendo executada dinamicamente em resposta a uma ação do usuário e, em seguida, armazena em buffer um amostra grande antes da transferência pode produzir uma latência inaceitável. Equilibrando essas trocas offs pode ser difícil, principalmente quando não se sabe se o destinatário se importa mais sobre latência (por exemplo, navegadores da Web) ou conclusão (por exemplo, aranhas da Web).

#### **2.3.1.3 Eficiência da Rede**

Uma observação interessante sobre aplicativos baseados em rede é que a melhor desempenho é obtido por não usar a rede. Isso significa essencialmente que o mais estilos arquiteturais eficientes para um aplicativo baseado em rede são aqueles que podem efetivamente minimizar o uso da rede quando possível, por meio da reutilização de interações (caching), redução da frequência de interações de rede em relação a ações do usuário (dados replicados e operação desconectada), ou eliminando a necessidade de algumas interações movendo o processamento de dados para mais perto da fonte dos dados (código móvel).

O impacto dos vários problemas de desempenho geralmente está relacionado ao escopo do distribuição para o aplicativo. Os benefícios de um estilo sob condições locais podem se tornar desvantagens quando confrontados com as condições globais. Assim, as propriedades de um estilo devem ser enquadrado em relação à distância de interação: dentro de um único processo, entre processos em um único host, dentro de uma rede local (LAN) ou espalhados por uma rede de longa distância (WAN). Preocupações adicionais tornam-se evidentes quando as interações em uma WAN, onde um uma única organização está envolvida, são comparados a interações através da Internet, envolvendo múltiplos limites de confiança.

#### **2.3.2 Escalabilidade**

Escalabilidade refere-se à capacidade da arquitetura de suportar um grande número de componentes, ou interações entre componentes, dentro de uma configuração ativa. A escalabilidade pode ser aprimorado simplificando componentes, distribuindo serviços em muitos componentes (descentralizando as interações), e controlando as interações e configurações como um

resultado do monitoramento. Os estilos influenciam esses fatores determinando a localização de estado do aplicativo, a extensão da distribuição e o acoplamento entre os componentes.

A escalabilidade também é afetada pela frequência das interações, seja a carga em um componente é distribuído uniformemente ao longo do tempo ou ocorre em picos, seja uma interação requer entrega garantida ou um melhor esforço, se uma solicitação envolve síncrona ou manipulação assíncrona e se o ambiente é controlado ou anárquico (ou seja, pode você confia nos outros componentes?).

### **2.3.3 Simplicidade**

O principal meio pelo qual os estilos arquitetônicos induzem a simplicidade é aplicando a princípio da separação de interesses para a alocação de funcionalidade dentro dos componentes. Se funcionalidade pode ser alocada de modo que os componentes individuais sejam substancialmente menos complexos, então eles serão mais fáceis de entender e implementar. Da mesma forma, essa separação facilita a tarefa de raciocinar sobre a arquitetura geral. Eu optei por juntar o qualidades de complexidade, comprehensibilidade e verificabilidade sob a propriedade geral de simplicidade, uma vez que andam de mãos dadas para um sistema baseado em rede.

A aplicação do princípio da generalidade aos elementos arquitetônicos também melhora simplicidade, uma vez que diminui a variação dentro de uma arquitetura. Generalidade dos conectores leva ao middleware [22].

### **2.3.4 Modificabilidade**

Modificabilidade é sobre a facilidade com que uma mudança pode ser feita em um aplicativo arquitetura. A modificabilidade pode ser ainda dividida em capacidade de evolução, extensibilidade, personalização, configurabilidade e reutilização, conforme descrito abaixo. Uma preocupação especial

de sistemas baseados em rede é a modificabilidade dinâmica [98], onde a modificação é feita para um aplicativo implantado sem parar e reiniciar todo o sistema.

Mesmo que fosse possível construir um sistema de software que combinasse perfeitamente com o requisitos de seus usuários, esses requisitos mudarão ao longo do tempo, assim como a sociedade muda. hora extra. Como os componentes que participam de um aplicativo baseado em rede podem ser distribuído por vários limites organizacionais, o sistema deve estar preparado para mudança gradual e fragmentada, onde coexistem velhas e novas implementações, sem impedindo que as novas implementações façam uso de seus recursos estendidos.

#### *2.3.4.1 Evolução*

Evolução representa o grau em que uma implementação de componente pode ser alterada sem afetar negativamente outros componentes. Evolução estática de componentes geralmente depende de quão bem a abstração arquitetural é aplicada pela implementação, e portanto, não é algo exclusivo de qualquer estilo arquitetônico em particular. Evolução dinâmica, no entanto, pode ser influenciado pelo estilo se incluir restrições na manutenção e localização do estado do aplicativo. As mesmas técnicas usadas para se recuperar de uma falha parcial condições em um sistema distribuído [133] podem ser usadas para suportar a evolução dinâmica.

#### *2.3.4.2 Extensibilidade*

A extensibilidade é definida como a capacidade de adicionar funcionalidade a um sistema [108]. Dinâmico extensibilidade implica que a funcionalidade pode ser adicionada a um sistema implantado sem afetando o resto do sistema. A extensibilidade é induzida dentro de um estilo arquitetônico por reduzindo o acoplamento entre componentes, como exemplificado pela integração baseada em eventos.

#### *2.3.4.3 Personalização*

A customização refere-se à habilidade de especializar temporariamente o comportamento de um elemento arquitetônico, de modo que possa então realizar um serviço incomum. Um componente é personalizável se puder ser estendido por um cliente dos serviços desse componente sem impactando negativamente outros clientes desse componente [50]. Estilos que suportam a personalização também podem melhorar a simplicidade e a escalabilidade, uma vez que os componentes de serviço podem ser reduzidos em tamanho e complexidade implementando diretamente apenas os mais frequentes serviços e permitindo que serviços pouco frequentes sejam definidos pelo cliente. A customização é uma propriedade induzida pela avaliação remota e estilos de código sob demanda.

#### *2.3.4.4 Configurabilidade*

A configurabilidade está relacionada tanto à extensibilidade quanto à reutilização, pois se refere à modificação de implantação de componentes, ou configurações de componentes, de modo que eles são capazes de usar um novo serviço ou tipo de elemento de dados. O pipe-and-filter e code-on estilos de demanda são dois exemplos que induzem a configurabilidade de configurações e componentes, respectivamente.

#### *2.3.4.5 Reutilização*

A reutilização é uma propriedade de uma arquitetura de aplicativo se seus componentes, conectores ou elementos de dados podem ser reutilizados, sem modificação, em outras aplicações. O primário mecanismo para induzir a reutilização dentro de estilos arquitetônicos é a redução do acoplamento (conhecimento de identidade) entre componentes e restringindo a generalidade de interfaces de componentes. O estilo uniforme de tubo e filtro exemplifica esses tipos de restrições.

### **2.3.5 Visibilidade**

Os estilos também podem influenciar a visibilidade das interações em um aplicativo baseado em rede restringindo interfaces via generalidade ou fornecendo acesso ao monitoramento. Visibilidade neste caso refere-se à capacidade de um componente para monitorar ou mediar a interação entre dois outros componentes. A visibilidade pode permitir um desempenho aprimorado por meio do cache compartilhado de interações, escalabilidade por meio de serviços em camadas, confiabilidade por meio de monitoramento reflexivo, e segurança, permitindo que as interações sejam inspecionadas por mediadores (por exemplo, rede firewalls). O estilo de agente móvel é um exemplo onde a falta de visibilidade pode levar a preocupações com segurança.

Esse uso do termo visibilidade difere daquele em Ghezzi et al. [58], onde estão referindo-se à visibilidade do processo de desenvolvimento em vez do produto.

### **2.3.6 Portabilidade**

O software é portátil se puder ser executado em diferentes ambientes [58]. Estilos que induzem portabilidade incluem aqueles que movem o código junto com os dados a serem processados, como o estilos de máquina virtual e agente móvel, e aqueles que restringem os elementos de dados a um conjunto de formatos padronizados.

### **2.3.7 Confiabilidade**

A confiabilidade, dentro da perspectiva das arquiteturas de aplicativos, pode ser vista como a grau em que uma arquitetura é suscetível a falhas no nível do sistema na presença de falhas parciais em componentes, conectores ou dados. Os estilos podem melhorar a confiabilidade evitando pontos únicos de falha, permitindo redundância, permitindo monitoramento ou reduzindo o escopo da falha a uma ação recuperável.

## 2.4 Resumo

Este capítulo examinou o escopo da dissertação, concentrando-se em redes baseadas em arquiteturas de aplicativos e descrevendo como os estilos podem ser usados para guiar suas arquiteturas Projeto. Também definiu o conjunto de propriedades arquitetônicas que serão utilizadas ao longo do resto da dissertação para a comparação e avaliação de estilos arquitectónicos.

O próximo capítulo apresenta um levantamento de estilos arquiteturais comuns para sistemas baseados em rede. software aplicativo dentro de uma estrutura de classificação que avalia cada estilo de acordo com às propriedades arquitetônicas que induziria se aplicado a uma arquitetura para um protótipo de sistema hipermídia baseado em rede.

## CAPÍTULO 3

### Estilos de arquitetura baseados em rede

Este capítulo apresenta um levantamento de estilos arquiteturais comuns para sistemas baseados em rede. software aplicativo dentro de uma estrutura de classificação que avalia cada estilo de acordo com às propriedades arquitetônicas que induziria se aplicado a uma arquitetura para um protótipo de sistema hipermédia baseado em rede.

#### 3.1 Metodologia de Classificação

O propósito de construir software não é criar uma topologia específica de interações ou usar um determinado tipo de componente - é criar um sistema que atenda ou exceda o necessidades de aplicação. Os estilos arquitetônicos escolhidos para o projeto de um sistema devem estar de acordo com essas necessidades, e não o contrário. Portanto, a fim de fornecer um design útil orientação, uma classificação de estilos arquitetônicos deve ser baseada na arquitetura propriedades induzidas por esses estilos.

##### 3.1.1 Seleção de Estilos Arquitetônicos para Classificação

O conjunto de estilos arquitetônicos incluídos na classificação não é abrangente de todos os estilos de aplicativos baseados em rede possíveis. De fato, um novo estilo pode ser formado simplesmente adicionando uma restrição arquitetônica a qualquer um dos estilos pesquisados. Meu objetivo é descrever uma amostra representativa de estilos, particularmente aqueles já identificados dentro do literatura de arquitetura de software e fornecer uma estrutura pela qual outros estilos podem ser adicionados à classificação à medida que são desenvolvidos.

Excluí intencionalmente estilos que não melhoram a comunicação ou propriedades de interação quando combinadas com um dos estilos pesquisados para formar uma rede aplicativo baseado. Por exemplo, o estilo arquitetônico do quadro-negro [95] consiste em um repositório central e um conjunto de componentes (fontes de conhecimento) que operam oportunisticamente sobre o repositório. Uma arquitetura de quadro-negro pode ser estendida a um sistema baseado em rede distribuindo os componentes, mas as propriedades de tal extensão são inteiramente baseados no estilo de interação escolhido para suportar a distribuição — notificações via integração baseada em eventos, polling *a la* client-server ou replicação do repositório. Assim, não haveria valor agregado em incluí-lo na classificação, mesmo que o estilo híbrido seja compatível com rede.

### **3.1.2 Propriedades arquitetônicas induzidas pelo estilo**

Minha classificação usa mudanças relativas nas propriedades arquitetônicas induzidas por cada estilo como meio de ilustrar o efeito de cada estilo arquitetônico quando aplicado a um sistema para hipermídia distribuída. Observe que a avaliação de um estilo para uma determinada propriedade depende do tipo de interação do sistema que está sendo estudado, conforme descrito na Seção 2.2. As propriedades arquitetônicas são relativas no sentido de que adicionar uma restrição arquitetônica pode melhorar ou reduzir uma determinada propriedade, ou simultaneamente melhorar um aspecto da propriedade e reduzir algum outro aspecto da propriedade. Da mesma forma, melhorar uma propriedade pode levar à redução de outro.

Embora nossa discussão de estilos arquitetônicos inclua aqueles aplicáveis a uma ampla variedade de sistemas baseados em rede, nossa avaliação de cada estilo será baseada em seu impacto sobre uma arquitetura para um único tipo de software: sistemas hipermídia baseados em rede.

Focar em um tipo específico de software nos permite identificar as vantagens de um estilo sobre o outro da mesma forma que um projetista de um sistema avaliaria essas vantagens.

Como não pretendemos declarar nenhum estilo único como universalmente desejável para todos tipos de software, restringir o foco de nossa avaliação simplesmente reduz as dimensões sobre o qual precisamos avaliar. Avaliando os mesmos estilos para outros tipos de aplicação software é uma área aberta para pesquisas futuras.

### **3.1.3 Visualização**

Eu uso uma tabela de estilo versus propriedades arquitetônicas como a visualização primária para isso classificação. Os valores da tabela indicam a influência relativa que o estilo de uma determinada linha tem na propriedade de uma coluna. Os símbolos de menos (-) acumulam-se para influências negativas e símbolos de mais (+) para positivo, com mais-menos ( $\pm$ ) indicando que depende de alguns aspecto do domínio do problema. Embora esta seja uma simplificação grosseira dos detalhes apresentado em cada seção, ele indica o grau em que um estilo abordou (ou ignorado) uma propriedade arquitetônica.

Uma visualização alternativa seria um gráfico de derivação baseado em propriedades para classificação de estilos arquitetônicos. Os estilos seriam classificados de acordo com a forma como são derivados de outros estilos, com os arcos entre estilos ilustrados por propriedades adquiridas ou perdidas. O ponto de partida do gráfico seria o estilo nulo (sem restrições). É possível derivar tal gráfico diretamente das descrições.

### 3.2 Estilos de fluxo de dados

Estilo	Derivação		
PF	$\pm \text{+++} \text{++}$		
UPF	PF	$\ddot{\gamma} \pm$	$\text{++} \text{+} \text{+}$
			$\text{++} \text{++} \text{+}$

Tabela 3-1. Avaliação de estilos de fluxo de dados para hipermídia baseada em rede

#### 3.2.1 Tubulação e Filtro (PF)

Em um estilo pipe and filter, cada componente (filtro) lê fluxos de dados em suas entradas e produz fluxos de dados em suas saídas, geralmente ao aplicar uma transformação ao fluxos de entrada e processando-os de forma incremental para que a saída comece antes que a entrada seja completamente consumido [53]. Esse estilo também é conhecido como rede de fluxo de dados unidirecional [6]. A restrição é que um filtro deve ser completamente independente de outros filtros (zero acoplamento): ele não deve compartilhar estado, thread de controle ou identidade com os outros filtros em seu interfaces a montante e a jusante [53].

Abowd et al. [1] fornece uma extensa descrição formal do estilo de tubo e filtro usando a linguagem Z. O ambiente de desenvolvimento de software *Khoros* para imagem processamento [112] fornece um bom exemplo de uso do estilo pipe and filter para construir um gama de aplicações.

Garlan e Shaw [53] descrevem as propriedades vantajosas do estilo de tubo e filtro do seguinte modo. Primeiro, o PF permite que o projetista entenda a entrada/saída geral do sistema como uma composição simples dos comportamentos dos filtros individuais (simplicidade). Em segundo lugar, o PF suporta a reutilização: quaisquer dois filtros podem ser conectados, desde que concordem os dados que estão sendo transmitidos entre eles (reutilização). Terceiro, os sistemas PF podem ser fácil manutenção e aprimoramento: novos filtros podem ser adicionados aos sistemas existentes

(extensibilidade) e filtros antigos podem ser substituídos por outros melhorados (evolução). Quarto, eles permitem certos tipos de análise especializada (verificabilidade), como taxa de transferência e análise de impasse. Finalmente, eles naturalmente suportam a execução simultânea (percepção do usuário atuação).

As desvantagens do estilo PF incluem: atraso de propagação é adicionado através de longos pipelines, o processamento sequencial em lote ocorre se um filtro não puder processar incrementalmente seu entradas, e nenhuma interatividade é permitida. Um filtro não pode interagir com seu ambiente porque ele não pode saber que qualquer fluxo de saída em particular compartilha um controlador com qualquer determinado fluxo de entrada. Essas propriedades diminuem o desempenho percebido pelo usuário se o problema que está sendo tratado não se encaixa no padrão de um fluxo de dados.

Um aspecto dos estilos de PF que raramente é mencionado é que há um “invisível” implícito. hand” que organiza a configuração dos filtros para estabelecer a aplicação geral. Uma rede de filtros é normalmente organizada imediatamente antes de cada ativação, permitindo que a aplicativo para especificar a configuração de componentes de filtro com base na tarefa em mãos e a natureza dos fluxos de dados (configurabilidade). Esta função de controlador é considerada uma fase operacional separada do sistema e, portanto, uma arquitetura separada, embora Um não pode existir sem o outro.

### **3.2.2 Tubo e Filtro Uniforme (UPF)**

O estilo de tubo e filtro uniforme adiciona a restrição de que todos os filtros devem ter o mesmo interface. O principal exemplo desse estilo é encontrado no sistema operacional Unix, onde processos de filtro têm uma interface que consiste em um fluxo de dados de entrada de caracteres (stdin) e dois fluxos de dados de saída de caracteres (stdout e stderr). Restringindo a interface

também simplifica a tarefa de entender como um determinado filtro funciona.

Uma desvantagem da interface uniforme é que ela pode reduzir o desempenho da rede se os dados precisam ser convertidos de ou para seu formato natural.

### 3.3 Estilos de replicação

Tabela 3-2. Avaliação de estilos de replicação para hipermídia baseada em rede

### **3.3.1 Repositório Replicado (RR)**

Sistemas baseados no estilo de repositório replicado [6] melhoraram a acessibilidade dos dados e escalabilidade de serviços por ter mais de um processo fornecendo o mesmo serviço. Esses servidores descentralizados interagem para fornecer aos clientes a ilusão de que existe apenas um, serviço centralizado. Sistemas de arquivos distribuídos, como XMS [49], e versionamento remoto sistemas, como CVS [[www.cyclic.com](http://www.cyclic.com)], são os principais exemplos.

Melhor desempenho percebido pelo usuário é a principal vantagem, tanto pela redução do latência de solicitações normais e habilitando a operação desconectada em face do primário falha do servidor ou roaming intencional fora da rede. A simplicidade permanece neutra, uma vez que a complexidade da replicação é compensada pela economia de permitir componentes para operar de forma transparente em dados replicados localmente. Manter a consistência é a preocupação primordial.

### 3.3.2 Cache (\$)

Uma variante de repositório replicado é encontrada no estilo cache: replicação do resultado de uma pedido individual de modo que possa ser reutilizado por pedidos posteriores. Essa forma de replicação é encontrado com mais frequência nos casos em que o conjunto de dados em potencial excede em muito a capacidade de qualquer um cliente, como na WWW [20], ou onde o acesso completo ao repositório é desnecessário.

A replicação lenta ocorre quando os dados são replicados em uma resposta ainda não armazenada em cache para um solicitação, contando com a localidade de referência e comunalidade de interesse para propagar itens no cache para reutilização posterior. A replicação ativa pode ser realizada por pré-busca entradas em cache com base em solicitações antecipadas.

O armazenamento em cache fornece um pouco menos de melhoria do que o estilo de repositório replicado em termos de desempenho percebido pelo usuário, já que mais solicitações perderão o cache e apenas os dados acessados recentemente estarão disponíveis para operação desconectada. Por outro lado, cache é muito mais fácil de implementar, não requer tanto processamento e armazenamento, e é mais eficiente porque os dados são transmitidos apenas quando solicitados. O estilo de cache torna-se baseado em rede quando combinado com um estilo de servidor sem estado de cliente.

## 3.4 Estilos Hierárquicos

Estilo	Derivação										
CS		+++									
LS	ÿ++ ++										
LCS	CS+LS		++ + ++		+ +	+ +					
CSS	CS		++ + +			+ +	+ +				
C\$SS	CSS+\$ ÿ + + + + +				+ +	+ +					
LC\$SS	LCS+C\$SS ÿ ± + + + + +				++ + +						
RS	CS	+ÿ++									
RDA	CS	+ÿÿ				+ +					

Tabela 3-3. Avaliação de estilos hierárquicos para hipermídia baseada em rede

### 3.4.1 Cliente-Servidor (CS)

O estilo cliente-servidor é o mais frequentemente encontrado dos estilos de arquitetura para aplicativos baseados em rede. Um componente de servidor, oferecendo um conjunto de serviços, escuta solicitações sobre esses serviços. Um componente cliente, desejando que um serviço seja executado, envia uma solicitação ao servidor por meio de um conector. O servidor rejeita ou executa a solicitação e envia uma resposta de volta ao cliente. Uma variedade de sistemas cliente-servidor são pesquisado por Sinha [123] e Umar [131].

Andrews [6] descreve os componentes cliente-servidor da seguinte forma: Um cliente é um processo; um servidor é um processo reativo. Os clientes fazem solicitações que desencadeiam reações de servidores. Assim, um cliente inicia a atividade nos momentos de sua escolha; muitas vezes, em seguida, atrasa até que seu pedido foi atendido. Por outro lado, um servidor espera que os pedidos sejam feitos e então reage a eles. Um servidor geralmente é um processo sem término e geralmente fornece atendimento a mais de um cliente.

A separação de interesses é o princípio por trás das restrições cliente-servidor. Um bom separação de funcionalidade deve simplificar o componente do servidor para melhorar escalabilidade. Essa simplificação geralmente assume a forma de mover toda a interface do usuário funcionalidade no componente cliente. A separação também permite que os dois tipos de componentes evoluam independentemente, desde que a interface não mude.

A forma básica de cliente-servidor não restringe como o estado do aplicativo é particionado entre os componentes cliente e servidor. É muitas vezes referido pelos mecanismos usados para a implementação do conector, como chamada de procedimento remoto [23] ou orientada a mensagens middleware [131].

### **3.4.2 Sistema em camadas (LS) e servidor de cliente em camadas (LCS)**

Um sistema em camadas é organizado hierarquicamente, cada camada fornecendo serviços para a camada acima dela e usando serviços da camada abaixo dela [53]. Embora o sistema em camadas seja considerado um estilo “puro”, seu uso em sistemas baseados em rede é limitado combinação com o estilo cliente-servidor para fornecer camadas-cliente-servidor.

Os sistemas em camadas reduzem o acoplamento em várias camadas, ocultando as camadas internas de todos, exceto a camada externa adjacente, melhorando assim a capacidade de evolução e reutilização. Exemplos incluem o processamento de protocolos de comunicação em camadas, como o protocolo TCP/IP e pilhas de protocolo OSI [138], e bibliotecas de interface de hardware. A principal desvantagem de sistemas em camadas é que eles adicionam sobrecarga e latência ao processamento de dados, reduzindo o desempenho percebido pelo usuário [32].

Layered-client-server adiciona componentes de proxy e gateway ao estilo cliente-servidor. UMA proxy [116] atua como um servidor compartilhado para um ou mais componentes clientes, recebendo requisições e

encaminhando-os, com possível tradução, para os componentes do servidor. Um componente de gateway parece ser um servidor normal para clientes ou proxies que solicitam seus serviços, mas na verdade é encaminhando essas solicitações, com possível tradução, para seus servidores de “camada interna”. Esses componentes mediadores adicionais podem ser adicionados em várias camadas para adicionar recursos como carga balanceamento e verificação de segurança do sistema.

Arquiteturas baseadas em camadas-cliente-servidor são chamadas de duas camadas, três camadas, ou arquiteturas multicamadas na literatura de sistemas de informação [131].

O LCS também é uma solução para gerenciamento de identidade em um sistema distribuído de grande escala, onde conhecimento completo de todos os servidores seria proibitivamente caro. Em vez disso, os servidores são organizados em camadas de tal forma que os serviços raramente usados são tratados por intermediários em vez de diretamente por cada cliente [6].

### **3.4.3 Cliente-Servidor sem Estado (CSS)**

O estilo cliente-servidor sem estado deriva de cliente-servidor com a restrição adicional que nenhum *estado de sessão* é permitido no componente do servidor. Cada solicitação do cliente para servidor deve conter todas as informações necessárias para entender a solicitação e não pode tirar proveito de qualquer contexto armazenado no servidor. O estado da sessão é mantido inteiramente no cliente.

Essas restrições melhoraram as propriedades de visibilidade, confiabilidade e escalabilidade. A visibilidade é melhorada porque um sistema de monitoramento não precisa ir além de um único datum do pedido, a fim de determinar a natureza completa do pedido. A confiabilidade é melhorada porque facilita a tarefa de recuperação de falhas parciais [133]. A escalabilidade é melhorada

porque não ter que armazenar o estado entre as solicitações permite que o componente do servidor recursos gratuitos e simplifica ainda mais a implementação.

A desvantagem do client-stateless-server é que ele pode diminuir desempenho aumentando os dados repetitivos (sobrecarga por interação) enviados em uma série de solicitações, pois esses dados não podem ser deixados no servidor em um contexto compartilhado.

#### **3.4.4 Cliente-Cache-Stateless-Servidor (C\$SS)**

O estilo client-cache-stateless-server deriva do client-stateless-server e do cache estilos através da adição de componentes de cache. Um cache atua como um mediador entre cliente e servidor no qual as respostas às requisições anteriores podem, caso sejam consideradas armazenáveis em cache, ser reutilizado em resposta a solicitações posteriores que são equivalentes e provavelmente resultarão em uma resposta idêntica ao do cache se a solicitação fosse encaminhada ao servidor. Um exemplo sistema que faz uso efetivo deste estilo é o NFS da Sun Microsystems [115].

A vantagem de adicionar componentes de cache é que eles têm o potencial de ou eliminar completamente algumas interações, melhorando a eficiência e percepção do usuário atuação.

#### **3.4.5 Servidor de cache de cliente em camadas sem estado (LC\$SS)**

O estilo layered-client-cache-stateless-server deriva de layered-client-server e client-cache-stateless-server através da adição de componentes proxy e/ou gateway.

Um exemplo de sistema que usa um estilo LC\$SS é o sistema de nomes de domínio da Internet (DNS).

As vantagens e desvantagens de LC\$SS são simplesmente uma combinação daquelas para LCS e C\$SS. No entanto, observe que não contamos as contribuições do estilo CS duas vezes, já que os benefícios não são aditivos se forem provenientes da mesma derivação ancestral.

### **3.4.6 Sessão Remota (RS)**

O estilo de sessão remota é uma variante do cliente-servidor que tenta minimizar a complexidade, ou maximizar a reutilização, dos componentes do cliente em vez do servidor componente. Cada cliente inicia uma sessão no servidor e então invoca uma série de services no servidor, finalmente saindo da sessão. O estado do aplicativo é mantido inteiramente no servidor. Este estilo é normalmente usado quando se deseja acessar um serviço remoto usando um cliente genérico (por exemplo, TELNET [106]) ou através de uma interface que imita um cliente genérico (por exemplo, FTP [107]).

As vantagens do estilo de sessão remota são que é mais fácil manter centralmente o interface no servidor, reduzindo preocupações sobre inconsistências em clientes implantados quando a funcionalidade é estendida e melhora a eficiência se as interações fizerem uso de contexto de sessão no servidor. As desvantagens são que ele reduz a escalabilidade do servidor, devido ao estado do aplicativo armazenado, e reduz a visibilidade das interações, uma vez que um monitor teria que saber o estado completo do servidor.

### **3.4.7 Acesso Remoto a Dados (RDA)**

O estilo de acesso remoto a dados [131] é uma variante de cliente-servidor que espalha a aplicação estado entre cliente e servidor. Um cliente envia uma consulta de banco de dados em um formato padrão, como SQL, para um servidor remoto. O servidor aloca um espaço de trabalho e realiza a consulta, o que pode resultar em um conjunto de dados muito grande. O cliente pode então fazer outras operações sobre o conjunto de resultados (como junções de tabela) ou recuperar o resultado uma parte de cada vez. O cliente deve conhecer a estrutura de dados do serviço para construir consultas dependentes de estrutura.

As vantagens do acesso remoto a dados são que um grande conjunto de dados pode ser iterativamente reduzido no lado do servidor sem transmiti-lo pela rede, melhorando eficiência e a visibilidade é melhorada usando uma linguagem de consulta padrão. O desvantagens são que o cliente precisa entender a mesma manipulação de banco de dados conceitos como a implementação do servidor (falta de simplicidade) e armazenamento do contexto da aplicação no servidor diminui a escalabilidade. A confiabilidade também sofre, pois uma falha parcial pode deixar a área de trabalho em um estado desconhecido. Mecanismos de transação (por exemplo, confirmação de duas fases) pode ser usado para corrigir o problema de confiabilidade, embora a um custo de complexidade e sobrecarga de interação.

### 3.5 Estilos de código móvel

Os estilos de código móvel usam a mobilidade para alterar dinamicamente a distância entre os processamento e origem dos dados ou destino dos resultados. Esses estilos são amplamente examinado em Fuggetta et al. [50]. Uma abstração de site é introduzida no nível arquitetural, como parte da configuração ativa, a fim de levar em conta a localização dos diferentes componentes. A introdução do conceito de localização permite modelar o custo de uma interação entre os componentes no nível do projeto. Em particular, uma interação entre componentes que compartilham a mesma localização são considerados de custo insignificante quando comparado a uma interação envolvendo comunicação através da rede. Ao alterar sua localização, um componente pode melhorar a proximidade e a qualidade de sua interação, reduzindo os custos de interação e, assim, melhorando a eficiência e percepção do usuário. atuação.

Em todos os estilos de código móvel, um elemento de dados é dinamicamente transformado em um componente. Fuggeta et al. [50] usam uma análise que compara o tamanho do código como um dado elemento para a economia na transferência normal de dados, a fim de determinar se a mobilidade é desejável para uma determinada ação. Isso seria impossível de modelar a partir de uma arquitetura ponto de vista se a definição de arquitetura de software exclui elementos de dados.

Estilo	Derivação				
VM		±	+		ÿ+
REV	CS+VM +ÿ± ++ ÿ+ÿ				
BACALHAU	CS + VM +++± +			+	
LCODC\$SS LC\$SS+COD	ÿ ++ ++ +4+ +-+ +-+			+ + ± + +	
MA REV+COD	+ ++	±	++ + +		+

Tabela 3-4. Avaliação de estilos de código móvel para hipermídia baseada em rede

### 3.5.1 Máquina Virtual (VM)

Subjacente a todos os estilos de código móvel está a noção de uma máquina virtual, ou intérprete, estilo [53]. O código deve ser executado de alguma forma, preferencialmente dentro de um ambiente para satisfazer as preocupações de segurança e confiabilidade, que é exatamente o que o virtual estilo de máquina fornece. Não é, em si, um estilo baseado em rede, mas é comumente usado como tal quando combinado com um componente no estilo cliente-servidor (REV e COD estilos).

As máquinas virtuais são comumente usadas como mecanismo para linguagens de script, incluindo linguagens de propósito geral como Perl [134] e linguagens de tarefas específicas como PostScript [2]. Os principais benefícios são a separação entre instrução e implementação em um plataforma específica (portabilidade) e facilidade de extensibilidade. A visibilidade é reduzida porque é difícil saber o que um executável fará simplesmente olhando para o código. Simplicidade é

reduzido devido à necessidade de gerenciar o ambiente de avaliação, mas isso pode ser compensado em alguns casos como resultado da simplificação da funcionalidade estática.

### **3.5.2 Avaliação Remota (REV)**

No estilo de avaliação remota [50], derivado do cliente-servidor e da máquina virtual estilos, um componente cliente possui o know-how necessário para executar um serviço, mas não possui os recursos (ciclos de CPU, fonte de dados, etc.) necessários, que estão localizados em um local remoto. Consequentemente, o cliente envia o know-how para um componente do servidor no local remoto. site, que por sua vez executa o código usando os recursos disponíveis lá. Os resultados disso execução são então enviados de volta ao cliente. O estilo de avaliação remota assume que o código fornecido será executado em um ambiente protegido, de modo que não afete outros clientes do mesmo servidor além dos recursos que estão sendo usados.

As vantagens da avaliação remota incluem a capacidade de personalizar o servidor serviços do componente, que fornece extensibilidade e personalização aprimoradas, e melhor eficiência quando o código pode adaptar suas ações ao ambiente dentro do servidor (em oposição ao cliente fazer uma série de interações para fazer o mesmo). Simplicidade é reduzida devido à necessidade de gerenciar o ambiente de avaliação, mas isso pode ser compensado em alguns casos como resultado da simplificação da funcionalidade do servidor estático.

A escalabilidade é reduzida; isso pode ser melhorado com o gerenciamento do servidor do ambiente de execução (eliminando código de longa execução ou de uso intensivo de recursos quando os recursos são apertados), mas a própria função de gestão leva a dificuldades quanto à falha parcial e confiabilidade. A limitação mais significativa, no entanto, é a falta de visibilidade devido à

cliente enviando código em vez de consultas padronizadas. A falta de visibilidade leva a problemas de implantação se o servidor não puder confiar nos clientes.

### **3.5.3 Código sob Demanda (COD)**

No estilo code-on-demand [50], um componente cliente tem acesso a um conjunto de recursos, mas não o know-how sobre como processá-los. Ele envia uma solicitação a um servidor remoto para o código que representa esse know-how, recebe esse código e o executa localmente.

As vantagens do código sob demanda incluem a capacidade de adicionar recursos a um cliente, que fornece extensibilidade e configurabilidade aprimoradas e melhor desempenho e eficiência percebidos quando o código consegue adequar suas ações às necessidades do cliente. ambiente e interagir com o usuário localmente em vez de por meio de interações remotas.

A simplicidade é reduzida devido à necessidade de gerenciar o ambiente de avaliação, mas isso pode ser compensado em alguns casos como resultado da simplificação da funcionalidade estática do cliente.

A escalabilidade do servidor é melhorada, pois pode descarregar o trabalho para o cliente que caso contrário, consumiram seus recursos. Assim como a avaliação remota, o mais significativo limite é a falta de visibilidade devido ao código de envio do servidor em vez de dados simples.

A falta de visibilidade leva a problemas óbvios de implantação se o cliente não puder confiar no servidores.

### **3.5.4 Servidor sem estado com código sob demanda em camadas (LCODC\$SS)**

Como exemplo de como algumas arquiteturas são complementares, considere a adição de code-on-demand para o estilo layered-client-cache-stateless-server discutido acima. Desde o código pode ser tratado como apenas mais um elemento de dados, isso não interfere na

vantagens do estilo LC\$SS. Um exemplo é o navegador da Web HotJava [java.sun.com], que permite que applets e extensões de protocolo sejam baixados como mídia digitada.

As vantagens e desvantagens do LCODC\$SS são apenas uma combinação daquelas para COD e LC\$SS. Poderíamos ir mais longe e discutir a combinação de COD com outros CS estilos, mas esta pesquisa não pretende ser exaustiva (nem exaustiva).

### **3.5.5 Agente Móvel (MA)**

No estilo de agente móvel [50], um componente computacional inteiro é movido para um site, juntamente com seu estado, o código que ele precisa e possivelmente alguns dados necessários para executar a tarefa. Isso pode ser considerado uma derivação da avaliação remota e código sob demanda estilos, já que a mobilidade funciona nos dois sentidos.

A principal vantagem do estilo de agente móvel, além das já descritas para REV e COD, é que há maior dinamismo na seleção de quando mover o código. Um aplicativo pode estar no meio do processamento de informações em um local quando decide se mudar para outro local, presumivelmente para reduzir a distância entre ele e o próximo conjunto de dados que deseja processar. Além disso, o problema de confiabilidade de a falha é reduzida porque o estado do aplicativo está em um local por vez [50].

### 3.6 Estilos ponto a ponto

Estilo	Derivação								
EBI	+ Ÿ Ÿ ± + +								++
C2	EBI+LCS	+	+ + +					+ + + ± + ±	
FAZ	CS + CS Ÿ	+		+ + + Ÿ Ÿ					
BDO DO+LCS Ÿ Ÿ			++ +		+ + + Ÿ				+

Tabela 3-5. Avaliação de estilos peer-to-peer para hipermídia baseada em rede

#### 3.6.1 Integração baseada em eventos (EBI)

O estilo de integração baseado em eventos, também conhecido como invocação implícita ou sistema de eventos estilo, reduz o acoplamento entre os componentes, eliminando a necessidade de identidade no interface do conector. Em vez de chamar outro componente diretamente, um componente pode anunciar (ou transmitir) um ou mais eventos. Outros componentes em um sistema podem registrar interesse nesse tipo de evento e, quando o evento é anunciado, o próprio sistema invoca todos os componentes registrados [53]. Exemplos incluem o Model-View-Controller paradigma em Smalltalk-80 [72] e os mecanismos de integração de muitos softwares ambientes de engenharia, incluindo Field [113], SoftBench [29] e Polylith [110].

O estilo de integração baseado em eventos fornece forte suporte para extensibilidade por meio do facilidade de adicionar novos componentes que ouvem eventos, para reutilização, incentivando uma interface de eventos e mecanismo de integração, e para evolução, permitindo que os componentes ser substituídos sem afetar as interfaces de outros componentes [53]. Como tubo e filtro sistemas, uma arquitetura de configuração de alto nível é necessária para a “mão invisível” que coloca componentes na interface de eventos. A maioria dos sistemas EBI também inclui invocação como forma complementar de interação [53]. Para aplicações que são

dominado pelo monitoramento de dados, em vez de recuperação de dados, o EBI pode melhorar a eficiência eliminando a necessidade de interações de votação.

A forma básica do sistema EBI consiste em um barramento de eventos no qual todos os componentes escutam para eventos de seu interesse. Claro, isso leva imediatamente a problemas de escalabilidade com em relação ao número de notificações, tempestades de eventos e outros componentes transmitidos como resultado de eventos causados por essa notificação, e um único ponto de falha na notificação sistema de entrega. Isso pode ser melhorado através do uso de sistemas em camadas e filtragem de eventos, ao custo da simplicidade.

Outras desvantagens dos sistemas EBI são que pode ser difícil prever o que acontecem em resposta a uma ação (má compreensão) e as notificações de eventos não são adequado para troca de dados de grande granularidade [53]. Além disso, não há suporte para recuperação de falha parcial.

### **3.6.2 C2**

O estilo de arquitetura C2 [128] é direcionado ao suporte à reutilização de grãos grandes e flexibilidade composição dos componentes do sistema reforçando a independência do substrato. Faz isso por combinando integração baseada em eventos com servidor-cliente em camadas. Notificação assíncrona mensagens caindo e mensagens de solicitação assíncrona subindo, são os únicos meios de comunicação intercomponente. Isso impõe um baixo acoplamento de dependência em camadas (solicitações de serviço podem ser ignoradas) e acoplamento zero com níveis mais baixos (sem conhecimento do uso de notificações), melhorando o controle sobre o sistema sem perder a maior parte das vantagens do EBI.

Notificações são anúncios de uma mudança de estado dentro de um componente. C2 não restringir o que deve ser incluído com uma notificação: um sinalizador, um delta de mudança de estado ou um representação completa do estado são todas as possibilidades. A principal responsabilidade de um conector é o encaminhamento e difusão de mensagens; sua responsabilidade secundária é a filtragem de mensagens. A introdução da filtragem de mensagens em camadas resolve os problemas de EBI com escalabilidade, enquanto melhora a capacidade de evolução e reutilização também. Conectores pesados que incluem recursos de monitoramento podem ser usados para melhorar a visibilidade e reduzir a confiabilidade problemas de falha parcial.

### **3.6.3 Objetos Distribuídos**

O estilo de objetos distribuídos organiza um sistema como um conjunto de componentes interagindo como pares. Um objeto é uma entidade que encapsula algumas informações ou dados de estado privado, um conjunto de operações ou procedimentos associados que manipulam os dados e, possivelmente, um segmento de controle, para que coletivamente possam ser considerados uma única unidade [31]. Em geral, um o estado do objeto está completamente oculto e protegido de todos os outros objetos. A única maneira que pode ser examinado ou modificado é fazendo uma solicitação ou invocação em um dos operações acessíveis ao público. Isso cria uma interface bem definida para cada objeto, permitindo que a especificação das operações de um objeto seja tornada pública ao mesmo tempo tempo mantendo a implementação de suas operações e a representação de seu estado informação privada, melhorando assim a capacidade de evolução.

Uma operação pode invocar outras operações, possivelmente em outros objetos. Essas operações pode, por sua vez, fazer invocações a outros, e assim por diante. Uma cadeia de invocações relacionadas é referido como uma acção [31]. O estado é distribuído entre os objetos. Isso pode ser

vantajosa em termos de manter o estado onde é mais provável que esteja atualizado, mas tem a desvantagem de que é difícil obter uma visão geral da atividade do sistema (pobres visibilidades).

Para que um objeto interaja com outro, ele deve conhecer a identidade desse outro objeto. Quando a identidade de um objeto muda, é necessário modificar todos os outros objetos que o invocam explicitamente [53]. Deve haver algum objeto controlador que seja responsável por manter o estado do sistema para completar os requisitos do aplicativo. Central problemas para sistemas de objetos distribuídos incluem: gerenciamento de objetos, interação de objetos gestão e gestão de recursos [31].

Os sistemas de objetos são projetados para isolar os dados que estão sendo processados. Como consequência, fluxo de dados não é suportado em geral. No entanto, isso fornece um melhor suporte para mobilidade de objetos quando combinado com o estilo de agente móvel.

### **3.6.4 Objetos Distribuídos Intermediados**

Para reduzir o impacto da identidade, os modernos sistemas de objetos distribuídos normalmente usam um ou mais estilos intermediários para facilitar a comunicação. Isso inclui eventos baseados em integração e intermediado cliente/servidor [28]. O estilo de objeto distribuído intermediado apresenta componentes de resolução de nomes cuja finalidade é responder a solicitações de objetos do cliente para nomes de serviços gerais com o nome específico de um objeto que satisfará a solicitação.

Apesar de melhorar a capacidade de reutilização e evolução, o nível extra de indireção requer interações de rede adicionais, reduzindo a eficiência e o desempenho percebido pelo usuário.

Os sistemas de objetos distribuídos intermediados são atualmente dominados pela indústria desenvolvimento de padrões de CORBA dentro da OMG [97] e os padrões internacionais desenvolvimento de Processamento Distribuído Aberto (ODP) dentro da ISO/IEC [66].

Apesar de todo o interesse associado a objetos distribuídos, eles se saem mal quando em comparação com a maioria dos outros estilos de arquitetura baseados em rede. São mais utilizados para aplicativos que envolvem a invocação remota de serviços encapsulados, como hardware dispositivos, onde a eficiência e a frequência das interações de rede são menos preocupantes.

### 3.7 Limitações

Cada estilo arquitetônico promove um certo tipo de interação entre os componentes. Quando componentes são distribuídos em uma rede de longa distância, uso ou mau uso da rede impulsiona a usabilidade do aplicativo. Ao caracterizar os estilos por sua influência na arquitetura propriedades, e particularmente no desempenho de aplicativos baseados em rede de um sistema hipermídia, ganhamos a capacidade de escolher melhor um projeto de software que seja apropriado para a aplicação. Existem, no entanto, algumas limitações com a escolha classificação.

A primeira limitação é que a avaliação é específica para as necessidades de hipermídia. Por exemplo, muitas das boas qualidades do estilo tubo e filtro desaparecem se a comunicação for mensagens de controle de baixa granularidade, e não são aplicáveis se o comunicação requer interatividade do usuário. Da mesma forma, o cache em camadas só aumenta a latência, sem nenhum benefício, se nenhuma das respostas às solicitações do cliente puder ser armazenada em cache. Esse tipo de distinção não aparece na classificação, e é tratada apenas informalmente na discussão de cada estilo. Acredito que essa limitação pode ser superada criando

tabelas de classificação para cada tipo de problema de comunicação. Exemplos de áreas problemáticas incluiria, entre outros, recuperação de dados de grãos grandes, monitoramento remoto de informações, pesquisa, sistemas de controle remoto e processamento distribuído.

Uma segunda limitação é com o agrupamento de propriedades arquitetônicas. Em alguns casos, é melhor identificar os aspectos específicos de, por exemplo, comprehensibilidade e verificabilidade induzidas por um estilo arquitetônico, em vez de agrupá-las sob a rubrica de simplicidade. Este é particularmente o caso de estilos que podem melhorar a verificabilidade no custa da compreensão. No entanto, a noção mais abstrata de propriedade também tem valor como uma métrica única, pois não queremos tornar a classificação tão específica que não dois estilos impactam a mesma categoria. Uma solução seria uma classificação que apresentasse as propriedades específicas e uma propriedade de resumo.

Independentemente disso, este levantamento e classificação inicial é um pré-requisito necessário para qualquer outras classificações que possam abordar suas limitações.

### **3.8 Trabalho Relacionado**

#### **3.8.1 Classificação de Estilos e Padrões Arquitetônicos**

A área de pesquisa mais diretamente relacionada a este capítulo é a identificação e classificação de estilos arquitetônicos e padrões de nível de arquitetura.

Shaw [117] descreve alguns estilos arquitetônicos, posteriormente expandidos em Garlan e Shaw [53]. Uma classificação preliminar desses estilos é apresentada em Shaw e Clements [122] e repetido em Bass et al. [9], em que uma estratégia de classificação tabular bidimensional é usado com questões de controle e dados como eixos primários, organizados pelos seguintes categorias de recursos: quais tipos de componentes e conectores são usados no estilo;

como o controle é compartilhado, alocado e transferido entre os componentes; como os dados são comunicados pelo sistema; como os dados e o controle interagem; e, que tipo de raciocínio é compatível com o estilo. O objetivo principal da taxonomia é identificar características de estilo, ao invés de auxiliar na sua comparação. Termina com um pequeno conjunto de “regras de ouro” como uma forma de orientação de design

Ao contrário deste capítulo, a classificação de Shaw e Clements [122] não auxilia na avaliar designs de uma maneira que seja útil para um designer de aplicativos. O problema é que o propósito de construir software não é construir uma forma, topologia ou componente específico tipo, então organizar a classificação dessa forma não ajuda um designer a encontrar um estilo que corresponda às suas necessidades. Também mistura as diferenças essenciais entre estilos com outras questões que têm apenas significado incidental, e obscurece a derivação relações entre estilos. Além disso, não se concentra em nenhum tipo específico de arquitetura, como aplicativos baseados em rede. Finalmente, não descreve como os estilos podem ser combinados, nem o efeito de sua combinação.

Buschmann e Meunier [27] descrevem um esquema de classificação que organiza padrões de acordo com a granularidade de abstração, funcionalidade e princípios estruturais. o granularidade de abstração separa padrões em três categorias: arquitetura frameworks (modelos para arquiteturas), padrões de projeto e expressões idiomáticas. Sua classificação aborda algumas das mesmas questões que esta dissertação, como a separação de preocupações e princípios estruturais que levam a propriedades arquitetônicas, mas abrange apenas dois dos estilos arquitetônicos descritos aqui. Sua classificação é consideravelmente ampliada em Buschmann et al. [28] com uma discussão mais extensa de padrões de arquitetura e suas relação à arquitetura de software.

Zimmer [137] organiza padrões de projeto usando um gráfico baseado em seus relacionamentos, tornando mais fácil entender a estrutura geral dos padrões no Gamma et al. [51] Catálogo. No entanto, os padrões classificados não são padrões de arquitetura, e o classificação é baseada exclusivamente na derivação ou relações de uso, em vez de propriedades arquitetônicas.

### **3.8.2 Sistemas Distribuídos e Paradigmas de Programação**

Andrews [6] pesquisa como os processos em um programa distribuído interagem por meio da passagem de mensagens. Ele define programas concorrentes, programas distribuídos, tipos de processos em um programa (filtros, clientes, servidores, peers), paradigmas de interação e comunicação canais. Paradigmas de interação representam os aspectos de comunicação do software estilos arquitetônicos. Ele descreve paradigmas para fluxo de dados unidirecional através de redes de filtros (pipe-and-filter), cliente-servidor, pulsação, sonda/eco, transmissão, passagem de token, servidores replicados e trabalhadores replicados com um pacote de tarefas. No entanto, a apresentação é da perspectiva de vários processos cooperando em uma única tarefa, em vez de estilos arquitetônicos gerais baseados em rede.

Sullivan e Notkin [126] fornecem um levantamento da pesquisa de invocação implícita e descrever sua aplicação para melhorar a qualidade da evolução de conjuntos de ferramentas de software. Barrett et ai. [8] apresentam um levantamento dos mecanismos de integração baseados em eventos através da construção de um framework para comparação e então ver como alguns sistemas se encaixam nessa estrutura. Rosenblum e Wolf [114] investigam uma estrutura de design para notificação de eventos em escala da Internet. Tudo estão preocupados com o escopo e os requisitos de um estilo EBI, em vez de fornecer soluções para sistemas baseados em rede.

Fuggetta et al. [50] fornecem um exame completo e classificação do código móvel paradigmas. Este capítulo se baseia no trabalho deles na medida em que comparo os estilos de código com outros estilos com capacidade de rede e coloque-os em uma única estrutura e conjunto de definições arquitetônicas.

### 3.8.3 Middleware

Bernstein [22] define middleware como um serviço de sistema distribuído que inclui interfaces e protocolos de programação. Esses serviços são chamados de middleware porque atuam como uma camada acima do SO e do software de rede e abaixo formulários. Umar [131] apresenta um tratamento extensivo do assunto.

A pesquisa de arquitetura sobre middleware foca nos problemas e efeitos de integrando componentes com middleware de prateleira. Di Nitto e Rosenblum [38] descrever como o uso de middleware e componentes predefinidos podem influenciar o arquitetura de um sistema que está sendo desenvolvido e, inversamente, como a arquitetura específica escolhas podem restringir a seleção de middleware. Dashofy et al. [35] discutem o uso de middleware com o estilo C2.

Garlan et al. [56] apontam algumas das suposições arquitetônicas dentro da prateleira componentes, examinando os problemas dos autores com a reutilização de subsistemas na criação do Ferramenta Aesop para projeto arquitetônico [54]. Eles classificam os problemas em quatro categorias de suposições que podem contribuir para a incompatibilidade arquitetônica: natureza do componentes, natureza dos conectores, estrutura arquitetônica global e processo de construção.

## 3.9 Resumo

Este capítulo apresentou um levantamento de estilos arquiteturais comuns para sistemas baseados em rede. software aplicativo dentro de uma estrutura de classificação que avalia cada estilo de acordo com às propriedades arquitetônicas que induziria se aplicado a uma arquitetura para um protótipo de sistema hipermídia baseado em rede. A classificação geral é resumida abaixo na Tabela 3-6.

O próximo capítulo usa o insight obtido a partir desta pesquisa e classificação para hipótese de métodos para desenvolver e avaliar um estilo arquitetônico para guiar o projeto de melhorias para a arquitetura moderna da World Wide Web.

Estilo	Derivação				
PF	$\pm \text{+++} \text{++}$				
UPF	PF	$\ddot{\text{y}} \pm$		$\text{++} \text{++} \text{+}$	$\text{++} \text{++} \text{+}$
RR		$\text{++}$	$\text{+}$		$\text{+}$
\$	RR $\text{++++}$				
CS			$\text{+++}$		
LS	$\ddot{\text{y}} \text{++} \text{++}$				
LCS	CS+LS		$\text{++} \text{++} \text{+}$	$\text{+}$	$\text{+}$
CSS	CS		$\text{++} \text{++} \text{+}$	$\text{+}$	$\text{+}$
C\$SS	CSS+\$	$\ddot{\text{y}} \text{++} \text{++} \text{++} \text{+}$		$\text{+}$	$\text{+}$
LC\$SS LCS+C\$SS $\ddot{\text{y}} \pm \text{+} \text{++} \text{++} \text{++}$					$\text{++} \text{++} \text{+}$
RS	CS	$\text{+}\ddot{\text{y}} \text{++}$			
RDA	CS	$\text{+}\ddot{\text{y}}\text{y}$		$\text{+}$	
VM			$\pm$	$\text{+}$	$\ddot{\text{y}}\text{+}$
REV	CS+VM	$\text{+}\ddot{\text{y}} \pm \text{+} \text{+}\ddot{\text{y}} \text{+}\ddot{\text{y}}$			
BACALHAU	CS + VM	$\text{++} \text{++} \text{+}$		$\text{+}$	
LCODC\$SS LC\$SS+COD $\ddot{\text{y}} \text{++} \text{++} \text{+} \text{4+} \text{++} \text{++} \text{++} \text{+}$					$\text{++} \text{+} \text{+}$
MA REV+COD		$\text{+} \text{++}$	$\pm$	$\text{++} \text{++} \text{+}$	$\text{+}\ddot{\text{y}} \text{+}$
EBI		$\text{+}\ddot{\text{y}} \text{y} \pm \text{+} \text{+}$		$\text{+} \text{+}$	
C2	EBI+LCS	$\text{+}$	$\text{+} \text{++} \text{+}$	$\text{+} \text{++} \text{+} \text{+} \text{+}$	
FAZ	CS + CS	$\text{+}$	$\text{++} \text{+}\ddot{\text{y}} \text{y}$		
BDO DO+LCS	$\ddot{\text{y}} \text{y}$		$\text{++} \text{+}$	$\text{+} \text{++} \text{+}\ddot{\text{y}} \text{+}$	

Tabela 3-6. Resumo da avaliação

## CAPÍTULO 4

### Projetando a Arquitetura da Web: Problemas e Insights

Este capítulo apresenta os requisitos da arquitetura da World Wide Web e a problemas enfrentados na concepção e avaliação de melhorias propostas para seus principais protocolos de comunicação. Eu uso os insights obtidos a partir do levantamento e classificação de estilos arquitetônicos para sistemas hipermídia baseados em rede para criar hipóteses de métodos para desenvolver um estilo arquitetônico que seria usado para orientar o projeto de melhorias para a arquitetura Web moderna.

#### 4.1 Requisitos de domínio do aplicativo WWW

Berners-Lee [20] escreve que “o principal objetivo da Web era ser um espaço de informação compartilhado através do qual pessoas e máquinas podem se comunicar”. O que era necessário era uma maneira para que as pessoas armazenem e estruturem suas próprias informações, sejam elas permanentes ou efêmeras na natureza, de modo que possa ser usado por eles mesmos e por outros, e ser capaz de referenciar e estruturar as informações armazenadas por outros para que não seja necessário todos para manter e manter cópias locais.

Os usuários finais pretendidos deste sistema estavam localizados em todo o mundo, em vários laboratórios de pesquisa de física de alta energia da universidade e do governo conectados via Internet. Suas máquinas eram um conjunto heterogêneo de terminais, estações de trabalho, servidores e supercomputadores, exigindo uma miscelânea de software de sistema operacional e formatos de arquivo. As informações variavam de notas pessoais de pesquisa a listas telefônicas organizacionais. o desafio era construir um sistema que fornecesse uma interface universalmente consistente para

esta informação estruturada, disponível em tantas plataformas quanto possível, e incrementalmente implantável à medida que novas pessoas e organizações se juntam ao projeto.

#### **4.1.1 Baixa barreira de entrada**

Como a participação na criação e estruturação da informação era voluntária, um baixo era necessário uma barreira de entrada para permitir uma adoção suficiente. Isso se aplica a todos os usuários do Arquitetura da Web: leitores, autores e desenvolvedores de aplicativos.

A hipermídia foi escolhida como interface do usuário por causa de sua simplicidade e generalidade: a mesma interface pode ser usada independentemente da fonte de informação, a flexibilidade de relacionamentos hipermídia (links) permitem uma estruturação ilimitada, e a A manipulação de links permite que os relacionamentos complexos dentro da informação guiem o leitor através de um aplicativo. Como as informações em grandes bancos de dados geralmente são muito mais fácil de acessar através de uma interface de busca ao invés de navegar, a Web também incorporou a capacidade de realizar consultas simples, fornecendo dados inseridos pelo usuário para um serviço e renderizando o resultado como hipermídia.

Para os autores, o principal requisito era que a disponibilidade parcial do sistema geral não deve impedir a autoria do conteúdo. A linguagem de autoria de hipertexto precisava ser simples e capaz de ser criado usando ferramentas de edição existentes. Esperava-se que os autores manter coisas como notas pessoais de pesquisa neste formato, sejam diretamente ligadas a Internet ou não, então o fato de algumas informações referenciadas não estarem disponíveis, seja temporária ou permanente, não poderia impedir a leitura e autoria de informação que estava disponível. Por razões semelhantes, era necessário poder criar referências a informações antes que o alvo dessa referência estivesse disponível. Uma vez que os autores

foram incentivados a colaborar no desenvolvimento de fontes de informação, referências

precisava ser fácil de se comunicar, seja na forma de instruções por e-mail ou escritas em

a parte de trás de um guardanapo em uma conferência.

A simplicidade também era um objetivo para os desenvolvedores de aplicativos. Uma vez que todos os protocolos foram definidos como texto, a comunicação pode ser visualizada e testada interativamente usando ferramentas de rede existentes. Isso permitiu que a adoção antecipada dos protocolos ocorresse em apesar da falta de padrões.

#### **4.1.2 Extensibilidade**

Embora a simplicidade possibilite implantar uma implementação inicial de um

sistema, a extensibilidade nos permite evitar ficar preso para sempre com as limitações do que

foi implantado. Mesmo que fosse possível construir um sistema de software que combinasse perfeitamente

os requisitos de seus usuários, esses requisitos mudarão ao longo do tempo, assim como a sociedade

Mudanças ao longo do tempo. Um sistema que pretende ser tão duradouro quanto a Web deve ser preparado

para mudar.

#### **4.1.3 Hipermídia Distribuída**

A hipermídia é definida pela presença de informações de controle de aplicativos incorporadas

dentro, ou como uma camada acima, da apresentação da informação. A hipermídia distribuída permite

as informações de apresentação e controle a serem armazenadas em locais remotos. Por sua natureza,

ações do usuário dentro de um sistema de hipermídia distribuído requerem a transferência de grandes quantidades

de dados de onde os dados são armazenados para onde são usados. Assim, a arquitetura Web deve

ser projetado para transferência de dados de grande granularidade.

A usabilidade da interação hipermídia é altamente sensível à latência percebida pelo usuário: o tempo entre a seleção de um link e a renderização de um resultado utilizável. Desde a Web fontes de informação são distribuídas pela Internet global, a arquitetura precisa minimizar as interações de rede (ida e volta dentro dos protocolos de transferência de dados).

#### **4.1.4 Escala da Internet**

A Web pretende ser um sistema de hipermídia distribuído *em escala da Internet*, o que significa consideravelmente mais do que apenas dispersão geográfica. A Internet é sobre interconexão redes de informação através de múltiplas fronteiras organizacionais. Fornecedores de informações serviços devem ser capazes de lidar com as demandas de escalabilidade anárquica e a implantação independente de componentes de software.

##### **4.1.4.1 Escalabilidade Anárquica**

A maioria dos sistemas de software é criada com a suposição implícita de que todo o sistema é sob o controle de uma entidade, ou pelo menos que todas as entidades participantes de um sistema sejam agindo em direção a um objetivo comum e não em objetivos opostos. Tal suposição não pode ser feito com segurança quando o sistema é executado abertamente na Internet. A escalabilidade anárquica refere-se à necessidade de elementos arquitetônicos continuarem operando quando submetidos a um carga imprevista, ou quando dados malformados ou construídos de forma maliciosa, uma vez que eles pode estar se comunicando com elementos fora de seu controle organizacional. O a arquitetura deve ser passível de mecanismos que melhorem a visibilidade e a escalabilidade.

O requisito de escalabilidade anárquica se aplica a todos os elementos arquitetônicos. Clientes não se pode esperar que mantenha o conhecimento de todos os servidores. Não se pode esperar que os servidores reter o conhecimento do estado entre as solicitações. Elementos de dados hipermídia não podem reter “back-

ponteiros”, um identificador para cada elemento de dados que os referencia, uma vez que o número de referências a um recurso é proporcional ao número de pessoas interessadas naquele em formação. Informações particularmente interessantes também podem levar a “flash crowds”: picos nas tentativas de acesso à medida que as notícias de sua disponibilidade se espalham pelo mundo.

A segurança dos elementos arquitetônicos e das plataformas em que operam, também torna-se uma preocupação significativa. Múltiplas fronteiras organizacionais implicam que múltiplas limites de confiança podem estar presentes em qualquer comunicação. Aplicações intermediárias, como como firewalls, devem ser capazes de inspecionar as interações do aplicativo e evitar que fora da política de segurança da organização. Os participantes em uma interação de aplicativo deve assumir que qualquer informação recebida não é confiável, ou exigir alguma autenticação adicional antes que a confiança possa ser concedida. Isso requer que o arquitetura seja capaz de comunicar dados de autenticação e controles de autorização.

No entanto, como a autenticação degrada a escalabilidade, a operação padrão da arquitetura deve ser limitado a ações que não precisam de dados confiáveis: um conjunto seguro de operações com semântica bem definida.

#### *4.1.4.2 Implantação Independente*

Múltiplas fronteiras organizacionais também significam que o sistema deve estar preparado para mudança gradual e fragmentada, onde velhas e novas implementações coexistem sem impedindo que as novas implementações façam uso de seus recursos estendidos. Os elementos arquitetônicos existentes precisam ser projetados com a expectativa de que mais tarde recursos arquitetônicos serão adicionados. Da mesma forma, implementações mais antigas precisam ser facilmente identificados para que o comportamento legado possa ser encapsulado sem afetar negativamente os novos elementos arquitetônicos. A arquitetura como um todo deve ser projetada para facilitar a

implantação de elementos arquitetônicos de forma parcial e iterativa, uma vez que não é possível forçar a implantação de forma ordenada.

## 4.2 Problema

No final de 1993, ficou claro que mais do que apenas pesquisadores estariam interessados na Rede. A adoção ocorreu primeiro em pequenos grupos de pesquisa, espalhados para dormitórios no campus, clubes e home pages pessoais e, posteriormente, para os departamentos institucionais do campus em formação. Quando os indivíduos começaram a publicar suas coleções pessoais de informações, sobre quaisquer tópicos sobre os quais eles possam se sentir fanáticos, o efeito de rede social lançou um crescimento exponencial de sites que continua até hoje. O interesse comercial na Web foi apenas começando, mas já estava claro que a capacidade de publicar em escala internacional seria irresistível para as empresas.

Embora exultante com seu sucesso, a comunidade de desenvolvedores da Internet ficou preocupada que o rápido crescimento no uso da Web, juntamente com algumas características de rede ruins de HTTP inicial, ultrapassaria rapidamente a capacidade da infraestrutura da Internet e levaria a um colapso geral. Isso foi agravado pela natureza mutável das interações de aplicativos em a teia. Enquanto os protocolos iniciais foram projetados para pares únicos de solicitação-resposta, novos sites usaram um número crescente de imagens em linha como parte do conteúdo das páginas da Web, resultando em um perfil de interação diferente para navegação. A arquitetura implantada tinha limitações significativas em seu suporte para extensibilidade, cache compartilhado e intermediários, o que dificultou o desenvolvimento de soluções ad hoc para os problemas crescentes. No mesmo tempo, a competição comercial no mercado de software levou a um influxo de novos e ocasionalmente propostas de recursos contraditórios para os protocolos da Web.

Grupos de trabalho dentro da Força-Tarefa de Engenharia da Internet foram formados para trabalhar em os três principais padrões da Web: URI, HTTP e HTML. A carta desses grupos era definir o subconjunto de comunicação arquitetônica existente que era comum e consistentemente implementado na arquitetura da Web inicial, identificar problemas dentro desse arquitetura e, em seguida, especificar um conjunto de padrões para resolver esses problemas. Isso nos apresentou com um desafio: como introduzimos um novo conjunto de funcionalidades em uma arquitetura que é já amplamente implantado, e como podemos garantir que sua introdução não impactar, ou mesmo destruir, as propriedades arquitetônicas que permitiram à Web ter sucesso?

### 4.3 Abordagem

A arquitetura da Web inicial era baseada em princípios sólidos – separação de interesses, simplicidade e generalidade - mas faltava uma descrição arquitetônica e lógica. o projeto foi baseado em um conjunto de notas informais de hipertexto [14], dois primeiros artigos orientados para a comunidade de usuários [12, 13], e discussões arquivadas no desenvolvedor da Web lista de discussão da comunidade ([www-talk@info.cern.ch](mailto:www-talk@info.cern.ch)). Na realidade, porém, a única verdade descrição da arquitetura da Web inicial foi encontrada dentro das implementações de libwww (a biblioteca de protocolos CERN para clientes e servidores), Mosaic (o navegador NCSA cliente) e uma variedade de outras implementações que interoperaram com eles.

Um estilo de arquitetura pode ser usado para definir os princípios por trás da arquitetura da Web de modo que sejam visíveis para os futuros arquitetos. Conforme discutido no Capítulo 1, um estilo é um conjunto de restrições em elementos arquitetônicos que induz o conjunto de propriedades desejadas do

arquitetura. O primeiro passo na minha abordagem, portanto, é identificar as restrições colocadas dentro da arquitetura da Web inicial que são responsáveis por suas propriedades desejáveis.

**Hipótese I:** A lógica de design por trás da arquitetura WWW pode ser descrita por um estilo arquitetural que consiste no conjunto de restrições aplicadas aos elementos dentro da arquitetura Web.

Restrições adicionais podem ser aplicadas a um estilo arquitetônico para estender o conjunto de propriedades induzidas em arquiteturas instanciadas. O próximo passo na minha abordagem é identificar as propriedades desejáveis em um sistema de hipermídia distribuído em escala da Internet, selecione estilos arquitetônicos adicionais que induzem essas propriedades e as combinam com o restrições iniciais da Web para formar um novo estilo de arquitetura híbrido para a Web moderna arquitetura.

**Hipótese II:** Restrições podem ser adicionadas ao estilo de arquitetura WWW para derivar um novo estilo híbrido que reflete melhor as propriedades desejadas de uma arquitetura Web moderna.

Usando o novo estilo arquitetônico como guia, podemos comparar as extensões propostas e modificações na arquitetura da Web em relação às restrições do estilo. Conflitos indicar que a proposta violaria um ou mais dos princípios de design por trás do Rede. Em alguns casos, o conflito pode ser removido exigindo o uso de um indicador sempre que o novo recurso é usado, como geralmente é feito para extensões HTTP que impactar a capacidade de cache padrão de uma resposta. Para conflitos graves, como uma mudança no estilo de interação, a mesma funcionalidade seria substituída por um design mais condizente com o estilo da Web, ou o proponente seria instruído a implementar a funcionalidade como uma arquitetura separada rodando em paralelo com a Web.

**Hipótese III:** As propostas para modificar a arquitetura da Web podem ser comparadas ao estilo de arquitetura WWW atualizado e analisadas quanto a conflitos antes da implantação.

Finalmente, a arquitetura da Web atualizada, conforme definido pelos padrões de protocolo revisados que foram escritos de acordo com as diretrizes do novo estilo arquitetônico, é implantado através da participação no desenvolvimento da infraestrutura e software de middleware que compõem a maioria dos aplicativos da Web. Isso incluiu minha participação direta em desenvolvimento de software para o projeto do servidor Apache HTTP e o cliente libwww-perl biblioteca, bem como a participação indireta em outros projetos, assessorando os desenvolvedores do projetos W3C libwww e jigsaw, os navegadores Netscape Navigator, Lynx e MSIE, e dezenas de outras implementações, como parte do discurso da IETF.

Embora eu tenha descrito essa abordagem como uma sequência única, ela é realmente aplicada em um moda não sequencial e iterativa. Ou seja, nos últimos seis anos venho construindo modelos, adicionando restrições ao estilo de arquitetura e testando seu efeito na Web padrões de protocolo por meio de extensões experimentais para software cliente e servidor. Da mesma maneira, outros sugeriram a adição de recursos à arquitetura que estavam fora do escopo do meu estilo de modelo então atual, mas não em conflito com ele, o que resultou em ir de volta e revisando as restrições arquitetônicas para refletir melhor a arquitetura aprimorada.

O objetivo sempre foi manter um modelo consistente e correto de como pretendo o arquitetura da Web para se comportar, de modo que possa ser usada para guiar os padrões de protocolo que definir o comportamento apropriado, em vez de criar um modelo artificial que seria limitado às restrições originalmente imaginadas quando o trabalho começou.

## 4.4 Resumo

Este capítulo apresentou os requisitos da arquitetura da World Wide Web e a problemas enfrentados na concepção e avaliação de melhorias propostas para seus principais protocolos de comunicação. O desafio é desenvolver um método para projetar melhorias em uma arquitetura de modo que as melhorias possam ser avaliadas antes sua implantação. Minha abordagem é usar um estilo arquitetônico para definir e melhorar o lógica de design por trás da arquitetura da Web, para usar esse estilo como o teste ácido para provar extensões propostas antes de sua implantação e implantar a arquitetura revisada por meio de envolvimento direto nos projetos de desenvolvimento de software que criaram o a infraestrutura.

O próximo capítulo apresenta e elabora a Transferência Representacional do Estado (REST) para sistemas de hipermídia distribuídos, como foi desenvolvido para representam o modelo de como a Web moderna deve funcionar. REST fornece um conjunto de restrições arquitetônicas que, quando aplicadas como um todo, enfatizam a escalabilidade de interações de componentes, generalidade de interfaces, implantação independente de componentes, e componentes intermediários para reduzir a latência de interação, reforçar a segurança e encapsular sistemas legados.

## CAPÍTULO 5

### Transferência Representacional do Estado (REST)

Este capítulo apresenta e elabora a Transferência de Estado Representacional (REST) estilo arquitetural para sistemas hipermídia distribuídos, descrevendo a engenharia de software princípios que orientam o REST e as restrições de interação escolhidas para reter esses princípios, enquanto os contrasta com as restrições de outros estilos arquitetônicos. REST é um híbrido estilo derivado de vários estilos de arquitetura baseados em rede descritos no Capítulo 3 e combinado com restrições adicionais que definem uma interface de conector uniforme. A estrutura de arquitetura de software do Capítulo 1 é usada para definir os elementos de arquitetura de REST e examinar o processo de amostra, conector e visualizações de dados de protótipos arquiteturas.

#### 5.1 Derivando REST

A lógica de design por trás da arquitetura da Web pode ser descrita por uma estilo que consiste no conjunto de restrições aplicadas aos elementos dentro da arquitetura. Por examinando o impacto de cada restrição à medida que é adicionada ao estilo em evolução, podemos identificar as propriedades induzidas pelas restrições da Web. Restrições adicionais podem então ser aplicado para formar um novo estilo arquitetônico que reflita melhor as propriedades desejadas de um arquitetura web moderna. Esta seção fornece uma visão geral do REST andando através do processo de derivá-lo como um estilo arquitetônico. As seções posteriores descreverão em mais detalhadamente as restrições específicas que compõem o estilo REST.

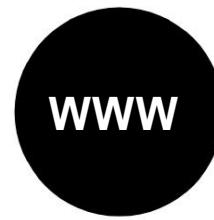


Figura 5-1. Estilo nulo

### **5.1.1 Começando com o Estilo Nulo**

Existem duas perspectivas comuns sobre o processo de projeto arquitetônico, seja para edifícios ou para software. A primeira é que um designer começa com nada - uma lousa em branco, quadro branco ou prancheta - e cria uma arquitetura a partir de componentes familiares até que satisfaça as necessidades do sistema pretendido. A segunda é que um designer começa com as necessidades do sistema como um todo, sem restrições, e então, de forma incremental, identifica e aplica restrições aos elementos do sistema para diferenciar o espaço de projeto e permitir que as forças que influenciam o comportamento do sistema fluam naturalmente, em harmonia com o sistema. Onde o primeiro enfatiza a criatividade e a visão ilimitada, o segundo enfatiza a contenção e a compreensão do contexto do sistema. REST foi desenvolvido usando este último processo. As Figuras 5-1 a 5-8 descrevem isso graficamente em termos de como o restrições aplicadas diferenciariam a visão de processo de uma arquitetura como a conjunto incremental de restrições é aplicado.

O estilo Null (Figura 5-1) é simplesmente um conjunto vazio de restrições. De uma perspectiva arquitetônica, o estilo nulo descreve um sistema no qual não há limites entre os componentes. É o ponto de partida para nossa descrição de REST.

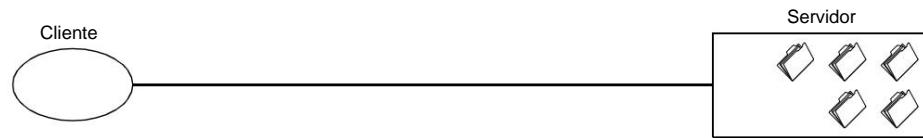


Figura 5-2. Servidor cliente

### 5.1.2 Cliente-Servidor

As primeiras restrições adicionadas ao nosso estilo híbrido são as da arquitetura cliente-servidor estilo (Figura 5-2), descrito na Seção 3.4.1. Separação de interesses é o princípio por trás das restrições cliente-servidor. Ao separar as preocupações da interface do usuário das preocupações com armazenamento de dados, melhoramos a portabilidade da interface do usuário em vários plataformas e melhorar a escalabilidade simplificando os componentes do servidor. Talvez a maioria significativo para a Web, no entanto, é que a separação permite que os componentes evoluam independentemente, apoiando assim o requisito de escala da Internet de vários domínios.

### 5.1.3 Apátrida

Em seguida, adicionamos uma restrição à interação cliente-servidor: a comunicação deve ser sem estado por natureza, como no estilo client-stateless-server (CSS) da Seção 3.4.3 (Figura 5-3), tal que cada solicitação do cliente ao servidor deve conter todas as informações necessárias para

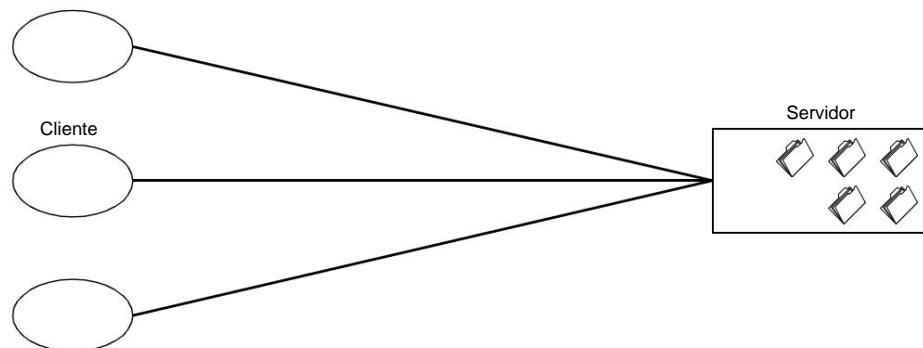


Figura 5-3. Cliente-Stateless-Servidor

entender a solicitação e não pode tirar proveito de nenhum contexto armazenado no servidor.

O estado da sessão é, portanto, mantido inteiramente no cliente.

Essa restrição induz as propriedades de visibilidade, confiabilidade e escalabilidade. Visibilidade é melhorado porque um sistema de monitoramento não precisa ir além de uma única solicitação datum para determinar a natureza completa do pedido. A confiabilidade é melhorada porque facilita a tarefa de recuperação de falhas parciais [133]. A escalabilidade é melhorada porque não ter que armazenar o estado entre as solicitações permite que o componente do servidor libere rapidamente recursos e simplifica ainda mais a implementação porque o servidor não precisa gerenciar o uso de recursos entre solicitações.

Como a maioria das escolhas arquitetônicas, a restrição stateless reflete uma troca de design. O desvantagem é que pode diminuir o desempenho da rede aumentando os dados repetitivos (sobrecarga por interação) enviado em uma série de solicitações, pois esses dados não podem ser deixados no servidor em um contexto compartilhado. Além disso, colocar o estado do aplicativo no lado do cliente reduz o controle do servidor sobre o comportamento consistente do aplicativo, pois o aplicativo torna-se dependente da implementação correta da semântica em vários clientes versões.

#### **5.1.4 Cache**

Para melhorar a eficiência da rede, adicionamos restrições de cache para formar o cache do cliente estilo de servidor sem estado da Seção 3.4.4 (Figura 5-4). As restrições de cache exigem que os dados dentro de uma resposta a uma solicitação ser rotulada implícita ou explicitamente como armazenável em cache ou não armazenável em cache. Se uma resposta pode ser armazenada em cache, um cache de cliente recebe o direito de reutilizar essa dados de resposta para solicitações equivalentes posteriores.

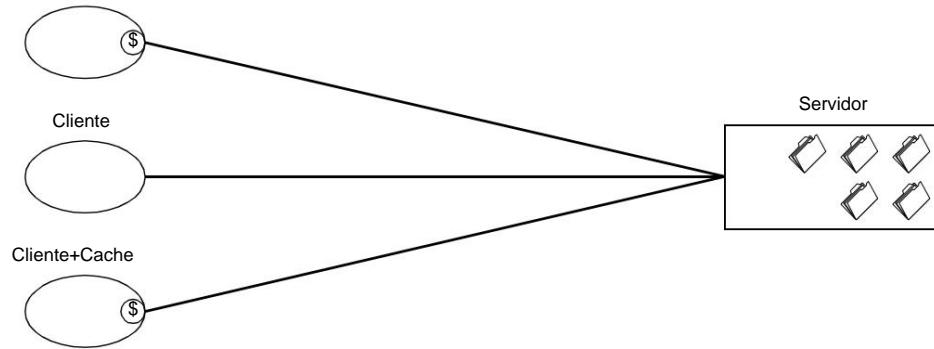
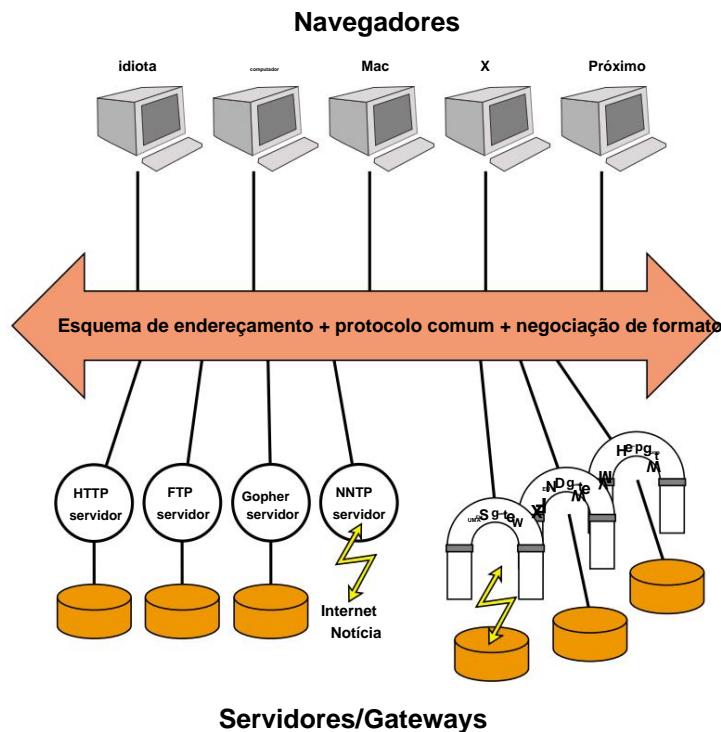


Figura 5-4. Client-Cache-Stateless-Server

A vantagem de adicionar restrições de cache é que elas têm o potencial de eliminar completamente algumas interações, melhorando a eficiência, escalabilidade e desempenho percebido reduzindo a latência média de uma série de interações. O desvantagem, no entanto, é que um cache pode diminuir a confiabilidade se dados obsoletos dentro do cache difere significativamente dos dados que teriam sido obtidos se a solicitação tivesse sido enviada diretamente ao servidor.

A arquitetura da Web inicial, conforme retratado pelo diagrama na Figura 5-5 [11], foi definido pelo conjunto de restrições client-cache-stateless-server. Ou seja, a lógica do projeto apresentado para a arquitetura da Web antes de 1994 focado em stateless client-server interação para a troca de documentos estáticos pela Internet. Os protocolos de interações de comunicação tinham suporte rudimentar para caches não compartilhados, mas não restringir a interface a um conjunto consistente de semântica para todos os recursos. Em vez disso, a Web baseou-se no uso de uma biblioteca de implementação cliente-servidor comum (CERN libwww) para manter a consistência entre os aplicativos da Web.

Os desenvolvedores de implementações da Web já haviam ultrapassado o design inicial. Dentro Além de documentos estáticos, as solicitações podem identificar serviços que geram dinamicamente respostas, como mapas de imagem [Kevin Hughes] e scripts do lado do servidor [Rob McCool].



© 1992 Tim Berners-Lee, Robert Cailliau, Jean-François Groff, CERN

Figura 5-5. Diagrama de arquitetura inicial da WWW

O trabalho também havia começado em componentes intermediários, na forma de proxies [79] e caches [59], mas foram necessárias extensões aos protocolos para que eles se comunicassem de forma confiável. As seções a seguir descrevem as restrições adicionadas à arquitetura da Web estilo para orientar as extensões que formam a arquitetura Web moderna.

### 5.1.5 Interface Uniforme

A característica central que distingue o estilo de arquitetura REST de outras redes estilos baseados é sua ênfase em uma interface uniforme entre os componentes (Figura 5-6). Por aplicando o princípio de engenharia de software de generalidade à interface do componente, o a arquitetura geral do sistema é simplificada e a visibilidade das interações é aprimorada. As implementações são dissociadas dos serviços que prestam, o que incentiva a

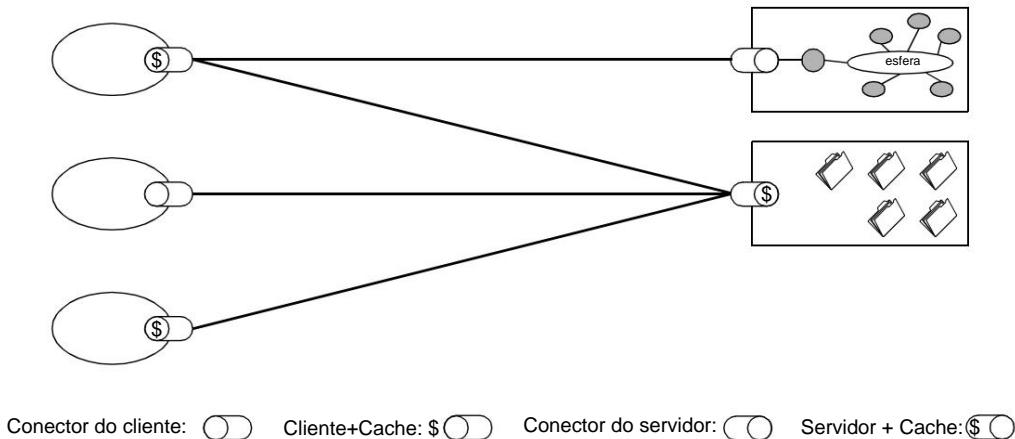


Figura 5-6. Uniform-Client-Cache-Stateless-Server

evolutividade independente. A desvantagem, porém, é que uma interface uniforme degrada eficiência, uma vez que a informação é transferida de forma padronizada e não de forma específico para as necessidades de um aplicativo. A interface REST foi projetada para ser eficiente para grandes transferências de dados hipermídia granular, otimizando para o caso comum da Web, mas resultando em uma interface que não é ideal para outras formas de interação arquitetônica.

A fim de obter uma interface uniforme, várias restrições arquiteturais são necessárias para orientar o comportamento dos componentes. REST é definido por quatro restrições de interface: identificação de recursos; manipulação de recursos por meio de representações; auto mensagens descritivas; e, hipermídia como motor do estado da aplicação. Essas restrições serão discutidas na Seção 5.2.

### 5.1.6 Sistema em camadas

Para melhorar ainda mais o comportamento dos requisitos de escala da Internet, adicionamos restrições do sistema (Figura 5-7). Conforme descrito na Seção 3.4.2, o estilo do sistema em camadas permite que uma arquitetura seja composta de camadas hierárquicas restringindo componentes comportamento tal que cada componente não pode “ver” além da camada imediata com a qual

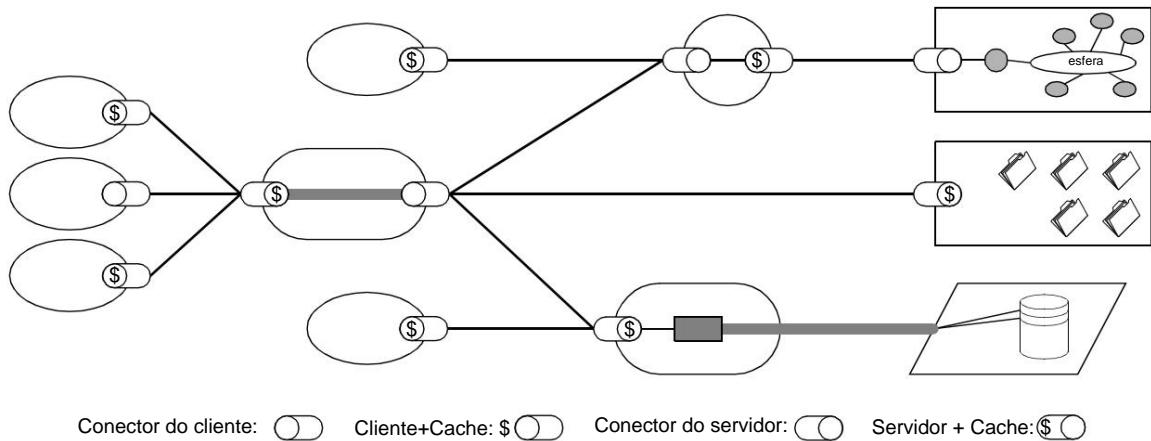


Figura 5-7. Uniform-Layered-Client-Cache-Stateless-Server

eles estão interagindo. Ao restringir o conhecimento do sistema a uma única camada, colocamos um vinculado à complexidade geral do sistema e promover a independência do substrato. As camadas podem ser usado para encapsular serviços legados e proteger novos serviços de clientes legados, simplificando componentes movendo funcionalidades raramente usadas para um intermediário. Os intermediários também podem ser usados para melhorar a escalabilidade do sistema, permitindo balanceamento de carga de serviços em várias redes e processadores.

A principal desvantagem dos sistemas em camadas é que eles adicionam sobrecarga e latência aos o processamento de dados, reduzindo o desempenho percebido pelo usuário [32]. Para uma rede sistema que suporta restrições de cache, isso pode ser compensado pelos benefícios do cache compartilhado em intermediários. Colocar caches compartilhados nos limites de um domínio organizacional pode resultar em benefícios de desempenho significativos [136]. Essas camadas também permitem que as políticas de segurança ser aplicado em dados que cruzam a fronteira organizacional, como é exigido por firewalls [79].

A combinação de sistema em camadas e restrições de interface uniforme induz propriedades arquitetônicas semelhantes às do estilo tubo e filtro uniforme (Seção 3.2.2). Embora a interação REST seja bidirecional, os fluxos de dados de grande granularidade de

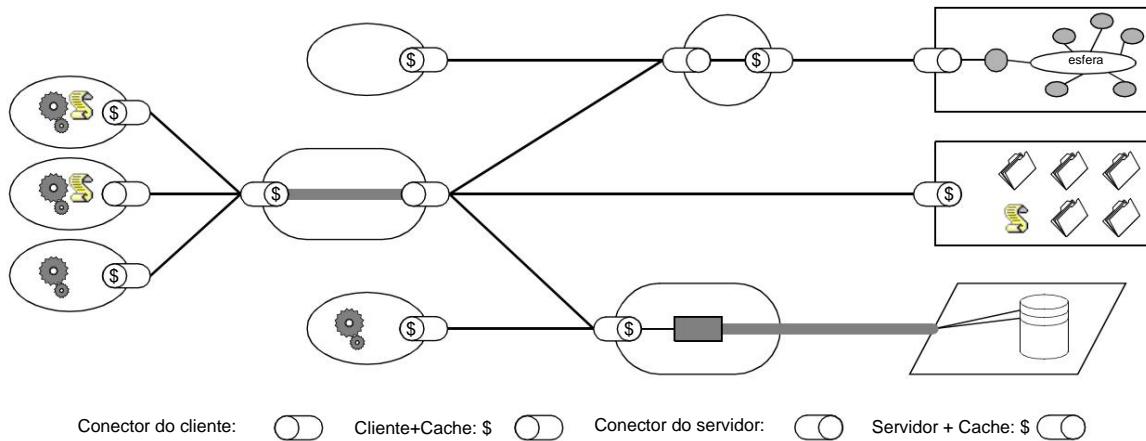


Figura 5-8. DESCANSO

interação hipermídia pode ser processada como uma rede de fluxo de dados, com filtro componentes aplicados seletivamente ao fluxo de dados para transformar o conteúdo conforme ele passa [26]. Dentro do REST, componentes intermediários podem transformar ativamente o conteúdo de mensagens porque as mensagens são autodescritivas e sua semântica é visível para intermediários.

### 5.1.7 Código sob demanda

A adição final ao nosso conjunto de restrições para REST vem do estilo de código sob demanda de Seção 3.5.3 (Figura 5-8). REST permite que a funcionalidade do cliente seja estendida por baixar e executar código na forma de applets ou scripts. Isso simplifica os clientes reduzindo o número de recursos necessários para serem pré-implementados. Permitindo que os recursos ser baixado após a implantação melhora a extensibilidade do sistema. No entanto, também reduz visibilidade e, portanto, é apenas uma restrição opcional no REST.

A noção de uma restrição opcional pode parecer um oxímoro. No entanto, faz têm um propósito no projeto de arquitetura de um sistema que engloba múltiplas limites organizacionais. Isso significa que a arquitetura só ganha o benefício (e sofre

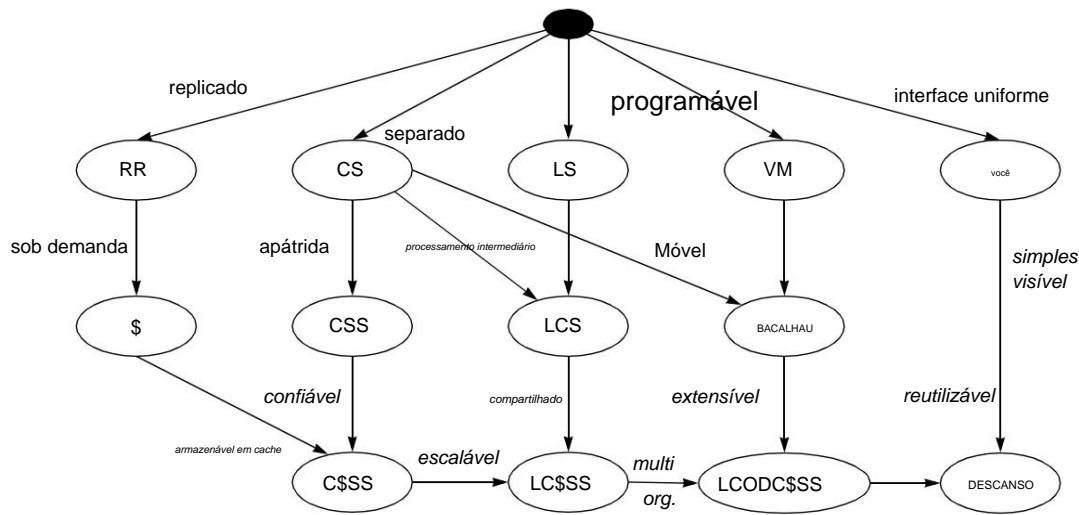


Figura 5-9. Derivação REST por restrições de estilo

as desvantagens) das restrições opcionais quando se sabe que estão em vigor para alguns domínio do sistema geral. Por exemplo, se todo o software cliente dentro de um organização é conhecida por suportar applets Java [45], então os serviços dentro dessa organização podem ser construídos de tal forma que obtenham o benefício de funcionalidade aprimorada por meio de classes Java para download. Ao mesmo tempo, porém, o firewall da organização pode impedir a transferência de applets Java de fontes externas e, portanto, para o resto da Web aparecerá como se esses clientes não suportassem código sob demanda. Uma restrição opcional nos permite projetar uma arquitetura que suporte o comportamento desejado no caso geral, mas com o entendimento de que pode ser desabilitado em alguns contextos.

### 5.1.8 Resumo de Derivação de Estilo

REST consiste em um conjunto de restrições arquitetônicas escolhidas para as propriedades que elas induzem em arquiteturas candidatas. Embora cada uma dessas restrições possa ser considerada isoladamente, descrevê-los em termos de sua derivação de estilos arquitetônicos comuns torna

mais fácil de entender o raciocínio por trás de sua seleção. A Figura 5-9 mostra a derivação das restrições do REST graficamente em termos dos estilos de arquitetura baseados em rede examinado no Capítulo 3.

## 5.2 Elementos Arquitetônicos REST

O estilo Representational State Transfer (REST) é uma abstração da arquitetura elementos dentro de um sistema de hipermídia distribuído. REST ignora os detalhes do componente implementação e sintaxe de protocolo para focar nos papéis dos componentes, o restrições sobre sua interação com outros componentes, e sua interpretação de elementos de dados significativos. Ela engloba as restrições fundamentais sobre os componentes, conectores e dados que definem a base da arquitetura da Web e, portanto, a essência do seu comportamento como um aplicativo baseado em rede.

### 5.2.1 Elementos de Dados

Ao contrário do estilo de objeto distribuído [31], onde todos os dados são encapsulados e ocultos pelos componentes de processamento, a natureza e o estado dos elementos de dados de uma arquitetura é um aspecto chave do REST. A razão para este projeto pode ser vista na natureza da distribuição hipermídia. Quando um link é selecionado, as informações precisam ser movidas do local onde é armazenado até o local onde será usado, na maioria dos casos, por um leitor humano. Isso é diferente de muitos outros paradigmas de processamento distribuído [6, 50], onde é possível, e geralmente mais eficiente, para mover o “agente de processamento” (por exemplo, código móvel, procedimento, expressão de pesquisa, etc.) para os dados em vez de mover os dados para o processador.

Um arquiteto de hipermídia distribuído tem apenas três opções fundamentais: 1) renderizar o dados onde está localizado e enviar uma imagem de formato fixo para o destinatário; 2) encapsular o

dados com um mecanismo de renderização e enviar ambos para o destinatário; ou, 3) enviar os dados brutos para o destinatário junto com metadados que descrevem o tipo de dados, para que o destinatário possa escolher seu próprio mecanismo de renderização.

Cada opção tem suas vantagens e desvantagens. Opção 1, o cliente tradicional estilo servidor [31], permite que todas as informações sobre a verdadeira natureza dos dados permaneçam ocultas dentro do remetente, evitando que sejam feitas suposições sobre a estrutura de dados e facilitando a implementação do cliente. No entanto, também restringe severamente a funcionalidade do destinatário e coloca a maior parte da carga de processamento no remetente, levando à escalabilidade problemas. A opção 2, o estilo de objeto móvel [50], fornece ocultação de informações enquanto permitindo o processamento especializado dos dados por meio de seu mecanismo de renderização exclusivo, mas limita a funcionalidade do destinatário para o que é antecipado dentro desse mecanismo e pode amplamente aumentar a quantidade de dados transferidos. A opção 3 permite que o remetente permaneça simples e escalável, minimizando os bytes transferidos, mas perde as vantagens da informação oculta e requer que o remetente e o destinatário compreendam os mesmos tipos de dados.

REST fornece um híbrido de todas as três opções, concentrando-se em uma compreensão compartilhada de tipos de dados com metadados, mas limitando o escopo do que é revelado a um interface. Os componentes REST se comunicam transferindo uma representação de um recurso em um formato que corresponde a um conjunto em evolução de tipos de dados padrão, selecionados dinamicamente com base nas capacidades ou desejos do destinatário e na natureza do recurso. Se a representação está no mesmo formato que a fonte bruta, ou é derivada da fonte, permanece escondido atrás da interface. Os benefícios do estilo de objeto móvel são aproximado enviando uma representação que consiste em instruções nos dados padrão formato de um mecanismo de renderização encapsulado (por exemplo, Java [45]). REST, portanto, ganha o

separação das preocupações do estilo cliente-servidor sem o problema de escalabilidade do servidor, permite ocultar informações através de uma interface genérica para permitir o encapsulamento e evolução dos serviços, e fornece um conjunto diversificado de funcionalidades por meio de download motores de recursos.

Os elementos de dados do REST estão resumidos na Tabela 5-1.

Tabela 5-1. Elementos de dados REST

Elemento de dados	Exemplos da Web Moderna
recurso	o alvo conceitual pretendido de uma referência de hipertexto
identificador de recurso	URL, URL
representação	Documento HTML, imagem JPEG
tipo de mídia de metadados de representação, hora da última modificação	
metadados de recursos	link de origem, alternativas, variar
dados de controle	if-modified-desde, controle de cache

### 5.2.1.1 Recursos e Identificadores de Recursos

A abstração chave de informações em REST é um *recurso*. Qualquer informação que possa ser nomeado pode ser um recurso: um documento ou imagem, um serviço temporal (por exemplo, “tempo de hoje em Los Angeles”), uma coleção de outros recursos, um objeto não virtual (por exemplo, uma pessoa), e em breve. Em outras palavras, qualquer conceito que possa ser alvo do hipertexto de um autor referência deve caber dentro da definição de um recurso. Um recurso é um mapeamento conceitual a um conjunto de entidades, não a entidade que corresponde ao mapeamento em qualquer ponto particular

Tempo.

Mais precisamente, um recurso  $R$  é uma função de pertinência temporalmente variável  $MR(t)$ , que para o tempo  $t$  mapeia para um conjunto de entidades, ou valores, que são equivalentes. Os valores no set podem ser *representações de recursos* e/ou *identificadores de recursos*. Um recurso pode mapear para o

conjunto vazio, que permite fazer referências a um conceito antes de qualquer realização desse existe um conceito - uma noção que era estranha à maioria dos sistemas de hipertexto antes da Web [61]. Alguns recursos são estáticos no sentido de que, quando examinados a qualquer momento após sua criação, correspondem sempre ao mesmo conjunto de valores. Outros têm um alto grau de variação de seu valor ao longo do tempo. A única coisa que precisa ser estática para um recurso é a semântica do mapeamento, pois a semântica é o que distingue um recurso de outro.

Por exemplo, a "versão preferida dos autores" de um artigo acadêmico é um mapeamento cujo valor muda ao longo do tempo, enquanto um mapeamento para "o artigo publicado no anais da conferência X" é estático. São dois recursos distintos, mesmo que ambos mapear para o mesmo valor em algum momento. A distinção é necessária para que ambos os recursos podem ser identificados e referenciados de forma independente. Um exemplo semelhante de engenharia de software é a identificação separada de um arquivo de código-fonte controlado por versão ao se referir à "última revisão", "revisão número 1.2.7" ou "revisão incluída com o lançamento Orange."

Essa definição abstrata de um recurso habilita os principais recursos da arquitetura da Web. Primeiro, ela fornece generalidade ao abranger muitas fontes de informação sem distinguindo-os artificialmente por tipo ou implementação. Em segundo lugar, permite vinculação tardia da referência a uma representação, permitindo que a negociação de conteúdo ocorra com base em características do pedido. Finalmente, permite que um autor faça referência ao conceito em vez de do que alguma representação singular desse conceito, eliminando assim a necessidade de alterar todos links existentes sempre que a representação mudar (assumindo que o autor usou o direito identificador).

REST usa um identificador de *recurso* para identificar o recurso específico envolvido em um interação entre os componentes. Os conectores REST fornecem uma interface genérica para acessar e manipular o conjunto de valores de um recurso, independentemente de como a associação função é definida ou o tipo de software que está processando a solicitação. A nomenclatura autoridade que atribuiu o identificador do recurso, tornando possível fazer referência ao recurso, é responsável por manter a validade semântica do mapeamento ao longo do tempo (ou seja, garantindo que a função de pertinência não mude).

Sistemas tradicionais de hipertexto [61], que normalmente operam em um ambiente fechado ou local. ambiente, use identificadores exclusivos de nó ou documento que mudam toda vez que o alterações de informações, contando com servidores de link para manter as referências separadamente do conteúdo [135]. Como os servidores de link centralizados são um anátema para a imensa escala e requisitos de domínio multi-organizacional da Web, REST depende, em vez disso, do autor escolher um identificador de recurso que melhor se ajuste à natureza do conceito que está sendo identificado. Naturalmente, a qualidade de um identificador é muitas vezes proporcional à quantidade de dinheiro gasto para manter sua validade, o que leva a links quebrados como efêmeros (ou mal suportados) informações se movem ou desaparecem com o tempo.

### 5.2.1.2 Representações

Os componentes REST executam ações em um recurso usando uma representação para capturar o estado atual ou pretendido desse recurso e transferir essa representação entre componentes. Uma *representação* é uma sequência de bytes, mas *metadados de representação* para descreva esses bytes. Outros nomes comumente usados, mas menos precisos, para uma representação incluem: documento, arquivo e entidade, instância ou variante de mensagem HTTP.

Uma representação consiste em dados, metadados que descrevem os dados e, ocasionalmente, metadados para descrever os metadados (geralmente com a finalidade de verificar a integridade da mensagem).

Os metadados estão na forma de pares nome-valor, onde o nome corresponde a um padrão que define a estrutura e a semântica do valor. As mensagens de resposta podem incluir tanto metadados de representação e metadados de *recursos*: informações sobre o recurso que não são específico para a representação fornecida.

Os *dados de controle* definem o propósito de uma mensagem entre os componentes, como a ação sendo solicitado ou o significado de uma resposta. Também é usado para parametrizar solicitações e substituir o comportamento padrão de alguns elementos de conexão. Por exemplo, comportamento de cache pode ser modificado pelos dados de controle incluídos na mensagem de solicitação ou resposta.

Dependendo dos dados de controle da mensagem, uma determinada representação pode indicar o estado atual do recurso solicitado, o estado desejado para o recurso solicitado ou o valor de algum outro recurso, como uma representação dos dados de entrada dentro de um cliente formulário de consulta ou uma representação de alguma condição de erro para uma resposta. Por exemplo, a autoria remota de um recurso requer que o autor envie uma representação ao servidor, estabelecendo assim um valor para esse recurso que pode ser recuperado por solicitações posteriores. Se o conjunto de valores de um recurso em um determinado momento consiste em múltiplas representações, conteúdo negociação pode ser usada para selecionar a melhor representação para inclusão em uma determinada mensagem.

O formato de dados de uma representação é conhecido como *tipo de mídia* [48]. Uma representação pode ser incluído em uma mensagem e processado pelo destinatário de acordo com os dados de controle da mensagem e a natureza do tipo de mídia. Alguns tipos de mídia destinam-se a processamento automatizado, alguns se destinam a ser renderizados para visualização por um usuário e alguns

são capazes de ambos. Os tipos de mídia composta podem ser usados para incluir vários representações em uma única mensagem.

O design de um tipo de mídia pode afetar diretamente o desempenho percebido pelo usuário de um sistema hipermídia distribuído. Quaisquer dados que devam ser recebidos antes que o destinatário possa começar a renderizar a representação aumenta a latência de uma interação. Um formato de dados que coloca as informações de renderização mais importantes na frente, de modo que as informações iniciais pode ser processado de forma incremental enquanto o restante das informações está sendo recebido, resulta em desempenho percebido pelo usuário muito melhor do que um formato de dados que deve ser totalmente recebido antes que a renderização possa começar.

Por exemplo, um navegador da Web que pode renderizar incrementalmente um grande documento HTML enquanto está sendo recebido fornece um desempenho percebido pelo usuário significativamente melhor do que aquele que espera até que todo o documento seja completamente recebido antes de renderizar, mesmo embora o desempenho da rede seja o mesmo. Observe que a capacidade de renderização de um a representação também pode ser impactada pela escolha do conteúdo. Se as dimensões de tabelas de tamanho dinâmico e objetos incorporados devem ser determinados antes que possam ser renderizada, sua ocorrência dentro da área de visualização de uma página hipermídia aumentará sua latência.

### **5.2.2 Conectores**

REST usa vários tipos de conectores, resumidos na Tabela 5-2, para encapsular as atividades de acessar recursos e transferir representações de recursos. Os conectores presentes uma interface abstrata para comunicação de componentes, aumentando a simplicidade ao fornecer um separação clara de preocupações e ocultando a implementação subjacente de recursos e

Tabela 5-2. Conectores REST

<b>Exemplos da Web Moderna do Conector</b>	
cliente	libwww, libwww-perl
servidor	libwww, Apache API, NSAPI
esconderijo	cache do navegador, rede de cache Akamai
resolver	bind (biblioteca de pesquisa DNS)
túnel	SOCKS, SSL após HTTP CONNECT

mecanismos de comunicação. A generalidade da interface também permite substituibilidade: se o único acesso dos usuários ao sistema for através de uma interface abstrata, a implementação pode ser substituídos sem afetar os usuários. Como um conector gerencia a rede comunicação para um componente, as informações podem ser compartilhadas através de múltiplas interações em para melhorar a eficiência e a capacidade de resposta.

Todas as interações REST são sem estado. Ou seja, cada solicitação contém todos as informações necessárias para que um conector entenda a solicitação, independente de qualquer solicitações que possam tê-lo precedido. Essa restrição cumpre quatro funções: 1) elimina qualquer necessidade de os conectores reterem o estado do aplicativo entre as solicitações, assim reduzir o consumo de recursos físicos e melhorar a escalabilidade; 2) permite interações sejam processadas em paralelo sem exigir que o mecanismo de processamento compreender a semântica da interação; 3) permite que um intermediário visualize e compreenda uma solicitação isolada, que pode ser necessária quando os serviços são reorganizados dinamicamente; e, 4) força todas as informações que podem influenciar na reutilização de um arquivo em cache resposta a estar presente em cada solicitação.

A interface do conector é semelhante à invocação procedural, mas com importantes diferenças na passagem de parâmetros e resultados. Os in-parâmetros consistem em solicitação

dados de controle, um identificador de recurso indicando o destino da solicitação e um opcional representação. Os parâmetros de saída consistem em dados de controle de resposta, recursos opcionais metadados e uma representação opcional. Do ponto de vista abstrato, a invocação é síncrono, mas os parâmetros de entrada e saída podem ser passados como fluxos de dados. Em outras palavras, o processamento pode ser invocado antes que o valor dos parâmetros seja completamente conhecido, evitando assim a latência do processamento em lote de grandes transferências de dados.

Os tipos de conector primário são cliente e servidor. A diferença essencial entre os dois é que um *cliente* inicia a comunicação fazendo um pedido, enquanto um *servidor* ouve conexões e responde a solicitações para fornecer acesso aos seus serviços. UMA componente pode incluir conectores de cliente e servidor.

Um terceiro tipo de conector, o conector de *cache*, pode ser localizado na interface para um cliente ou conector de servidor para salvar respostas em cache às interações atuais para que elas pode ser reutilizado para interações solicitadas posteriormente. Um cache pode ser usado por um cliente para evitar repetição da comunicação de rede, ou por um servidor para evitar a repetição do processo de gerando uma resposta, com ambos os casos servindo para reduzir a latência da interação. Um cache é normalmente implementado no espaço de endereço do conector que o utiliza.

Alguns conectores de cache são compartilhados, o que significa que suas respostas em cache podem ser usadas em responder a um cliente diferente daquele para o qual a resposta foi originalmente obtida. O cache compartilhado pode ser eficaz na redução do impacto de “flash crowds” na carga de um servidor popular, particularmente quando o cache é organizado hierarquicamente para cobrir grandes grupos de usuários, como os da intranet de uma empresa, os clientes de um provedor de serviços, ou universidades que compartilham um backbone de rede nacional. No entanto, compartilhado cache também pode levar a erros se a resposta em cache não corresponder ao que teria

foi obtido por um novo pedido. REST tenta equilibrar o desejo de transparência na

comportamento de cache com o desejo de uso eficiente da rede, em vez de assumir que

é sempre necessária transparência absoluta.

Um cache é capaz de determinar a capacidade de cache de uma resposta porque a interface é genérico em vez de específico para cada recurso. Por padrão, a resposta a uma solicitação de recuperação é armazenável em cache e as respostas a outras solicitações não são armazenáveis em cache. Se alguma forma de usuário autenticação faz parte do pedido, ou se a resposta indicar que não deve ser compartilhado, a resposta só pode ser armazenada em cache por um cache não compartilhado. Um componente pode substituir esses padrões incluindo dados de controle que marcam a interação como armazenável em cache, não armazenável em cache ou armazenável em cache apenas por um tempo limitado.

Um *resolvedor* traduz identificadores de recursos parciais ou completos no endereço de rede informações necessárias para estabelecer uma conexão entre componentes. Por exemplo, a maioria dos URIs incluir um nome de host DNS como o mecanismo para identificar a autoridade de nomenclatura para o recurso. Para iniciar uma solicitação, um navegador da Web extrairá o nome do host do URI e fazer uso de um resolvedor de DNS para obter o endereço de protocolo da Internet para esse autoridade. Outro exemplo é que alguns esquemas de identificação (por exemplo, URN [124]) exigem um intermediário para traduzir um identificador permanente para um endereço mais transitório, a fim de acessar o recurso identificado. O uso de um ou mais resolvedores intermediários pode melhorar a longevidade de referências de recursos por meio de indireção, embora isso aumente a solicitação latência.

A forma final do tipo de conector é um *túnel*, que simplesmente retransmite a comunicação através de um limite de conexão, como um firewall ou gateway de rede de nível inferior. O único razão pela qual é modelado como parte do REST e não abstraído como parte da rede

infraestrutura é que alguns componentes REST podem mudar dinamicamente de ativos comportamento do componente ao de um túnel. O exemplo principal é um proxy HTTP que muda para um túnel em resposta a uma solicitação do método CONNECT [71], permitindo assim a sua cliente para se comunicar diretamente com um servidor remoto usando um protocolo diferente, como TLS, que não permite proxies. O túnel desaparece quando ambas as extremidades terminam sua comunicação.

### 5.2.3 Componentes

Os componentes REST, resumidos na Tabela 5-3, são tipificados por suas funções em um ação de aplicação.

Tabela 5-3. Componentes REST

<b>Exemplos da Web Moderna de Componentes</b>	
	servidor de origem Apache httpd, Microsoft IIS
Porta de entrada	Lula, CGI, Proxy Reverso
procurador	Proxy CERN, Proxy Netscape, Gauntlet
agente de usuário	Netscape Navigator, Lynx, MOMspider

Um *agente de usuário* usa um conector de cliente para iniciar uma solicitação e se torna o melhor destinatário da resposta. O exemplo mais comum é um navegador da Web, que fornece acesso a serviços de informação e fornece respostas de serviço de acordo com a aplicação precisa.

Um *servidor de origem* usa um conector de servidor para controlar o namespace de um recurso. É a fonte definitiva para representações de seus recursos e deve ser o destinatário final de qualquer solicitação que pretenda modificar o valor de seus recursos. Cada

servidor de origem fornece uma interface genérica para seus serviços como uma hierarquia de recursos. os detalhes de implementação de recursos estão ocultos por trás da interface.

Os componentes intermediários atuam como cliente e servidor para encaminhar, com possível tradução, pedidos e respostas. Um componente *proxy* é um intermediário selecionado por um cliente para fornecer encapsulamento de interface de outros serviços, tradução de dados, aprimoramento de desempenho ou proteção de segurança. Um *gateway* (também conhecido como *proxy reverso*) componente é um intermediário imposto pela rede ou servidor de origem para fornecer um encapsulamento de interface de outros serviços, para tradução de dados, aprimoramento de desempenho, ou aplicação de segurança. Observe que a diferença entre um proxy e um gateway é que um cliente determina quando usará um proxy.

### 5.3 Visões arquitetônicas REST

Agora que temos uma compreensão dos elementos arquiteturais REST isoladamente, pode usar visões arquitetônicas [105] para descrever como os elementos trabalham juntos para formar um arquitetura. Três tipos de visualização – processo, conector e dados – são úteis para iluminando os princípios de design do REST.

#### 5.3.1 Visualização do Processo

Uma visão de processo de uma arquitetura é principalmente eficaz em eliciar a interação relacionamentos entre os componentes, revelando o caminho dos dados conforme eles fluem pelo sistema. Infelizmente, a interação de um sistema real geralmente envolve um extenso vários componentes, resultando em uma visão geral obscurecida pelos detalhes.

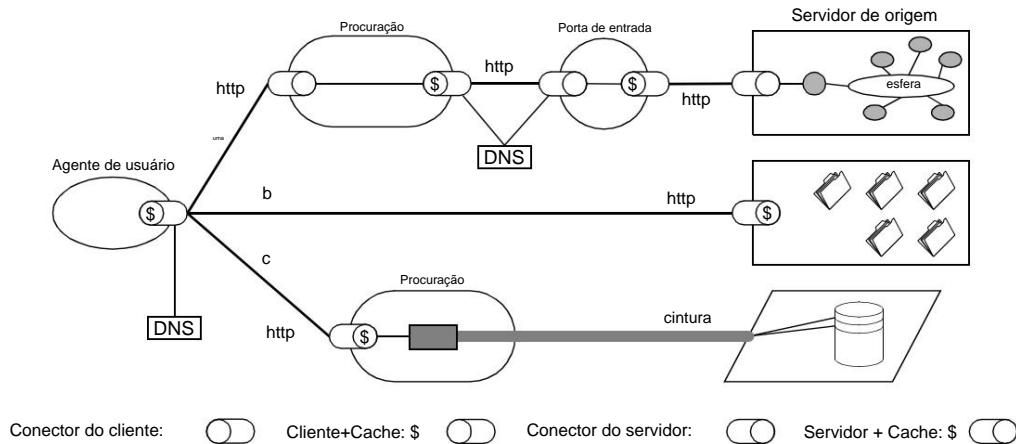


Figura 5-10. Visão de processo de uma arquitetura baseada em REST Um

agente de usuário é retratado no meio de três interações paralelas: a, b e c. As interações não foram atendidas pelo cache do conector do cliente do agente do usuário, portanto, cada solicitação foi roteada para a origem do recurso de acordo com as propriedades de cada identificador de recurso e a configuração do conector do cliente. A solicitação (a) foi enviada para um proxy local, que por sua vez acessa um gateway de cache encontrado pela pesquisa de DNS, que encaminha a solicitação para ser atendida por um servidor de origem cujos recursos internos são definidos por uma arquitetura de agente de solicitação de objeto encapsulado. A solicitação (b) é enviada diretamente para um servidor de origem, que é capaz de atender a solicitação de seu próprio cache. A solicitação (c) é enviada a um proxy capaz de acessar diretamente o WAIS, um serviço de informações separado da arquitetura Web, e traduzir a resposta WAIS em um formato reconhecido pela interface genérica do conector. Cada componente está cliente apenas da interação com seus próprios conectores de cliente ou servidor; a topologia geral do processo é um artefato de nossa visão.

A Figura 5-10 fornece uma amostra da visualização do processo de uma arquitetura baseada em REST em um instância particular durante o processamento de três solicitações paralelas.

A separação de interesses cliente-servidor do REST simplifica a implementação de componentes, reduz a complexidade da semântica do conector, melhora a eficácia do desempenho ajuste e aumenta a escalabilidade de componentes de servidor puro. Sistema em camadas restrições permitem que intermediários - proxies, gateways e firewalls - sejam introduzidos vários pontos na comunicação sem alterar as interfaces entre componentes, permitindo assim que eles auxiliem na tradução da comunicação ou melhorem desempenho por meio de cache compartilhado em grande escala. REST permite o processamento intermediário por restringindo as mensagens a serem autodescritivas: a interação é sem estado entre solicitações,

métodos padrão e tipos de mídia são usados para indicar semântica e trocar informações e as respostas indicam explicitamente a capacidade de armazenamento.

Como os componentes estão conectados dinamicamente, seu arranjo e função para um determinada ação de aplicativo tem características semelhantes a um estilo pipe-and-filter. Embora Os componentes REST se comunicam por meio de fluxos bidirecionais, o processamento de cada direção é independente e, portanto, suscetível a transdutores de fluxo (filtros). o interface de conector genérico permite que componentes sejam colocados no fluxo com base no propriedades de cada solicitação ou resposta.

Os serviços podem ser implementados usando uma hierarquia complexa de intermediários e vários servidores de origem distribuída. A natureza sem estado do REST permite que cada interação ser independente dos outros, eliminando a necessidade de uma consciência do topologia de componentes, uma tarefa impossível para uma arquitetura em escala de Internet, e permitindo componentes para atuar como destinos ou intermediários, determinados dinamicamente pelo alvo de cada solicitação. Os conectores precisam apenas estar cientes da existência um do outro durante o escopo de sua comunicação, embora possam armazenar em cache a existência e as capacidades de outros componentes por motivos de desempenho.

### **5.3.2 Visualização do Conector**

Uma visão de conector de uma arquitetura concentra-se na mecânica da comunicação entre componentes. Para uma arquitetura baseada em REST, estamos particularmente interessados na restrições que definem a interface de recursos genéricos.

Os conectores de cliente examinam o identificador de recurso para selecionar um mecanismo de comunicação para cada solicitação. Por exemplo, um cliente pode ser configurado para

conectar a um componente de proxy específico, talvez um agindo como um filtro de anotação, quando o identificador indica que é um recurso local. Da mesma forma, um cliente pode ser configurado para rejeitar solicitações para algum subconjunto de identificadores.

REST não restringe a comunicação a um protocolo específico, mas restringe a interface entre os componentes e, portanto, o escopo de interação e implementação suposições que poderiam ser feitas entre os componentes. Por exemplo, a Web protocolo de transferência principal é HTTP, mas a arquitetura também inclui acesso contínuo a recursos que se originam em servidores de rede pré-existentes, incluindo FTP [107], Gopher [7], e WAIS [36]. A interação com esses serviços é restrita à semântica de um REST conector. Essa restrição sacrifica algumas das vantagens de outras arquiteturas, como a interação com estado de um protocolo de feedback de relevância como WAIS, a fim de manter a vantagens de uma interface única e genérica para a semântica do conector. Em contrapartida, o genérico interface torna possível acessar uma infinidade de serviços através de um único proxy. Se um aplicativo precisa dos recursos adicionais de outra arquitetura, ele pode implementar e invocar esses recursos como um sistema separado rodando em paralelo, semelhante a como a Web interfaces de arquitetura com recursos “telnet” e “mailto”.

#### **5.3.3 Visualização de Dados**

Uma visão de dados de uma arquitetura revela o estado do aplicativo à medida que as informações fluem os componentes. Como o REST é especificamente direcionado a sistemas de informação distribuídos, vê um aplicativo como uma estrutura coesa de informações e alternativas de controle através do qual um usuário pode executar uma tarefa desejada. Por exemplo, procurando uma palavra em um dicionário on-line é uma aplicação, assim como passear por um museu virtual ou revisar

um conjunto de notas de aula para estudar para um exame. Cada aplicativo define metas para o sistema, contra o qual o desempenho do sistema pode ser medido.

As interações de componentes ocorrem na forma de mensagens de tamanho dinâmico. Pequeno ou mensagens de grão médio são usadas para semântica de controle, mas a maior parte do trabalho do aplicativo é realizado por meio de mensagens de grande granularidade contendo uma representação completa de recursos.

A forma mais frequente de semântica de solicitação é a de recuperar uma representação de um recurso (por exemplo, o método "GET" em HTTP), que muitas vezes pode ser armazenado em cache para reutilização posterior.

REST concentra todo o estado de controle nas representações recebidas em resposta às interações. O objetivo é melhorar a escalabilidade do servidor, eliminando qualquer necessidade de servidor para manter um conhecimento do estado do cliente além da solicitação atual. Um o estado do aplicativo é, portanto, definido por suas solicitações pendentes, a topologia de componentes (alguns dos quais podem estar filtrando dados em buffer), as solicitações ativas nesses conectores, o fluxo de dados de representações em resposta a essas solicitações e o processamento dessas representações à medida que são recebidas pelo agente do usuário.

Um aplicativo atinge um estado estável sempre que não tiver solicitações pendentes; isto é, isso não tem solicitações pendentes e todas as respostas ao seu conjunto atual de solicitações foram completamente recebidas ou recebidas até o ponto em que possam ser tratadas como uma representação fluxo de dados. Para um aplicativo de navegador, esse estado corresponde a uma "página da Web", incluindo a representação primária e as representações auxiliares, como imagens em linha, miniaplicativos incorporados e folhas de estilo. O significado dos estados estacionários de aplicação é visto em seu impacto tanto no desempenho percebido pelo usuário quanto na explosão da solicitação de rede tráfego.

O desempenho percebido pelo usuário de um aplicativo de navegador é determinado pela latência entre estados estacionários: o período de tempo entre a seleção de um link hipermídia no uma página da web e o ponto em que as informações úteis foram renderizadas para a próxima web página. A otimização do desempenho do navegador é, portanto, centrada na redução desse latência de comunicação.

Como as arquiteturas baseadas em REST se comunicam principalmente por meio da transferência de representações de recursos, a latência pode ser afetada tanto pelo design do protocolos de comunicação e o desenho dos formatos de dados de representação. A habilidade de renderizar incrementalmente os dados de resposta à medida que são recebidos é determinado pelo design do tipo de mídia e a disponibilidade de informações de layout (dimensões visuais de objetos em linha) dentro de cada representação.

Uma observação interessante é que a solicitação de rede mais eficiente é aquela que não usar a rede. Em outras palavras, a capacidade de reutilizar uma resposta armazenada em cache resulta em um melhoria considerável no desempenho do aplicativo. Embora o uso de um cache adicione alguns latência para cada solicitação individual devido à sobrecarga de pesquisa, a latência média da solicitação é significativamente reduzido quando mesmo uma pequena porcentagem de solicitações resulta em acertos de cache utilizáveis.

O próximo estado de controle de uma aplicação reside na representação do primeiro recurso solicitado, portanto, obter essa primeira representação é uma prioridade. A interação REST é portanto, aprimorado por protocolos que “respondem primeiro e pensam depois”. Em outras palavras, um protocolo que requer múltiplas interações por ação do usuário, a fim de fazer coisas como negociar recursos de recursos antes de enviar uma resposta de conteúdo, será perceptivelmente mais lento do que um protocolo que envia o que é mais provável que seja o ideal primeiro e depois fornece uma lista de alternativas para o cliente recuperar se a primeira resposta for insatisfatória.

O estado do aplicativo é controlado e armazenado pelo agente do usuário e pode ser composto de representações de vários servidores. Além de liberar o servidor da problemas de escalabilidade de armazenamento de estado, isso permite que o usuário manipule diretamente o estado (por exemplo, o histórico de um navegador da Web), antecipar mudanças nesse estado (por exemplo, mapas de links e pré-busca de representações) e pular de um aplicativo para outro (por exemplo, marcadores e diálogos de entrada de URI).

A aplicação do modelo é, portanto, um motor que se move de um estado para o outro por examinar e escolher entre as transições de estado alternativas no conjunto atual de representações. Não surpreendentemente, isso corresponde exatamente à interface do usuário de um hipermídia navegador. No entanto, o estilo não pressupõe que todos os aplicativos sejam navegadores. Na verdade, o os detalhes do aplicativo são ocultos do servidor pela interface do conector genérico e, portanto, um agente de usuário poderia igualmente ser um robô automatizado realizando a recuperação de informações para um serviço de indexação, um agente pessoal procurando dados que correspondam a determinados critérios ou um aranha de manutenção ocupada patrulhando as informações em busca de referências quebradas ou modificadas conteúdo [39].

#### 5.4 Trabalho Relacionado

Bass, et al. [9] dedicam um capítulo sobre arquitetura para a World Wide Web, mas sua descrição abrange apenas a arquitetura de implementação dentro do CERN/W3C desenvolveu libwww (bibliotecas cliente e servidor) e software Jigsaw. Embora aqueles implementações refletem muitas das restrições de design do REST, tendo sido desenvolvidas por pessoas familiarizadas com o design arquitetônico e a lógica da Web, a verdadeira WWW arquitetura é independente de qualquer implementação única. A Web moderna é definida por

suas interfaces e protocolos padrão, não como essas interfaces e protocolos são implementado em um determinado software.

O estilo REST se baseia em muitos paradigmas de processos distribuídos preexistentes [6, 50], protocolos de comunicação e campos de software. As interações do componente REST são estruturado em um estilo cliente-servidor em camadas, mas as restrições adicionais do recurso genérico interface criam a oportunidade de substituibilidade e inspeção por intermediários.

Solicitações e respostas têm a aparência de um estilo de invocação remota, mas REST as mensagens são direcionadas a um recurso conceitual em vez de um identificador de implementação.

Várias tentativas foram feitas para modelar a arquitetura da Web como uma forma de sistema de arquivos distribuído (por exemplo, WebNFS) ou como um sistema de objeto distribuído [83]. No entanto, eles excluem vários tipos de recursos da Web ou estratégias de implementação como sendo “não interessante”, quando de fato sua presença invalida os pressupostos que fundamentam tais modelos. REST funciona bem porque não limita a implementação de recursos a determinados modelos predefinidos, permitindo que cada aplicação escolha uma implementação que melhor atende às suas próprias necessidades e permitindo a substituição de implementações sem impactando o usuário.

O método de interação de enviar representações de recursos para consumidores componentes tem alguns paralelos com estilos de integração baseada em eventos (EBI). A chave A diferença é que os estilos EBI são baseados em push. O componente que contém o estado (equivalente a um servidor de origem em REST) emite um evento sempre que o estado muda, se algum componente está realmente interessado ou atento a tal evento. No Estilo REST, consumindo componentes geralmente puxam representações. Embora isso seja menos

eficiente quando visto como um único cliente que deseja monitorar um único recurso, a escala de a Web torna inviável um modelo push não regulamentado.

O uso de princípios do estilo REST na Web, com sua noção clara de componentes, conectores e representações relaciona-se intimamente com o estilo arquitetônico C2 [128]. O C2 estilo suporta o desenvolvimento de aplicativos distribuídos e dinâmicos, concentrando-se em uso estruturado de conectores para obter independência de substrato. As aplicações C2 dependem notificação assíncrona de mudanças de estado e mensagens de solicitação. Assim como em outro evento esquemas baseados, C2 é nominalmente baseado em push, embora uma arquitetura C2 possa operar em Estilo pull do REST, emitindo apenas uma notificação após o recebimento de uma solicitação. No entanto, o O estilo C2 não possui as restrições amigáveis ao intermediário do REST, como o recurso genérico interface, interações sem estado garantidas e suporte intrínseco para armazenamento em cache.

## 5.5 Resumo

Este capítulo introduziu o estilo arquitetônico Representational State Transfer (REST) para sistemas hipermídia distribuídos. REST fornece um conjunto de restrições arquitetônicas que, quando aplicado como um todo, enfatiza a escalabilidade de interações de componentes, generalidade de interfaces, implantação independente de componentes e componentes intermediários para reduzir a latência de interação, reforçar a segurança e encapsular sistemas legados. Eu descrevi os princípios de engenharia de software que orientam o REST e as restrições de interação escolhidas manter esses princípios, ao mesmo tempo em que os contrasta com as restrições de outros estilos.

O próximo capítulo apresenta uma avaliação da arquitetura REST através da experiência e lições aprendidas com a aplicação do REST ao design, especificação e

implantação da arquitetura Web moderna. Este trabalho incluiu a autoria do atual Especificações de rastreamento de padrões da Internet do Hypertext Transfer Protocol (HTTP/1.1) e Uniform Resource Identifiers (URI) e implementando a arquitetura através do biblioteca de protocolos do cliente libwww-perl e servidor Apache HTTP.

## CAPÍTULO 6

### Experiência e Avaliação

Desde 1994, o estilo arquitetônico REST tem sido utilizado para orientar o projeto e desenvolvimento da arquitetura para a Web moderna. Este capítulo descreve a experiência e lições aprendidas com a aplicação do REST ao criar a Internet padrões para o protocolo de transferência de hipertexto (HTTP) e identificadores uniformes de recursos (URI), as duas especificações que definem a interface genérica usada por todos os componentes interações na Web, bem como da implantação dessas tecnologias na forma da biblioteca cliente libwww-perl, o Apache HTTP Server Project e outras implementações dos padrões de protocolo.

#### 6.1 Padronizando a Web

Conforme descrito no Capítulo 4, a motivação para desenvolver REST foi criar um modelo arquitetônico de como a Web deve funcionar, de modo que possa servir como guia framework para os padrões de protocolo da Web. REST foi aplicado para descrever o arquitetura da Web, ajudar a identificar problemas existentes, comparar soluções alternativas e garantir que as extensões de protocolo não violem as principais restrições que tornam a Web bem sucedido. Este trabalho foi feito como parte da Internet Engineering Taskforce (IETF) e esforços do World Wide Web Consortium (W3C) para definir os padrões de arquitetura para o Web: HTTP, URI e HTML.

Meu envolvimento no processo de padrões da Web começou no final de 1993, enquanto desenvolvia a biblioteca de protocolo libwww-perl que serviu como interface do conector do cliente para

MOMaranha [39]. Na época, a arquitetura da Web era descrita por um conjunto de notas de hipertexto [14], dois primeiros artigos introdutórios [12, 13], rascunho de especificações de hipertexto representando recursos propostos para a Web (alguns dos quais já haviam sido implementado), e o arquivo da lista de discussão pública www-talk que foi usada para discussão informal entre os participantes do projeto WWW em todo o mundo. Cada um dos especificações estavam significativamente desatualizadas quando comparadas com implementações da Web, principalmente devido à rápida evolução da Web após a introdução do mosaico gráfico navegador [NCSA]. Várias extensões experimentais foram adicionadas ao HTTP para permitir proxies, mas na maioria das vezes o protocolo assumiu uma conexão direta entre o usuário agente e um servidor de origem HTTP ou um gateway para sistemas legados. Não havia consciência dentro da arquitetura de cache, proxies ou aranhas, mesmo que implementações estavam prontamente disponíveis e correndo descontroladamente. Muitas outras extensões foram sendo proposto para inclusão nas próximas versões dos protocolos.

Ao mesmo tempo, havia uma pressão crescente dentro da indústria para padronizar alguma versão, ou versões, dos protocolos de interface da Web. O W3C foi formado por Berners-Lee [20] para atuar como um think-tank para arquitetura Web e fornecer a autoria recursos necessários para escrever os padrões da Web e implementações de referência, mas o a padronização em si era regida pela Internet Engineering Taskforce [[www.ietf.org](http://www.ietf.org)] e seus grupos de trabalho em URI, HTTP e HTML. Devido à minha experiência desenvolvendo Web software, fui escolhido pela primeira vez para criar a especificação para Relative URL [40], depois juntou-se a Henrik Frystyk Nielsen para criar a especificação HTTP/1.0 [19], tornou-se o principal arquiteto do HTTP/1.1 [42], e finalmente foi o autor da revisão da URL especificações para formar o padrão de sintaxe genérica de URI [21].

A primeira edição do REST foi desenvolvida entre outubro de 1994 e agosto de 1995, principalmente como um meio para comunicar conceitos da Web enquanto escrevemos o protocolo HTTP/1.0 especificação e a proposta inicial HTTP/1.1. Foi melhorado iterativamente ao longo do próximo cinco anos e aplicado a várias revisões e extensões dos padrões de protocolo da Web.

REST foi originalmente referido como o “modelo de objeto HTTP”, mas esse nome muitas vezes levar à má interpretação do mesmo como o modelo de implementação de um servidor HTTP. O nome “Transferência de Estado Representacional” pretende evocar uma imagem de como um bem projetado A aplicação web comporta-se: uma rede de páginas web (uma máquina de estado virtual), onde o usuário progride através do aplicativo selecionando links (transições de estado), resultando em a próxima página (representando o próximo estado do aplicativo) sendo transferida para o usuário e prestados para seu uso.

REST não se destina a capturar todos os usos possíveis dos padrões de protocolo da Web. Lá são aplicativos de HTTP e URI que não correspondem ao modelo de aplicativo de um sistema hipermídia. O ponto importante, no entanto, é que o REST captura todos esses aspectos de um sistema de hipermídia distribuído que são considerados centrais para o comportamento e requisitos de desempenho da Web, de modo que otimizar o comportamento dentro do resultará em um comportamento ideal dentro da arquitetura da Web implantada. Em outros palavras, REST é otimizado para o caso comum, de modo que as restrições que ele aplica ao A arquitetura da Web também será otimizada para o caso comum.

## 6.2 REST Aplicado ao URI

Uniform Resource Identifiers (URI) são os elementos mais simples da arquitetura da Web e o mais importante. URI são conhecidos por muitos nomes: endereços WWW,

Identificadores Universais de Documentos, Identificadores Universais de Recursos [15] e, finalmente, o combinação de Uniform Resource Locators (URL) [17] e Nomes (URN) [124]. Aparte de seu nome, a sintaxe URI permaneceu relativamente inalterada desde 1992. No entanto, a especificação de endereços da Web também define o escopo e a semântica do que queremos dizer por *recurso*, que mudou desde o início da arquitetura da Web. REST foi usado para definir o termo recurso para o padrão URI [21], bem como a semântica geral do interface genérica para manipulação de recursos através de suas representações.

### **6.2.1 Redefinição de Recurso**

A arquitetura da Web inicial definia URI como identificadores de documentos. Os autores foram instruídos para definir identificadores em termos de localização de um documento na rede. Protocolos da web poderia então ser usado para recuperar esse documento. No entanto, esta definição mostrou-se insatisfatório por vários motivos. Em primeiro lugar, sugere que o autor está identificando o conteúdo transferido, o que implicaria que o identificador deveria mudar sempre que o alterações de conteúdo. Em segundo lugar, existem muitos endereços que correspondem a um serviço em vez de do que um documento - os autores podem estar pretendendo direcionar os leitores para esse serviço, em vez de a qualquer resultado específico de um acesso prévio a esse serviço. Finalmente, existem endereços que não correspondem a um documento em alguns períodos de tempo, como quando o documento não ainda não existe ou quando o endereço está sendo usado apenas para nomear, em vez de localizar, em formação.

A definição de *recurso* em REST é baseada em uma premissa simples: os identificadores devem mudar com a menor frequência possível. Como a Web usa identificadores incorporados em vez de servidores de link, os autores precisam de um identificador que corresponda à semântica que pretendem por um

referência hipermídia, permitindo que a referência permaneça estática mesmo que o resultado da acessar essa referência pode mudar ao longo do tempo. REST faz isso definindo um recurso seja a semântica do que o autor pretende identificar, ao invés do valor correspondente a essa semântica no momento em que a referência é criada. Resta então ao autor para garantir que o identificador escolhido para uma referência identifica de fato o semântica.

### **6.2.2 Manipulando Sombras**

Definir *recurso* de forma que um URI identifique um conceito em vez de um documento nos deixa com outra pergunta: como um usuário acessa, manipula ou transfere um conceito tal que eles podem obter algo útil quando um link de hipertexto é selecionado? REST responde isso questão definindo as coisas que são manipuladas para serem *representações* do recurso, em vez do próprio recurso. Um servidor de origem mantém um mapeamento de identificadores de recursos ao conjunto de representações correspondentes a cada recurso. UMA recurso é, portanto, manipulado pela transferência de representações através do genérico interface definida pelo identificador de recurso.

A definição de recurso do REST deriva do requisito central da Web:  
autoria independente de hipertexto interconectado em vários domínios de confiança. Forçando as definições de interface para corresponder aos requisitos de interface fazem com que os protocolos pareçam vago, mas isso é apenas porque a interface que está sendo manipulada é apenas uma interface e não uma implementação. Os protocolos são específicos sobre a intenção de uma ação de aplicativo, mas o mecanismo por trás da interface deve decidir como essa intenção afeta o implementação do mapeamento de recursos para representações.

A ocultação de informações é um dos principais princípios de engenharia de software que motiva a interface uniforme do REST. Porque um cliente está restrito à manipulação de representações em vez de acessar diretamente a implementação de um recurso, o implementação pode ser construída em qualquer forma desejada pela autoridade de nomeação sem impactar os clientes que venham a utilizar suas representações. Além disso, se vários representações do recurso existem no momento em que é acessado, uma seleção de conteúdo algoritmo pode ser usado para selecionar dinamicamente uma representação que melhor se adapta às capacidades desse cliente. A desvantagem, é claro, é que a autoria remota de um recurso não é tão simples como a autoria remota de um arquivo.

### **6.2.3 Autoria Remota**

O desafio da autoria remota via interface uniforme da Web se deve à separação entre a representação que pode ser recuperada por um cliente e o mecanismo que pode ser usado no servidor para armazenar, gerar ou recuperar o conteúdo dessa representação. Um servidor individual pode mapear alguma parte de seu namespace para um sistema de arquivos, que por sua vez mapeia ao equivalente de um inode que pode ser mapeado em um local de disco, mas aqueles subjacentes mecanismos fornecem um meio de associar um recurso a um conjunto de representações em vez de do que identificar o próprio recurso. Muitos recursos diferentes podem ser mapeados para o mesmo representação, enquanto outros recursos podem não ter representação mapeada.

Para criar um recurso existente, o autor deve primeiro obter a fonte específica URI de recurso: o conjunto de URI que se vincula à representação subjacente do manipulador para o recurso alvo. Um recurso nem sempre é mapeado para um arquivo singular, mas todos os recursos que não são estáticos são derivados de alguns outros recursos, e seguindo a árvore de derivação

um autor pode eventualmente encontrar todos os recursos de origem que devem ser editados para modificar a representação de um recurso. Esses mesmos princípios se aplicam a qualquer forma de representação derivada, seja de negociação de conteúdo, scripts, servlets, configurações, versões, etc.

O recurso não é o objeto de armazenamento. O recurso não é um mecanismo que o servidor usa para manipular o objeto de armazenamento. O recurso é um mapeamento conceitual — o servidor recebe o identificador (que identifica o mapeamento) e o aplica ao seu mapeamento atual implementação (geralmente uma combinação de travessia de árvore profunda específica da coleção e/ou tabelas de hash) para encontrar a implementação do manipulador atualmente responsável e o manipulador. A implementação então seleciona a ação+resposta apropriada com base no conteúdo da solicitação.

Todos esses problemas específicos de implementação estão ocultos por trás da interface da Web; suas natureza não pode ser assumida por um cliente que só tem acesso através da interface Web.

Por exemplo, considere o que acontece quando um site cresce na base de usuários e decide para substituir seu antigo servidor Brand X, baseado em uma plataforma XOS, por um novo servidor Apache rodando no FreeBSD. O hardware de armazenamento em disco é substituído. O sistema operacional é substituído. O servidor HTTP é substituído. Talvez até o método de gerar respostas pois todo o conteúdo é substituído. No entanto, o que não precisa mudar é a Web interface: se projetado corretamente, o namespace no novo servidor pode espelhar o do antigo, ou seja, do ponto de vista do cliente, que só conhece recursos e não sobre como eles são implementados, nada mudou além da melhoria robustez do site.

#### 6.2.4 Semântica de ligação ao URI

Como mencionado acima, um recurso pode ter muitos identificadores. Em outras palavras, pode existir dois ou mais URIs diferentes que possuem semântica equivalente quando usados para acessar um servidor. Isto também é possível ter dois URIs que resultem no mesmo mecanismo sendo usado no acesso para o servidor e, no entanto, esses URI identificam dois recursos diferentes porque não significam a mesma coisa.

A semântica é um subproduto do ato de atribuir identificadores de recursos e preencher esses recursos com representações. Em nenhum momento o servidor ou cliente software precisa saber ou entender o significado de um URI - eles simplesmente agem como um canal através do qual o criador de um recurso (uma autoridade de nomeação humana) pode associar representações com a semântica identificada pelo URI. Em outras palavras, não há recursos no servidor; apenas mecanismos que fornecem respostas através de uma interface abstrata definidos por recursos. Pode parecer estranho, mas essa é a essência do que torna a Web trabalhar em tantas implementações diferentes.

É da natureza de todo engenheiro definir as coisas em termos das características do componentes que serão utilizados para compor o produto acabado. A Web não funciona isso caminho. A arquitetura da Web consiste em restrições no modelo de comunicação entre componentes, com base na função de cada componente durante uma ação do aplicativo. este impede que os componentes assumam qualquer coisa além da abstração de recursos, ocultando os mecanismos reais em ambos os lados da interface abstrata.

### 6.2.5 REST Incompatibilidades no URI

Como a maioria dos sistemas do mundo real, nem todos os componentes da arquitetura Web implantada obedecem cada constrangimento presente em seu projeto arquitetônico. REST tem sido usado tanto como um meio para definir melhorias arquitetônicas e identificar incompatibilidades arquitetônicas. Incompatibilidades ocorrem quando, por desconhecimento ou descuido, é implantada uma implementação de software que viola as restrições arquitetônicas. Embora as incompatibilidades não possam ser evitadas em geral, é possível identificá-los antes que se tornem padronizados.

Embora o design do URI corresponda à noção arquitetônica de identificadores do REST, a sintaxe por si só é insuficiente para forçar as autoridades de nomeação a definir seu próprio URI de acordo com o modelo de recursos. Uma forma de abuso é incluir informações que identifiquem o atual usuário dentro de todo o URI referenciado por uma representação de resposta hipermídia. Tal IDs de usuário incorporados podem ser usados para manter o estado da sessão no servidor, rastrear o comportamento do usuário registrando suas ações, ou transportar as preferências do usuário em várias ações (por exemplo, Hyper-G's portas [84]). No entanto, ao violar as restrições do REST, esses sistemas também causam cache compartilhado se torne ineficaz, reduza a escalabilidade do servidor e resulte em efeitos quando um usuário compartilha essas referências com outras pessoas.

Outro conflito com a interface de recursos do REST ocorre quando o software tenta tratar a Web como um sistema de arquivos distribuído. Como os sistemas de arquivos expõem a implementação de suas informações, existem ferramentas para “espelhar” essas informações para vários sites como um meio de balanceamento de carga e redistribuição do conteúdo para mais perto dos usuários. No entanto, eles podem fazer isso apenas porque os arquivos têm um conjunto fixo de semântica (uma sequência nomeada de bytes) que pode ser duplicado facilmente. Em contraste, as tentativas de espelhar o conteúdo de um servidor Web como arquivos falhará porque a interface do recurso nem sempre corresponde à semântica de um arquivo

sistema, e porque tanto os dados quanto os metadados estão incluídos e são significativos para o semântica de uma representação. O conteúdo do servidor Web pode ser replicado em sites remotos, mas apenas replicando todo o mecanismo e configuração do servidor, ou seletivamente replicando apenas os recursos com representações conhecidas como estáticas (por exemplo, cache as redes contratam sites da Web para replicar representações de recursos específicos para o "bordas" de toda a Internet para reduzir a latência e distribuir a carga para longe do servidor de origem).

### 6.3 REST Aplicado ao HTTP

O Hypertext Transfer Protocol (HTTP) tem um papel especial na arquitetura da Web, tanto o protocolo de nível de aplicativo primário para comunicação entre componentes da Web e o único protocolo projetado especificamente para a transferência de representações de recursos. Diferente URI, havia um grande número de mudanças necessárias para que o HTTP suportasse o arquitetura web moderna. Os desenvolvedores de implementações HTTP foram conservadores em sua adoção de melhorias propostas e, portanto, extensões necessárias para ser comprovados e submetidos à revisão de padrões antes que possam ser implantados. REST era usado para identificar problemas com as implementações HTTP existentes, especifique um subconjunto interoperável desse protocolo como HTTP/1.0 [19], analisar as extensões propostas para HTTP/1.1 [42], e fornecem uma justificativa motivacional para a implantação do HTTP/1.1.

As principais áreas problemáticas em HTTP que foram identificadas pelo REST incluíram planejamento para a implantação de novas versões de protocolo, separando a análise de mensagens do HTTP semântica e a camada de transporte subjacente (TCP), distinguindo entre autoridade e respostas não autoritárias, controle refinado de cache e vários aspectos de

o protocolo que falhou em ser autodescritivo. REST também foi usado para modelar o desempenho de aplicações Web baseadas em HTTP e antecipar o impacto de tais extensões como conexões persistentes e negociação de conteúdo. Finalmente, REST foi usado limitar o escopo de extensões HTTP padronizadas àquelas que se encaixam na arquitetura modelo, em vez de permitir que os aplicativos que abusam do HTTP influenciem igualmente o padrão.

### **6.3.1 Extensibilidade**

Um dos principais objetivos do REST é apoiar a implantação gradual e fragmentada de mudanças dentro de uma arquitetura já implantada. HTTP foi modificado para suportar esse objetivo através da introdução de requisitos de versão e regras para estender cada um dos elementos de sintaxe do protocolo.

#### *6.3.1.1 Versão de protocolo*

HTTP é uma família de protocolos, distinguidos por números de versão principal e secundária, que compartilham o nome principalmente porque correspondem ao protocolo esperado quando comunicando-se diretamente com um serviço baseado no namespace de URL "http". Um conector deve obedecer às restrições colocadas no elemento de protocolo da versão HTTP incluído em cada mensagem [90].

A versão HTTP de uma mensagem representa os recursos de protocolo do remetente e a compatibilidade bruta (número da versão principal) da mensagem que está sendo enviada. Isso permite uma cliente use um subconjunto reduzido (HTTP/1.0) de recursos para fazer um HTTP/1.1 normal pedido, ao mesmo tempo que indica ao destinatário que é capaz de suportar comunicação HTTP/1.1 completa. Em outras palavras, ele fornece uma forma provisória de protocolo

negociação na escala HTTP. Cada conexão em uma cadeia de solicitação/resposta pode operar em seu melhor nível de protocolo, apesar das limitações de alguns clientes ou servidores que fazem parte da cadeia.

A intenção do protocolo é que o servidor sempre responda com o maior versão secundária do protocolo que ele entende dentro da mesma versão principal do protocolo do cliente mensagem de solicitação. A restrição é que o servidor não pode usar esses recursos opcionais do protocolo de nível superior que são proibidos de serem enviados para um cliente de versão mais antiga. Lá não são recursos obrigatórios de um protocolo que não pode ser usado com todas as outras versões secundárias dentro dessa versão principal, uma vez que seria uma mudança incompatível e, portanto, exigiria uma alteração na versão principal. Os únicos recursos do HTTP que podem depender de um menor mudança de número de versão são aqueles que são interpretados por vizinhos imediatos no comunicação, porque o HTTP não exige que toda a cadeia de solicitação/resposta de componentes intermediários falam a mesma versão.

Essas regras existem para auxiliar na implantação de várias revisões de protocolo e para evitar que os arquitetos HTTP esqueçam que a implantação do protocolo é uma aspecto importante de seu projeto. Eles fazem isso facilitando a diferenciação entre alterações compatíveis no protocolo e alterações incompatíveis. As alterações compatíveis são fácil de implantar e a comunicação das diferenças pode ser alcançada dentro do protocolo fluxo. Mudanças incompatíveis são difíceis de implantar porque exigem alguns determinação de aceitação do protocolo antes que o fluxo de protocolo possa começar.

#### *6.3.1.2 Elementos de Protocolo Extensível*

O HTTP inclui vários namespaces separados, cada um com restrições diferentes, mas todos os quais compartilham o requisito de serem extensíveis sem limites. Alguns dos

namespaces são governados por padrões de Internet separados e compartilhados por vários protocolos (por exemplo, esquemas de URI [21], tipos de mídia [48], nomes de campos de cabeçalho MIME [47], valores de charset, tags de idioma), enquanto outros são governados por HTTP, incluindo os namespaces para o método nomes, códigos de status de resposta, nomes de campo de cabeçalho não MIME e valores dentro do padrão Campos de cabeçalho HTTP. Como o HTTP inicial não definia um conjunto consistente de regras sobre como alterações nesses namespaces pudessem ser implantadas, esse foi um dos primeiros problemas abordado pelo esforço de especificação.

A semântica de solicitação HTTP é indicada pelo nome do método de solicitação. Extensão do método é permitido sempre que um conjunto padronizado de semântica pode ser compartilhado entre cliente, servidor, e quaisquer intermediários que possam estar entre eles. Infelizmente, o HTTP inicial extensões, especificamente o método HEAD, fez a análise de uma resposta HTTP mensagem dependente de conhecer a semântica do método de solicitação. Isso levou a uma contradição de implantação: se um destinatário precisa conhecer a semântica de um método antes que ele pode ser encaminhado com segurança por um intermediário, então todos os intermediários devem ser atualizados antes que um novo método possa ser implantado.

Esse problema de implantação foi corrigido separando as regras de análise e encaminhamento Mensagens HTTP da semântica associada aos novos elementos do protocolo HTTP. Por exemplo, HEAD é o único método para o qual o campo de cabeçalho Content-Length tem um significado diferente de significar o comprimento do corpo da mensagem, e nenhum novo método pode mudar o cálculo do comprimento da mensagem. GET e HEAD também são os únicos métodos para os quais campos de cabeçalho de solicitação condicional têm a semântica de uma atualização de cache, enquanto para todos outros métodos eles têm o significado de uma pré-condição.

Da mesma forma, o HTTP precisava de uma regra geral para interpretar novos códigos de status de resposta, para que novas respostas possam ser implantadas sem prejudicar significativamente os clientes mais antigos. Portanto, expandimos a regra de que cada código de status pertencia a uma classe significada por o primeiro dígito do seu número decimal de três dígitos: 100-199 indicando que a mensagem contém uma resposta informativa provisória, 200-299, indicando que a solicitação sucesso, 300-399 indicando que a solicitação precisa ser redirecionada para outro recurso, 400-499 indicando que o cliente cometeu um erro que não deve ser repetido e 500-599 indicando que o servidor encontrou um erro, mas que o cliente pode obter uma resposta melhor mais tarde (ou através de algum outro servidor). Se um destinatário não entender a semântica específica de o código de status em uma determinada mensagem, eles devem tratá-lo da mesma maneira que o código x00 dentro de sua classe. Como a regra para nomes de métodos, esta regra de extensibilidade coloca um requisito na arquitetura atual, de modo que ele antecipe mudanças futuras. As alterações podem portanto, ser implantado em uma arquitetura existente com menos medo de componentes adversos reações.

#### 6.3.1.3 Atualização

A adição do campo de cabeçalho Upgrade no HTTP/1.1 reduz a dificuldade de implantação mudanças incompatíveis, permitindo que o cliente anuncie sua disposição para um melhor protocolo durante a comunicação em um fluxo de protocolo mais antigo. A atualização foi especificamente adicionado para suportar a substituição seletiva de HTTP/1.x por outros protocolos futuros que pode ser mais eficiente para algumas tarefas. Assim, o HTTP não só suporta extensibilidade, mas também substituição completa de si mesmo durante uma conexão ativa. Se o servidor suporta o protocolo aprimorado e deseja mudar, ele simplesmente responde com um 101 status e continua como se a solicitação fosse recebida nesse protocolo atualizado.

### 6.3.2 Mensagens Autodescritivas

REST restringe as mensagens entre os componentes para serem autodescritivas, a fim de suportar processamento intermediário de interações. No entanto, havia aspectos do HTTP inicial que falhou em ser autodescritivo, incluindo a falta de identificação do host nas solicitações, falha em distinguir sintaticamente entre dados de controle de mensagem e representação metadados, falha em diferenciar entre dados de controle destinados apenas para o imediato peer de conexão versus metadados destinados a todos os destinatários, falta de suporte para extensões e a necessidade de metadados para descrever representações com codificações em camadas.

#### 6.3.2.1 Anfitrião

Um dos piores erros no projeto HTTP inicial foi a decisão de não enviar o URI completo que é o destino de uma mensagem de solicitação, mas envia apenas as partes que não foram usados na configuração da conexão. A suposição era que um servidor conhecer sua própria autoridade de nomenclatura com base no endereço IP e na porta TCP da conexão. No entanto, isso não previu que várias autoridades de nomeação podem existir em um único servidor, que se tornou um problema crítico à medida que a Web crescia a uma taxa exponencial e novos nomes de domínio (a base para a autoridade de nomenclatura dentro do namespace http URL) excede a disponibilidade de novos endereços IP.

A solução definida e implantada para HTTP/1.0 e HTTP/1.1 deveria incluir as informações de host da URL de destino em um campo de cabeçalho Host da mensagem de solicitação. A implantação desse recurso foi considerada tão importante que a especificação HTTP/1.1 exige que os servidores rejeitem qualquer solicitação HTTP/1.1 que não inclua um campo Host. Como um resultado, agora existem muitos servidores ISP grandes que executam dezenas de milhares de sites de host virtual em um único endereço IP.

### 6.3.2.2 Codificações em Camadas

HTTP herdou sua sintaxe para descrever metadados de representação do Multipurpose

Extensões de correio do Internet (MIME) [47]. MIME não define tipos de mídia em camadas,

preferindo incluir apenas o rótulo do tipo de mídia mais externo dentro do

Valor do campo Content-Type. No entanto, isso impede um destinatário de determinar a natureza

de uma mensagem codificada sem decodificar as camadas. Uma extensão HTTP inicial funcionou

em torno desta falha listando as codificações externas separadamente dentro do Content-Encoding

campo e colocando o rótulo para o tipo de mídia mais interno no Content-Type. Isso foi um

má decisão de design, uma vez que mudou a semântica de Content-Type sem alterar sua

nome do campo, resultando em confusão sempre que os agentes de usuário mais antigos encontravam o ramal.

Uma solução melhor teria sido continuar tratando Content-Type como o mais externo

tipo de mídia e use um novo campo para descrever os tipos aninhados nesse tipo.

Infelizmente, a primeira extensão foi implantada antes que suas falhas fossem identificadas.

REST identificou a necessidade de outra camada de codificações: aquelas colocadas em uma mensagem

por um conector para melhorar sua transferibilidade pela rede. Essa nova camada,

chamado de codificação de transferência em referência a um conceito semelhante em MIME, permite que as mensagens

ser codificado para transferência sem implicar que a representação seja codificada por natureza.

Codificações de transferência podem ser adicionadas ou removidas por agentes de transferência, por qualquer motivo,

sem alterar a semântica da representação.

### 6.3.2.3 Independência Semântica

Conforme descrito acima, a análise de mensagens HTTP foi separada de sua semântica.

A análise de mensagens, incluindo localizar e agrupar os campos de cabeçalho, ocorre

totalmente separado do processo de análise do conteúdo do campo de cabeçalho. Nesse caminho,

intermediários podem processar e encaminhar rapidamente mensagens HTTP, e as extensões podem ser implantado sem quebrar os analisadores existentes.

#### *6.3.2.4 Independência de Transporte*

O HTTP inicial, incluindo a maioria das implementações de HTTP/1.0, usava o transporte subjacente protocolo como meio para sinalizar o fim de uma mensagem de resposta. Um servidor seria indicar o fim de um corpo de mensagem de resposta fechando a conexão TCP. Infelizmente, isso criou uma condição de falha significativa no protocolo: um cliente não tinha meios para distinguir entre uma resposta completa e uma que foi truncada por alguma falha de rede errônea. Para resolver isso, os campos de cabeçalho Content-Length foram redefinido dentro do HTTP/1.0 para indicar o comprimento do corpo da mensagem em bytes, sempre que o comprimento é conhecido antecipadamente, e a codificação de transferência "em partes" foi introduzida para HTTP/1.1.

A codificação em partes permite uma representação cujo tamanho é desconhecido no início de sua geração (quando os campos de cabeçalho são enviados) para ter seus limites delineados por uma série de pedaços que podem ser dimensionados individualmente antes de serem enviados. Também permite que os metadados ser enviados no final da mensagem como trailers, permitindo a criação de metadados opcionais em a origem enquanto a mensagem está sendo gerada, sem adicionar latência de resposta.

#### *6.3.2.5 Limites de Tamanho*

Uma barreira frequente para a flexibilidade dos protocolos da camada de aplicação é a tendência de especificar limites de tamanho nos parâmetros do protocolo. Embora sempre existam algumas práticas limites dentro de implementações do protocolo (por exemplo, memória disponível), especificando aqueles limites dentro do protocolo restringe todos os aplicativos aos mesmos limites, independentemente de sua

implementação. O resultado é muitas vezes um protocolo de menor denominador comum que não pode ser estendido muito além da visão de seu criador original.

Não há limite no protocolo HTTP para o comprimento do URI, o comprimento do cabeçalho campos, o comprimento de uma representação ou o comprimento de qualquer valor de campo que consiste em uma lista de itens. Embora os clientes da Web mais antigos tenham um problema bem conhecido com URI que consiste em mais de 255 caracteres, basta notar que problema na especificação HTTP em vez de exigir que todos os servidores sejam tão limitados. A razão pela qual isso não faz um máximo do protocolo é que os aplicativos dentro de um contexto controlado (como uma intranet) pode evitar esses limites substituindo os componentes mais antigos.

Embora não precisemos inventar limitações artificiais, o HTTP/1.1 precisou definir um conjunto apropriado de códigos de status de resposta para indicar quando um determinado elemento de protocolo é muito tempo para um servidor processar. Esses códigos de resposta foram adicionados para as condições de Request-URI muito longo, campo de cabeçalho muito longo e corpo muito longo. Infelizmente, não há forma de um cliente indicar a um servidor que pode ter limites de recursos, o que leva a problemas quando dispositivos com recursos limitados, como PDAs, tentam usar HTTP sem um intermediário específico do dispositivo ajustando a comunicação.

#### **6.3.2.6 Controle de Cache**

Porque o REST tenta equilibrar a necessidade de um comportamento eficiente e de baixa latência com o desejo para um comportamento de cache semanticamente transparente, é fundamental que o HTTP permita que o aplicativo para determinar os requisitos de cache em vez de codificá-los no próprio protocolo. o coisa mais importante para o protocolo fazer é descrever completa e precisamente os dados sendo transferido, para que nenhum aplicativo seja enganado pensando que tem uma coisa quando

realmente tem outra coisa. HTTP/1.1 faz isso através da adição do Cache  
Campos de cabeçalho Control, Age, Etag e Vary.

#### *6.3.2.7 Negociação de Conteúdo*

Todos os recursos mapeiam uma solicitação (consistindo de método, identificador, campos de cabeçalho de solicitação e às vezes uma representação) a uma resposta (consistindo em um código de status, cabeçalho de resposta campos e, às vezes, uma representação). Quando uma solicitação HTTP é mapeada para vários representações no servidor, o servidor pode se envolver em negociação de conteúdo com o cliente a fim de determinar qual deles melhor atende às necessidades do cliente. Isso é realmente mais um processo de “seleção de conteúdo” do que de negociação.

Embora houvesse várias implementações implantadas de negociação de conteúdo, foi não incluído na especificação de HTTP/1.0 porque não havia nenhum subconjunto interoperável de implementações no momento em que foi publicado. Isso se deveu, em parte, a uma má implementação dentro do NCSA Mosaic, que enviaria 1 KB de informações de preferência nos campos de cabeçalho em cada solicitação, independentemente da negociabilidade do recurso [125].

Como muito menos de 0,01% de todos os URIs são negociáveis em conteúdo, o resultado foi substancialmente maior latência de solicitação para ganho muito pequeno, o que levou navegadores posteriores a desconsiderar a recursos de negociação do HTTP/1.0.

A negociação preemptiva (orientada pelo servidor) ocorre quando o servidor varia a resposta representação para um método de solicitação específico\*identificador\*combinação de código de status de acordo com o valor dos campos do cabeçalho da solicitação, ou algo externo ao normal parâmetros de solicitação acima. O cliente precisa ser notificado quando isso ocorrer, para que um cache pode saber quando é semanticamente transparente usar uma resposta em cache específica para um solicitação futura, e também para que um agente do usuário possa fornecer preferências mais detalhadas do que

normalmente pode enviar uma vez que sabe que está afetando a resposta recebida.

O HTTP/1.1 introduziu o campo de cabeçalho *Vary* para esta finalidade. Variar simplesmente lista a solicitação dimensões do campo de cabeçalho sob as quais a resposta pode variar.

Na negociação preemptiva, o agente do usuário informa ao servidor do que ele é capaz. aceitando. O servidor deve então selecionar a representação que melhor corresponde ao que o agente do usuário afirma ser suas capacidades. No entanto, este é um problema não tratável porque requer não só informação sobre o que o UA irá aceitar, mas também quanto bem aceita cada recurso e com que finalidade o usuário pretende colocar a representação. Por exemplo, um usuário que deseja visualizar uma imagem na tela pode preferir um bitmap simples representação, mas o mesmo usuário com o mesmo navegador pode preferir um PostScript representação caso pretendam enviá-lo para uma impressora. Também depende do usuário configurando corretamente seu navegador de acordo com suas próprias preferências de conteúdo pessoal. Em suma, um servidor raramente é capaz de fazer uso efetivo da negociação preemptiva, mas foi a única forma de seleção de conteúdo automatizada definida pelo HTTP inicial.

O HTTP/1.1 adicionou a noção de negociação reativa (dirigida por agente). Neste caso, quando um agente do usuário solicita um recurso negociado, o servidor responde com uma lista dos recursos disponíveis representações. O agente do usuário pode então escolher qual é o melhor de acordo com sua própria capacidades e propósito. As informações sobre as representações disponíveis podem ser fornecido por meio de uma representação separada (por exemplo, uma resposta 300), dentro dos dados de resposta (por exemplo, HTML condicional), ou como um complemento para a resposta "mais provável". Este último funciona melhor para a Web porque uma interação adicional só se torna necessária se o usuário agente decide que uma das outras variantes seria melhor. A negociação reativa é simplesmente uma

reflexo automatizado do modelo de navegador normal, o que significa que pode tirar o máximo proveito de todos os benefícios de desempenho do REST.

Tanto a negociação preventiva quanto a reativa sofrem com a dificuldade de comunicação as características reais das dimensões de representação (por exemplo, como dizer que um navegador suporta tabelas HTML, mas não o elemento INSERT). No entanto, a negociação reativa as vantagens distintas de não ter que enviar preferências a cada solicitação, tendo mais informações de contexto com as quais tomar uma decisão quando confrontados com alternativas, e não interferindo nos caches.

Uma terceira forma de negociação, negociação transparente [64], é uma licença para um cache intermediário para atuar como um agente, em nome de outros agentes, para selecionar um melhor representação e iniciar solicitações para recuperar essa representação. O pedido pode ser resolvido internamente por outro acerto de cache e, portanto, é possível que nenhuma rede adicional pedido será feito. Ao fazer isso, no entanto, eles estão realizando uma negociação orientada ao servidor, e, portanto, deve adicionar as informações Vary apropriadas para que outros caches de saída não ficará confuso.

### **6.3.3 Desempenho**

HTTP/1.1 focado em melhorar a semântica de comunicação entre componentes, mas também houve algumas melhorias no desempenho percebido pelo usuário, embora limitado por o requisito de compatibilidade de sintaxe com HTTP/1.0.

#### *6.3.3.1 Conexões Persistentes*

Embora o comportamento de solicitação/resposta única por conexão do HTTP inicial tenha sido feito para simples implementações, resultou em uso inefficiente do transporte TCP subjacente devido à

sobrecarga dos custos de configuração por interação e a natureza do congestionamento de início lento do TCP algoritmo de controle [63, 125]. Como resultado, várias extensões foram propostas para combinar várias solicitações e respostas em uma única conexão.

A primeira proposta foi definir um novo conjunto de métodos para encapsular múltiplos solicitações dentro de uma única mensagem (MGET, MHEAD, etc.) e retornando a resposta como um MIME multipartes. Isso foi rejeitado porque violou várias das restrições REST.

Primeiro, o cliente precisaria conhecer todas as solicitações que deseja empacotar antes da primeira solicitação pode ser gravada na rede, pois um corpo de solicitação deve ser delimitado por comprimento por um campo de comprimento de conteúdo definido nos campos de cabeçalho de solicitação inicial. Em segundo lugar, os intermediários tem que extrair cada uma das mensagens para determinar quais poderia satisfazer localmente.

Finalmente, ele efetivamente dobraria o número de métodos de solicitação e complica mecanismos para negar seletivamente o acesso a determinados métodos.

Em vez disso, adotamos uma forma de conexões persistentes, que usa mensagens para enviar várias mensagens HTTP em uma única conexão [100]. Por HTTP/1.0, isso foi feito usando a diretiva “keep-alive” dentro do cabeçalho Connection campo. Infelizmente, isso não funcionou em geral porque o campo de cabeçalho poderia ser encaminhados por intermediários para outros intermediários que não entendem keep-alive, resultando em uma condição de bloqueio. O HTTP/1.1 acabou decidindo tornar persistente conexões o padrão, sinalizando assim sua presença através do valor da versão HTTP, e apenas usando a diretiva de conexão “fechar” para reverter o padrão.

É importante notar que conexões persistentes só se tornaram possíveis após HTTP mensagens foram redefinidas para serem autodescritivas e independentes do protocolo de transporte.

### 6.3.3.2 Cache de Gravação

HTTP não suporta cache de write-back. Um cache HTTP não pode assumir que o que fica escrito através dele é o mesmo que seria recuperável de uma solicitação subsequente de esse recurso e, portanto, não pode armazenar em cache um corpo de solicitação PUT e reutilizá-lo para um GET posterior resposta. Existem duas razões para esta regra: 1) os metadados podem ser gerados por trás do cenas e 2) o controle de acesso em solicitações GET posteriores não pode ser determinado a partir do PUT solicitar. No entanto, como as ações de escrita usando a Web são extremamente raras, a falta de escrita o cache de retorno não tem um impacto significativo no desempenho.

### 6.3.4 Incompatibilidades REST em HTTP

Existem várias incompatibilidades arquitetônicas presentes no HTTP, algumas devido a terceiros extensões que foram implantadas fora do processo de padrões e outras devido à necessidade de permanecer compatível com os componentes HTTP/1.0 implantados.

#### 6.3.4.1 Diferenciando Respostas Não Autoritárias

Uma fraqueza que ainda existe no HTTP é que não existe um mecanismo consistente para diferenciar entre respostas autoritativas, que são geradas pelo servidor de origem em resposta à solicitação atual e respostas não autorizadas que são obtidas de um intermediário ou cache sem acessar o servidor de origem. A distinção pode ser importante para aplicações que exigem respostas autoritárias, como a segurança crítica dispositivos de informação usados no setor de saúde e para aqueles momentos em que um erro resposta é retornada e o cliente fica se perguntando se o erro foi devido ao origem ou a algum intermediário. As tentativas de resolver isso usando códigos de status adicionais não ter sucesso, uma vez que a natureza autoritária é geralmente ortogonal ao status de resposta.

O HTTP/1.1 adicionou um mecanismo para controlar o comportamento do cache de tal forma que o desejo de um resposta autoritária pode ser indicada. A diretiva 'no-cache' em uma mensagem de solicitação requer que qualquer cache encaminhe a solicitação para o servidor de origem, mesmo que tenha um cache cópia do que está sendo solicitado. Isso permite que um cliente atualize uma cópia em cache que é sabidamente corrompidos ou obsoletos. No entanto, o uso regular deste campo interfere na os benefícios de desempenho do armazenamento em cache. Uma solução mais geral seria exigir que respostas sejam marcadas como não autorizadas sempre que uma ação não resultar em contato o servidor de origem. Um campo de cabeçalho de resposta de aviso foi definido em HTTP/1.1 para este propósito (e outros), mas não tem sido amplamente implementado na prática.

#### 6.3.4.2 Cookies

Um exemplo de onde uma extensão inadequada foi feita ao protocolo para suportar características que contradizem as propriedades desejadas da interface genérica é a introdução de informações de estado de todo o site na forma de cookies HTTP [73]. A interação do cookie não corresponder ao modelo de estado do aplicativo REST, muitas vezes resultando em confusão para o típico aplicativo do navegador.

Um cookie HTTP é um dado opaco que pode ser atribuído pelo servidor de origem a um usuário agente, incluindo-o dentro de um campo de cabeçalho de resposta Set-Cookie, com a intenção de ser que o agente do usuário deve incluir o mesmo cookie em todas as solicitações futuras a esse servidor até é substituído ou expira. Esses cookies geralmente contêm uma série de informações específicas do usuário opções de configuração ou um token a ser comparado com o banco de dados do servidor em futuras solicitações de. O problema é que um cookie é definido como anexado a qualquer solicitação futura para um determinado conjunto de identificadores de recursos, geralmente abrangendo um site inteiro, em vez de sendo associado ao estado do aplicativo específico (o conjunto de

representações) no navegador. Quando a funcionalidade do histórico do navegador (o "Voltar" botão) é posteriormente usado para fazer backup de uma visualização anterior àquela refletida pelo cookie, o estado do aplicativo do navegador não corresponde mais ao estado armazenado representado na bolacha. Portanto, a próxima solicitação enviada ao mesmo servidor conterá um cookie que deturpa o contexto de aplicação atual, levando a confusão em ambos os lados.

Os cookies também violam o REST porque permitem que os dados sejam passados sem identificando sua semântica, tornando-se uma preocupação tanto para segurança quanto para privacidade. A combinação de cookies com o campo de cabeçalho Referer [sic] permite rastrear um usuário enquanto navegam entre os sites.

Como resultado, os aplicativos baseados em cookies na Web nunca serão confiáveis. O mesmo a funcionalidade deveria ter sido realizada por meio de autenticação anônima e estado do lado do cliente. Um mecanismo de estado que envolve preferências pode ser mais eficiente implementado usando o uso criterioso de URI de configuração de contexto em vez de cookies, onde judicioso significa um URI por estado em vez de um número ilimitado de URI devido ao incorporação de um ID de usuário. Da mesma forma, o uso de cookies para identificar um "shopping basket" dentro de um banco de dados do lado do servidor pode ser implementado de forma mais eficiente, definindo a semântica dos itens de compras dentro dos formatos de dados hipermídia, permitindo ao usuário agente para selecionar e armazenar esses itens em sua própria cesta de compras do lado do cliente, completo com um URI a ser usado para check-out quando o cliente estiver pronto para comprar.

#### *6.3.4.3 Extensões Obrigatórias*

Os nomes dos campos de cabeçalho HTTP podem ser estendidos à vontade, mas somente quando as informações conter não é necessário para a compreensão adequada da mensagem. Campo de cabeçalho obrigatório extensões requerem uma grande revisão de protocolo ou uma mudança substancial na semântica do método,

como o proposto em [94]. Este é um aspecto da arquitetura moderna da Web que não ainda não correspondem às restrições de mensagens autodescritivas do estilo de arquitetura REST, principalmente porque o custo de implementação de uma estrutura de extensão obrigatória dentro do sintaxe HTTP existente excede quaisquer benefícios claros que podemos obter com a obrigatoriedade extensões. No entanto, é razoável esperar que as extensões de nome de campo obrigatórias será suportado na próxima grande revisão do HTTP, quando as restrições existentes sobre compatibilidade com versões anteriores da sintaxe não se aplica mais.

#### 6.3.4.4 Misturando Metadados

O HTTP foi projetado para estender a interface do conector genérico em uma conexão de rede. Como tal, destina-se a corresponder às características dessa interface, incluindo o delineamento de parâmetros como dados de controle, metadados e representação. No entanto, dois dos as limitações mais significativas da família de protocolos HTTP/1.x são que ela não consegue distinguir sintaticamente entre metadados de representação e controle de mensagens informações (ambos transmitidos como campos de cabeçalho) e não permite que metadados sejam efetivamente em camadas para verificações de integridade de mensagens.

REST identificou isso como limitações no protocolo no início da padronização processo, antecipando que levariam a problemas na implantação de outros recursos, como conexões persistentes e autenticação digest. Soluções alternativas foram desenvolvidas, incluindo a adição do campo de cabeçalho Connection para identificar dados de controle por conexão que não é seguro ser encaminhado por intermediários, assim como um algoritmo para o tratamento de resumos de campo de cabeçalho [46].

#### 6.3.4.5 Sintaxe MIME

HTTP herdou sua sintaxe de mensagem de MIME [47] para manter a semelhança com outros protocolos da Internet e reutilizar muitos dos campos padronizados para descrever mídia tipos nas mensagens. Infelizmente, MIME e HTTP têm objetivos muito diferentes, e o sintaxe é projetada apenas para os objetivos do MIME.

Em MIME, um agente de usuário está enviando um monte de informações, que devem ser tratados como um todo coerente, a um destinatário desconhecido com o qual nunca interagir. MIME assume que o agente gostaria de enviar todas essas informações em um mensagem, já que enviar várias mensagens pelo correio da Internet é menos eficiente. Desta forma, A sintaxe MIME é construída para empacotar mensagens dentro de uma parte ou multipartes em grande parte do como as transportadoras postais embrulham os pacotes em papel extra.

Em HTTP, empacotar objetos diferentes em uma única mensagem não faz sentido exceto para encapsulamento seguro ou archives empacotados, pois é mais eficiente fazer solicitações separadas para os documentos ainda não armazenados em cache. Assim, os aplicativos HTTP usam tipos de mídia como HTML como contêineres para referências ao “pacote” – um agente de usuário pode em seguida, escolha quais partes do pacote serão recuperadas como solicitações separadas. Embora seja possível que o HTTP possa usar um pacote multipartes no qual apenas os recursos não URI foram incluídos após a primeira parte, não houve muita demanda por ele.

O problema com a sintaxe MIME é que ela assume que o transporte tem perdas, deliberadamente corrompendo coisas como quebras de linha e comprimentos de conteúdo. A sintaxe é, portanto, detalhada e ineficiente para qualquer sistema que não seja baseado em um transporte com perdas, o que o torna inadequado para HTTP. Como o HTTP/1.1 tem a capacidade de oferecer suporte à implantação de protocolos, manter a sintaxe MIME não será necessário para a próxima versão principal do

HTTP, embora provavelmente continue a usar os muitos elementos de protocolo padronizados para metadados de representação.

### **6.3.5 Correspondência de respostas a solicitações**

As mensagens HTTP não são autodescritivas quando se trata de descrever qual resposta pertence a qual pedido. O HTTP inicial era baseado em uma única solicitação e resposta por conexão, portanto, não havia necessidade percebida de dados de controle de mensagens que vinculassem o resposta de volta ao pedido que o invocou. Assim, a ordenação dos pedidos determina a ordenação das respostas, o que significa que o HTTP depende do transporte conexão para determinar a correspondência.

HTTP/1.1, embora definido para ser independente do protocolo de transporte, ainda assume que a comunicação ocorre em um transporte síncrono. Poderia facilmente ser estendido para trabalhar em um transporte assíncrono, como e-mail, por meio da adição de uma solicitação identificador. Tal extensão seria útil para agentes em uma transmissão ou multicast situação, onde as respostas podem ser recebidas em um canal diferente daquele do solicitar. Além disso, em uma situação em que muitas solicitações estão pendentes, isso permitiria que o servidor escolha a ordem em que as respostas são transferidas, de modo que as respostas menores ou mais significativas as respostas são enviadas primeiro.

## **6.4 Transferência de Tecnologia**

Embora REST tenha tido sua influência mais direta sobre a autoria de padrões Web, A validação de seu uso como modelo de projeto arquitetônico veio através da implantação do padrões na forma de implementações de nível comercial.

#### 6.4.1 Experiência de implantação com libwww-perl

Meu envolvimento na definição de padrões Web começou com o desenvolvimento do robô de manutenção MOMspider [39] e sua biblioteca de protocolos associada, libwww-perl. Modelado após o libwww original desenvolvido por Tim Berners-Lee e o projeto WWW no CERN, libwww-perl forneceu uma interface uniforme para fazer solicitações da Web e interpretando respostas da Web para aplicações cliente escritas na linguagem Perl [134]. Isto foi a primeira biblioteca de protocolos da Web a ser desenvolvida independente do CERN original projeto, refletindo uma interpretação mais moderna da interface Web do que estava presente no bases de código mais antigas. Essa interface se tornou a base para o design do REST.

libwww-perl consistia em uma única interface de solicitação que usava a autoavaliação do Perl recursos de código para carregar dinamicamente o pacote de protocolo de transporte apropriado com base no esquema do URI solicitado. Por exemplo, quando solicitado a fazer uma solicitação “GET” no URL <<http://www.ebuilt.com/>>, libwww-perl extrairia o esquema da URL (“http”) e use-o para construir uma chamada para `wwwhttp'request()`, usando uma interface que foi comum a todos os tipos de recursos (HTTP, FTP, WAIS, arquivos locais, etc.). Para alcançar nessa interface genérica, a biblioteca tratava todas as chamadas da mesma maneira que um proxy HTTP. Ele forneceu uma interface usando estruturas de dados Perl que tinham a mesma semântica de um HTTP solicitação, independentemente do tipo de recurso.

libwww-perl demonstrou os benefícios de uma interface genérica. Dentro de um ano de sua lançamento inicial, mais de 600 desenvolvedores de software independentes estavam usando a biblioteca para seus próprias ferramentas de cliente, desde scripts de download de linha de comando até navegadores completos. Isto é atualmente a base para a maioria das ferramentas de administração de sistemas Web.

#### 6.4.2 Experiência de implantação com Apache

À medida que o esforço de especificação para HTTP começou a tomar a forma de especificações completas, software de servidor necessário que pudesse demonstrar efetivamente o padrão proposto protocolo e servir como um banco de testes para extensões que valem a pena. Na época, o mais popular O servidor HTTP (httpd) foi o software de domínio público desenvolvido por Rob McCool na Centro Nacional de Aplicações de Supercomputação, Universidade de Illinois, Urbana Champaign (NCSA). No entanto, o desenvolvimento parou depois que Rob deixou a NCSA em meados 1994, e muitos webmasters desenvolveram suas próprias extensões e correções de bugs que foram precisam de uma distribuição comum. Um grupo de nós criou uma lista de discussão com o propósito de coordenando nossas mudanças como "patches" para a fonte original. No processo, criamos o Projeto Apache HTTP Server [89].

O projeto Apache é um esforço colaborativo de desenvolvimento de software destinado a criar um implementação de software de código aberto robusto, de nível comercial, completo e de código aberto de um servidor HTTP. O projeto é gerenciado em conjunto por um grupo de voluntários localizados ao redor do mundo, usando a Internet e a Web para comunicar, planejar e desenvolver o servidor e sua documentação relacionada. Esses voluntários são conhecidos como Grupo Apache. Mais recentemente, o grupo formou a Apache Software Foundation, sem fins lucrativos, para atuar como representante legal e organização guarda-chuva financeira para apoiar o desenvolvimento contínuo do Apache aberto projetos de origem.

O Apache tornou-se conhecido por seu comportamento robusto em resposta às variadas demandas de um serviço de Internet e por sua implementação rigorosa dos padrões do protocolo HTTP. Atuei como o "policial do protocolo" dentro do Grupo Apache, escrevendo código para o núcleo HTTP funções de análise, apoiando os esforços de outros, explicando os padrões e agindo

como um defensor da visão dos desenvolvedores do Apache sobre “o jeito certo de implementar HTTP” dentro dos fóruns de padrões. Muitas das lições descritas neste capítulo foram aprendidas como resultado da criação e teste de várias implementações de HTTP dentro do Apache projeto, e submetendo as teorias por trás do protocolo às críticas do Grupo Apache Reveja.

Apache httpd é amplamente considerado como um dos projetos de software de maior sucesso, e um dos primeiros produtos de software de código aberto a dominar um mercado no qual existe concorrência comercial significativa. A pesquisa da Netcraft de julho de 2000 sobre a Internet pública sites encontraram mais de 20 milhões de sites baseados no software Apache, representando mais de 65% de todos os sites pesquisados [<http://www.netcraft.com/survey/>]. Apache foi o primeiro grande servidor para suportar o protocolo HTTP/1.1 e geralmente é considerado a referência implementação contra a qual todo o software cliente é testado. O Grupo Apache recebeu o 1999 ACM Software System Award como reconhecimento do nosso impacto nos padrões para o Arquitetura da Web.

#### **6.4.3 Implantação de software compatível com URI e HTTP/1.1**

Além do Apache, muitos outros projetos, tanto de natureza comercial quanto de código aberto, adotaram e implantaram produtos de software baseados nos protocolos da Web moderna arquitetura. Embora possa ser apenas uma coincidência, o Microsoft Internet Explorer ultrapassou Netscape Navigator na fatia de mercado de navegadores da Web logo após serem os primeiros navegador principal para implementar o padrão de cliente HTTP/1.1. Além disso, muitos dos extensões HTTP individuais que foram definidas durante o processo de padronização, como o campo de cabeçalho do host, agora alcançaram a implantação universal.

O estilo arquitetural REST conseguiu orientar o projeto e a implantação do arquitetura web moderna. Até o momento, não houve problemas significativos causados pela introdução das novas normas, ainda que tenham sido sujeitas a alterações graduais e implantação fragmentada ao lado de aplicativos Web legados. Além disso, o novo padrões tiveram um efeito positivo na robustez da Web e permitiram novos métodos para melhorar o desempenho percebido pelo usuário por meio de hierarquias de cache e redes de distribuição de conteúdo.

## 6.5 Lições de Arquitetura

Há uma série de lições gerais de arquitetura a serem aprendidas com a Web moderna. arquitetura e os problemas identificados pelo REST.

### 6.5.1 Vantagens de uma API baseada em rede

O que distingue a Web moderna de outros middlewares [22] é a forma como ela usa HTTP como uma interface de programação de aplicativos (API) baseada em rede. Isso não foi sempre o caso. O design inicial da Web fez uso de um pacote de biblioteca, CERN libwww, como a biblioteca de implementação única para todos os clientes e servidores. CERN libwww forneceu um API baseada em biblioteca para construir componentes Web interoperáveis.

Uma API baseada em biblioteca fornece um conjunto de pontos de entrada de código e símbolos associados conjuntos de parâmetros para que um programador possa usar o código de outra pessoa para fazer o trabalho sujo de mantendo a interface real entre sistemas semelhantes, desde que o programador obedece às restrições arquitetônicas e de linguagem que acompanham esse código. A suposição é que todos os lados da comunicação usam a mesma API e, portanto, os internos do interface são importantes apenas para o desenvolvedor da API e não para o desenvolvedor do aplicativo.

A abordagem de biblioteca única terminou em 1993 porque não correspondia à dinâmica das organizações envolvidas no desenvolvimento da Web. Quando a equipe da NCSA aumentou o ritmo de desenvolvimento da Web com uma equipe de desenvolvimento muito maior do que já esteve presente no CERN, a fonte libwww foi “bifurcada” (dividida em bases de código mantidas) para que o pessoal da NCSA não tivesse que esperar que o CERN acompanhar suas melhorias. Ao mesmo tempo, desenvolvedores independentes como eu mesmo comecei a desenvolver bibliotecas de protocolos para linguagens e plataformas ainda não suportadas pelo código CERN. O design da Web teve que mudar do desenvolvimento de um biblioteca de protocolos de referência para o desenvolvimento de uma API baseada em rede, estendendo a semântica desejada da Web em várias plataformas e implementações.

Uma API baseada em rede é uma sintaxe on-the-wire, com semântica definida, para aplicação interações. Uma API baseada em rede não impõe restrições ao código do aplicativo além da necessidade de ler/gravar na rede, mas impõe restrições ao conjunto de semântica que pode ser efetivamente comunicada através da interface. Do lado positivo, desempenho é limitado apenas pelo design do protocolo e não por qualquer implementação desse desenho.

Uma API baseada em biblioteca faz muito mais pelo programador, mas ao fazer isso cria um muito mais complexidade e bagagem do que é necessário para qualquer sistema, é menos portátil em uma rede heterogênea, e sempre resulta na preferência pela generalidade sobre atuação. Como efeito colateral, também leva a um desenvolvimento preguiçoso (culpando o código da API por tudo) e falha em explicar o comportamento não cooperativo de outras partes na comunicação.

No entanto, é importante ter em mente que existem várias camadas envolvidas em qualquer arquitetura, incluindo a da Web moderna. Sistemas como a Web usam uma API de biblioteca (sockets) para acessar várias APIs baseadas em rede (por exemplo, HTTP e FTP), mas o A própria API do socket está abaixo da camada de aplicação. Da mesma forma, libwww é um cruzamento interessante raça na medida em que evoluiu para uma API baseada em biblioteca para acessar uma API baseada em rede, e, portanto, fornece código reutilizável sem assumir que outros aplicativos de comunicação são usando libwww também.

Isso contrasta com middleware como CORBA [97]. Como CORBA só permite comunicação via ORB, seu protocolo de transferência, IIOP, assume muito sobre o que o partes estão se comunicando. As mensagens de solicitação HTTP incluem aplicativos padronizados semântica, enquanto as mensagens IIOP não. O token “Request” no IIOP apenas fornece direcionalidade para que o ORB possa roteá-lo de acordo com se o próprio ORB é suposto para responder (por exemplo, “LocateRequest”) ou se será interpretado por um objeto. A semântica é expressa pela combinação de uma chave de objeto e operação, que são específicas do objeto em vez de padronizados em todos os objetos.

Um desenvolvedor independente pode gerar os mesmos bits que uma solicitação IIOP sem usar o mesmo ORB, mas os próprios bits são definidos pela API CORBA e sua Interface Linguagem de Definição (IDL). Eles precisam de um UUID gerado por um compilador IDL, um conteúdo binário estruturado que espelha a assinatura dessa operação IDL e a definição de o(s) tipo(s) de dados de resposta de acordo com a especificação IDL. A semântica não é, portanto, definido pela interface de rede (IIOP), mas pela especificação IDL do objeto. Quer isto seja um coisa boa ou não depende da aplicação — para objetos distribuídos é uma necessidade, para a Web não é.

Por que isso é importante? Porque diferencia um sistema onde a rede intermediários podem ser agentes efetivos de um sistema onde podem ser, no máximo, roteadores.

Esse tipo de diferença também é visto na interpretação de uma mensagem como unidade ou como fluxo. O HTTP permite que o destinatário ou o remetente decidam por conta própria. CORBA IDL nem permite streams (ainda), mas mesmo quando é estendido para suportar streams, ambos os lados da comunicação estarão vinculados à mesma API, em vez de serem livres para usar o que for mais adequado ao seu tipo de aplicação.

### 6.5.2 HTTP não é RPC

As pessoas muitas vezes se referem erroneamente ao HTTP como um mecanismo de chamada de procedimento remoto (RPC) [23] simplesmente porque envolve pedidos e respostas. O que distingue o RPC de outros formas de comunicação de aplicativos baseados em rede é a noção de invocar um procedimento na máquina remota, onde o protocolo identifica o procedimento e lhe passa um conjunto de parâmetros e, em seguida, espera que a resposta seja fornecida em uma mensagem de retorno usando a mesma interface. A invocação de método remoto (RMI) é semelhante, exceto que o procedimento é identificado como uma tupla {objeto, método} em vez de um procedimento de serviço. O RMI intermediado adiciona indireção de serviço de nomes e alguns outros truques, mas a interface é basicamente o mesmo.

O que distingue HTTP de RPC não é a sintaxe. Não é mesmo o diferente características adquiridas com o uso de um fluxo como parâmetro, embora isso ajude a explicar por que os mecanismos RPC existentes não eram utilizáveis para a Web. O que torna o HTTP significativamente diferente do RPC é que as requisições são direcionadas aos recursos usando um interface genérica com semântica padrão que pode ser interpretada por intermediários quase

bem como pelas máquinas que originam os serviços. O resultado é um aplicativo que permite para camadas de transformação e indireção que são independentes da origem da informação, que é muito útil para uma escala de Internet, multi-organização, anarquicamente escalável sistema de informação. Os mecanismos RPC, em contraste, são definidos em termos de APIs de linguagem, não aplicativos baseados em rede.

### **6.5.3 HTTP não é um protocolo de transporte**

O HTTP não foi projetado para ser um protocolo de transporte. É um protocolo de transferência no qual o mensagens refletem a semântica da arquitetura da Web executando ações nos recursos através da transferência e manipulação de representações desses recursos. É possível para alcançar uma ampla gama de funcionalidades usando esta interface muito simples, mas seguindo as interface é necessária para que a semântica HTTP permaneça visível para os intermediários.

É por isso que o HTTP passa por firewalls. A maioria das extensões recentemente propostas para O HTTP, além do WebDAV [60], tem usado apenas o HTTP como forma de mover outros protocolos de aplicativos por meio de um firewall, o que é uma ideia fundamentalmente equivocada. Não apenas anula o propósito de ter um firewall, mas não funcionará a longo prazo porque os fornecedores de firewall simplesmente terão que realizar filtragem de protocolo adicional. Isto portanto, não faz sentido fazer essas extensões em cima do HTTP, já que a única coisa

O HTTP realiza nessa situação é adicionar sobrecarga de uma sintaxe herdada. Um verdadeiro aplicação de HTTP mapeia as ações do usuário do protocolo para algo que pode ser expresso usando semântica HTTP, criando assim uma API baseada em rede para serviços que podem ser entendido por agentes e intermediários sem qualquer conhecimento da aplicação.

#### 6.5.4 Projeto de Tipos de Mídia

Um aspecto do REST que é incomum para um estilo arquitetônico é o grau em que ele influencia a definição de elementos de dados dentro da arquitetura da Web.

##### 6.5.4.1 Estado do Aplicativo em um Sistema Baseado em Rede

REST define um modelo de comportamento esperado do aplicativo que suporta aplicações que são amplamente imunes às condições de falha parcial que afetam a maioria aplicativos baseados em rede. No entanto, isso não impede os desenvolvedores de aplicativos de introduzindo características que violam o modelo. As violações mais frequentes dizem respeito a as restrições no estado do aplicativo e interação sem estado.

As incompatibilidades de arquitetura devido ao estado do aplicativo mal colocado não estão limitadas ao HTTP biscoitos. A introdução de “frames” na Hypertext Markup Language (HTML) causou confusão semelhante. Os quadros permitem que uma janela do navegador seja particionada em subjanelas, cada um com seu próprio estado de navegação. As seleções de link em uma subjanela são indistinguível das transições normais, mas a representação da resposta resultante é renderizado dentro da subjanela em vez do espaço de trabalho completo do aplicativo do navegador. Isto é multa desde que nenhum link saia do domínio das informações destinadas à subjanela tratamento, mas quando isso ocorre, o usuário se vê vendo um aplicativo bloqueado dentro do subcontexto de outro aplicativo.

Tanto para frames quanto para cookies, a falha estava em fornecer um estado de aplicativo indireto que não pôde ser gerenciado ou interpretado pelo agente do usuário. Um desenho que colocou este informações dentro de uma representação primária, informando assim o agente do usuário sobre como gerenciar o espaço de trabalho hipermídia para um domínio específico de recursos, poderia ter

realizou as mesmas tarefas sem violar as restrições REST, ao mesmo tempo levando a um melhor interface de usuário e menos interferência com o cache.

#### *6.5.4.2 Processamento Incremental*

Ao incluir a redução de latência como um objetivo de arquitetura, o REST pode diferenciar a mídia tipos (o formato de dados das representações) de acordo com o desempenho percebido pelo usuário.

Tamanho, estrutura e capacidade de renderização incremental têm impacto na latência encontrou transferência, renderização e manipulação de tipos de mídia de representação, e assim, pode afetar significativamente o desempenho do sistema.

HTML [18] é um exemplo de tipo de mídia que, em sua maioria, tem boa latência características. As informações dentro do HTML antigo podiam ser renderizadas à medida que eram recebidas, porque todas as informações de renderização estavam disponíveis antecipadamente - dentro do padrão definições do pequeno conjunto de tags de marcação que compunham o HTML. No entanto, existem aspectos do HTML que não foram bem projetados para latência. Exemplos incluem: colocação de metadados incorporados no HEAD de um documento, resultando em informações opcionais precisam ser transferidos e processados antes que o mecanismo de renderização possa ler as partes que exibir algo útil para o usuário [93]; imagens incorporadas sem renderizar dicas de tamanho, exigindo que os primeiros bytes da imagem (a parte que contém o tamanho do layout) sejam recebido antes que o restante do HTML ao redor possa ser exibido; dimensionado dinamicamente colunas da tabela, exigindo que o renderizador leia e determine os tamanhos da tabela inteira antes que ele possa começar a exibir o topo; e, regras preguiçosas sobre a análise de dados malformados sintaxe de marcação, geralmente exigindo que o mecanismo de renderização analise um arquivo inteiro antes que ele possa determinar que um caractere de marcação de chave está ausente.

#### 6.5.4.3 Java versus JavaScript

REST também pode ser usado para entender por que alguns tipos de mídia tiveram maior adoção dentro da arquitetura da Web do que outros, mesmo quando o equilíbrio do desenvolvedor opinião não é a seu favor. O caso de applets Java versus JavaScript é um exemplo.

JavaTM [45] é uma linguagem de programação popular que foi originalmente desenvolvida para aplicações em set-top boxes de televisão, mas ganhou notoriedade quando foi introduzido na Web como um meio para implementar a funcionalidade *de código sob demanda*. Embora a linguagem tenha recebido um tremendo apoio da imprensa de seu proprietário, Sun Microsystems, Inc., e elogios de desenvolvedores de software que procuram uma alternativa à linguagem C++, ela não foi amplamente adotada pelos desenvolvedores de aplicativos para código sob demanda na Web.

Logo após a introdução do Java, os desenvolvedores da Netscape Communications Corporation criou uma linguagem separada para código sob demanda embutido, originalmente chamado LiveScript, mas depois mudou para o nome JavaScript por razões de marketing (os dois línguas têm relativamente pouco em comum além disso) [44]. Embora inicialmente ridicularizado por ser incorporado com HTML e ainda não compatível com a sintaxe HTML adequada, O uso de JavaScript tem aumentado constantemente desde a sua introdução.

A questão é: por que o JavaScript é mais bem-sucedido na Web do que o Java? Certamente não é por sua qualidade técnica como linguagem, pois tanto sua sintaxe quanto sua execução ambiente são considerados pobres quando comparados ao Java. Também não é por causa marketing: a Sun gastou muito mais do que a Netscape nesse aspecto, e continua a fazê-lo. Não é também por causa de quaisquer características intrínsecas das linguagens, já que Java tem sido mais bem sucedido do que JavaScript em todas as outras áreas de programação (aplicativos autônomos,

servlets, etc). Para entender melhor as razões dessa discrepância, precisamos avaliar Java em termos de suas características como um tipo de mídia de representação dentro do REST.

O JavaScript se ajusta melhor ao modelo de implantação da tecnologia da Web. tem muito mais baixo barreira de entrada, tanto em termos de sua complexidade geral como idioma quanto da quantidade de esforço inicial exigido por um programador iniciante para montar sua primeira peça de trabalho código. O JavaScript também tem menos impacto na visibilidade das interações. Independente organizações podem ler, verificar e copiar o código-fonte JavaScript da mesma forma que eles poderiam copiar HTML. Java, por outro lado, é baixado como arquivos empacotados binários — o usuário é, portanto, deixado para confiar nas restrições de segurança dentro da execução Java meio Ambiente. Da mesma forma, Java tem muito mais recursos que são considerados questionáveis para permitir dentro de um ambiente seguro, incluindo a capacidade de enviar solicitações RMI de volta ao servidor de origem. A RMI não oferece suporte à visibilidade para intermediários.

Talvez a distinção mais importante entre os dois, no entanto, seja que JavaScript causa menos latência percebida pelo usuário. O JavaScript geralmente é baixado como parte do representação primária, enquanto os applets Java requerem uma solicitação separada. código Java, uma vez convertido para o formato de código de byte, é muito maior do que o JavaScript típico. Finalmente, enquanto JavaScript pode ser executado enquanto o resto da página HTML está baixando, Java requer que o pacote completo de arquivos de classe seja baixado e instalado antes do aplicativo pode começar. Java, portanto, não suporta renderização incremental.

Uma vez que as características das línguas são dispostas na mesma linha das lógica por trás das restrições do REST, fica muito mais fácil avaliar as tecnologias em termos de seu comportamento dentro da arquitetura moderna da Web.

## 6.6 Resumo

Este capítulo descreveu as experiências e lições aprendidas com a aplicação do REST enquanto criação dos padrões da Internet para o Hypertext Transfer Protocol (HTTP) e Uniform Identificadores de recursos (URI). Essas duas especificações definem a interface genérica usada pelo todas as interações de componentes na Web. Além disso, descrevi as experiências e lições aprendidas com a implantação dessas tecnologias na forma do libwww-perl biblioteca cliente, o Apache HTTP Server Project e outras implementações do protocolo padrões.

## CONCLUSÕES

*Cada um de nós tem, em algum lugar do coração, o sonho de fazer um mundo vivo, um universo. Aqueles de nós que foram formados como arquitetos têm esse desejo talvez no centro de nossas vidas: que um dia, em algum lugar, de alguma forma, possamos construir um edifício que seja maravilhoso, bonito, de tirar o fôlego, um lugar onde as pessoas possam caminhar e sonhar durante séculos.*

— Christopher Alexander [3]

No início de nossos esforços dentro da Força-Tarefa de Engenharia da Internet para definir o Protocolo de Transferência de Hipertexto existente (HTTP/1.0) [19] e projetar as extensões para o novos padrões de HTTP/1.1 [42] e Uniform Resource Identifiers (URI) [21], I reconheceu a necessidade de um modelo de como a World Wide Web *deveria* funcionar. Este idealizado modelo das interações dentro de um aplicativo da Web geral, conhecido como O estilo arquitetônico Representational State Transfer (REST), tornou-se a base para o arquitetura moderna da Web, fornecendo os princípios orientadores pelos quais as falhas no a arquitetura preexistente pode ser identificada e as extensões validadas antes da implantação. REST é um conjunto coordenado de restrições arquitetônicas que tenta minimizar latência e comunicação de rede e, ao mesmo tempo, maximizar a independência e escalabilidade de implementações de componentes. Isto é conseguido através da imposição de restrições semântica do conector onde outros estilos se concentraram na semântica do componente. DESCANSO permite o armazenamento em cache e a reutilização de interações, substituibilidade dinâmica de componentes e processamento de ações por intermediários, atendendo assim às necessidades de um sistema hipermídia distribuído.

- As seguintes contribuições para o campo da Ciência da Informação e da Computação feito como parte desta dissertação:
- uma estrutura para entender a arquitetura de software por meio de estilos de arquitetura, incluindo um conjunto consistente de terminologia para descrever a arquitetura de software;
  - uma classificação de estilos de arquitetura para software de aplicativo baseado em rede por as propriedades arquitetônicas que induziriam quando aplicadas à arquitetura de um sistema hipermídia distribuído;
  - REST, um novo estilo de arquitetura para sistemas hipermídia distribuídos; e,
  - aplicação e avaliação do estilo arquitetural REST no projeto e implantação da arquitetura para a World Wide Web moderna.

A Web moderna é uma instância de uma arquitetura no estilo REST. Embora baseado na Web aplicativos podem incluir acesso a outros estilos de interação, o foco central de sua preocupações de protocolo e desempenho é hipermídia distribuída. REST elabora apenas aqueles partes da arquitetura que são consideradas essenciais para a distribuição em escala da Internet. interação hipermídia. Áreas para melhoria da arquitetura Web podem ser vistas onde protocolos existentes falham em expressar toda a semântica potencial para interação de componentes, e onde os detalhes da sintaxe podem ser substituídos por formas mais eficientes sem alterando os recursos da arquitetura. Da mesma forma, as extensões propostas podem ser comparadas com REST para ver se eles se encaixam na arquitetura; se não, é mais eficiente redirecionar isso funcionalidade para um sistema executado em paralelo com um estilo de arquitetura mais aplicável.

Em um mundo ideal, a implementação de um sistema de software corresponderia exatamente ao seu Projeto. Alguns recursos da arquitetura moderna da Web correspondem exatamente às suas critérios de design em REST, como o uso de URI [21] como identificadores de recursos e o uso de Tipos de mídia da Internet [48] para identificar formatos de dados de representação. No entanto, também existem alguns aspectos dos protocolos Web modernos que existem apesar do design arquitetônico,

devido a experimentos legados que falharam (mas devem ser mantidos para compatibilidade com versões anteriores) e extensões implantadas por desenvolvedores que desconhecem o estilo de arquitetura. REST fornece um modelo não só para o desenvolvimento e avaliação de novas funcionalidades, mas também para a identificação e compreensão de recursos quebrados.

A World Wide Web é indiscutivelmente o maior aplicativo distribuído do mundo. Compreender os principais princípios de arquitetura subjacentes à Web pode ajudar a explicar seu sucesso técnico e pode levar a melhorias em outros aplicativos distribuídos, particularmente aqueles que são passíveis de métodos de interação iguais ou semelhantes. DESCANSO contribui tanto com a lógica por trás da arquitetura de software da Web moderna quanto com uma lição significativa sobre como os princípios de engenharia de software podem ser sistematicamente aplicados em o projeto e avaliação de um sistema de software real.

Para aplicativos baseados em rede, o desempenho do sistema é dominado pela rede comunicação. Para um sistema de hipermídia distribuído, as interações dos componentes consistem em transferências de dados de grande granularidade em vez de tarefas de computação intensiva. O estilo REST foi desenvolvido em resposta a essas necessidades. Seu foco na interface do conector genérico de recursos e representações permitiram processamento intermediário, armazenamento em cache e substituibilidade de componentes, o que, por sua vez, permitiu que aplicativos baseados na Web fossem dimensionados de 100.000 solicitações/dia em 1994 para 600.000.000 solicitações/dia em 1999.

O estilo arquitetural REST foi validado ao longo de seis anos de desenvolvimento de os padrões HTTP/1.0 [19] e HTTP/1.1 [42], elaboração do URI [21] e relativo padrões URL [40] e implantação bem-sucedida de várias dezenas de forma independente sistemas de software de nível comercial desenvolvidos dentro da arquitetura moderna da Web. Isto

serviu tanto como um modelo para orientação de projeto e como um teste ácido para arquitetura extensões para os protocolos da Web.

O trabalho futuro se concentrará em estender a orientação arquitetônica para o desenvolvimento de um substituto para a família de protocolos HTTP/1.x, usando um sintaxe tokenizada, mas sem perder as propriedades desejáveis identificadas pelo REST. o necessidades de dispositivos sem fio, que têm muitas características em comum com os princípios por trás do REST, motivará melhorias adicionais para o design de protocolos em nível de aplicativo e arquiteturas envolvendo intermediários ativos. Houve também algum interesse em estendendo o REST para considerar prioridades de solicitação variáveis, qualidade de serviço diferenciada, e representações que consistem em fluxos de dados contínuos, como aqueles gerados por transmitir fontes de áudio e vídeo.

## REFERÊNCIAS

1. GD Abowd, R. Allen e D. Garlan. Estilo de formalização para entender as descrições da arquitetura de software. *ACM Transactions on Software Engineering and Methodology*, 4(4), outubro de 1995, pp. 319–364. Uma versão mais curta também apareceu como: Usando estilo para entender as descrições da arquitetura de software. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'93)*, Los Angeles, CA, dezembro de 1993, pp. 9–20.
2. *Manual de referência da linguagem PostScript* da Adobe Systems Inc. Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.
3. C. Alexandre. *A forma intemporal de construir*. Oxford University Press, Nova York, 1979.
4. C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King e S. Angel. *Uma Linguagem Padrão*. Oxford University Press, Nova York, 1977.
5. R. Allen e D. Garlan. Uma base formal para conexão arquitetônica. *ACM Transactions on Software Engineering and Methodology*, 6(3), julho de 1997. Uma versão mais curta também apareceu como: Formalizing architecture connection. In *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Itália, maio de 1994, pp. 71–80. Também como: Além da Definição/Uso: Interconexão Arquitetural. In *Proceedings of the ACM Interface Definition Language Workshop*, Portland, Oregon, *SIGPLAN Notices*, 29(8), agosto de 1994.
6. G. Andrews. Paradigmas para interação de processos em programas distribuídos. *ACM Computing Surveys*, 23(1), março de 1991, pp. 49–90.
7. F. Anklesaria, et al. O protocolo Internet Gopher (um protocolo distribuído de busca e recuperação de documentos). *Internet RFC 1436*, março de 1993.
8. DJ Barrett, LA Clarke, PL Tarr, AE Wise. Uma estrutura para integração de software baseada em eventos. *ACM Transactions on Software Engineering and Methodology*, 5(4), outubro de 1996, pp. 378–421.
9. L. Bass, P. Clements e R. Kazman. *Arquitetura de Software na Prática*. Addison Wesley, Reading, Massachusetts, 1998.
10. D. Batory, L. Coglianese, S. Shafer e W. Tracz. A arquitetura de referência de aviônicos ADAGE. In *Proceedings of AIAA Computing in Aerospace 10*, San Antonio, 1995.

11. T. Berners-Lee, R. Cailliau e J.-F. Groff. Rede mundial de computadores. Folheto distribuído na *3rd Joint European Networking Conference*, Innsbruck, Áustria, maio de 1992.
12. T. Berners-Lee, R. Cailliau, J.-F. Groff e B. Pollermann. World-Wide Web: O universo da informação. *Electronic Networking: Research, Applications and Policy*, 2(1), Meckler Publishing, Westport, CT, Primavera de 1992, pp. 52–58.
13. T. Berners-Lee e R. Cailliau. Rede mundial de computadores. In *Proceedings of Computing in High Energy Physics 92*, Annecy, França, 23-27 de setembro de 1992.
14. T. Berners-Lee, R. Cailliau, C. Barker e J.-F. Groff. Projeto W3: Notas de projeto variadas. Publicado na Web, novembro de 1992. Arquivado em <<http://www.w3.org/History/19921103-hypertext/hypertext/WWW/WorkingNotes/Overview.html>>, setembro de 2000.
15. T. Berners-Lee. Identificadores de recursos universais na WWW. *Internet RFC 1630*, junho de 1994.
16. T. Berners-Lee, R. Cailliau, A. Luotonen, H. Frystyk Nielsen e A. Secret. A World Wide Web. *Comunicações da ACM*, 37(8), agosto de 1994, pp. 76–82.
17. T. Berners-Lee, L. Masinter e M. McCahill. Localizadores uniformes de recursos (URL). *Internet RFC 1738*, dezembro de 1994.
18. T. Berners-Lee e D. Connolly. Linguagem de marcação de hipertexto — 2.0. *Internet RFC 1866*, novembro de 1995.
19. T. Berners-Lee, RT Fielding e HF Nielsen. Protocolo de Transferência de Hipertexto - HTTP/1.0. *Internet RFC 1945*, maio de 1996.
20. T. Berners-Lee. WWW: Passado, presente e futuro. *IEEE Computer*, 29(10), outubro de 1996, pp. 69-77.
21. T. Berners-Lee, RT Fielding e L. Masinter. Uniform Resource Identifiers (URI): Sintaxe genérica. *Internet RFC 2396*, agosto de 1998.
22. P. Bernstein. Middleware: Um modelo para serviços de sistemas distribuídos. *Comunicações da ACM*, fevereiro de 1996, pp. 86-98.
23. AD Birrell e BJ Nelson. Implementando chamada de procedimento remoto. *ACM Transactions on Computer Systems*, 2 de janeiro de 1984, pp. 39-59.
24. M. Boasson. A arte da arquitetura de software. *Software IEEE*, 12(6), novembro de 1995, pp. 13–16.

25. G. Booch. Desenvolvimento orientado a objetos. *IEEE Transactions on Software Engineering*, 12(2), fevereiro de 1986, pp. 211–221.
26. C. Brooks, MS Mazer, S. Meeks e J. Miller. Servidores proxy específicos do aplicativo como transdutores de fluxo HTTP. In *Proceedings of the Fourth International World Wide Web Conference*, Boston, Massachusetts, dezembro de 1995, pp. 539–548.
27. F. Buschmann e R. Meunier. Um sistema de padrões. Coplien e Schmidt (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, pp. 325-343.
28. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad e M. Stal.  
*Arquitetura de Software Orientada a Padrões: Um sistema de padrões*. John Wiley & Sons Ltd., Inglaterra, 1996.
29. MR Cagan. O ambiente HP SoftBench: Uma arquitetura para uma nova geração de ferramentas de software. *Hewlett-Packard Journal*, 41(3), junho de 1990, pp. 36-47.
30. J.R. Cameron. Uma visão geral do JSD. *IEEE Transactions on Software Engineering*, 12(2), fevereiro de 1986, pp. 222–240.
31. RS Chin e ST Chanson. Sistemas de programação baseados em objetos distribuídos. *ACM Computing Surveys*, 23(1), março de 1991, pp. 91–124.
32. DD Clark e DL Tennenhouse. Considerações arquitetônicas para uma nova geração de protocolos. In *Proceedings of ACM SIGCOMM'90 Symposium*, Filadélfia, PA, setembro de 1990, pp. 200-208.
33. JO Coplien e DC Schmidt, ed. *Linguagens de Padrões de Design de Programas*. Addison-Wesley, Reading, Massachusetts, 1995.
34. JO Coplien. Idiomas e padrões como literatura arquitetônica. *Software IEEE*, 14(1), janeiro de 1997, pp. 36–42.
35. EM Dashofy, N. Medvidovic, RN Taylor. Usando middleware de prateleira para implementar conectores em arquiteturas de software distribuídas. In *Proceedings of the 1999 International Conference on Software Engineering*, Los Angeles, 16–22 de maio de 1999, pp. 3–12.
36. F. Davis, et. al. *Especificação funcional do protótipo do protocolo de interface WAIS (v. 1.5)*. Thinking Machines Corporation, abril de 1990.
37. F. DeRemer e HH Kron. Programação-no-grande versus programação-no-pequeno. *IEEE Transactions on Software Engineering*, SE-2(2), junho de 1976, pp. 80–86.

38. E. Di Nitto e D. Rosenblum. Explorando ADLs para especificar estilos arquitetônicos induzidos por infraestruturas de middleware. In *Proceedings of the 1999 International Conference on Software Engineering*, Los Angeles, 16–22 de maio de 1999, pp. 13–22.
39. Campo RT. Mantendo infoestruturas de hipertexto distribuídas: Bem-vindo à web do MOMspider. *Computer Networks and ISDN Systems*, 27(2), novembro de 1994, pp. 193–204.
40. Campo RT. Localizadores de recursos uniformes relativos. *Internet RFC 1808*, junho de 1995.
41. RT Fielding, EJ Whitehead, Jr., KM Anderson, G. Bolcer, P. Oreizy e RN Taylor. Desenvolvimento baseado na Web de produtos de informação complexos. *Comunicações da ACM*, 41(8), agosto de 1998, pp. 84–92.
42. RT Fielding, J. Gettys, JC Mogul, HF Nielsen, L. Masinter, P. Leach e T. Berners-Lee. Protocolo de transferência de hipertexto — HTTP/1.1. *Internet RFC 2616*, junho de 1999. [Obsoletos RFC 2068, janeiro de 1997.]
43. RT Fielding e RN Taylor. Design baseado em princípios da arquitetura moderna da Web. In *Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000)*, Limerick, Irlanda, junho de 2000, pp. 407–416.
44. D. Flanagan. *JavaScript: O Guia Definitivo*, 3<sup>a</sup> edição. O'Reilly & Associates, Sebastopol, CA, 1998.
45. D. Flanagan. *JavaTM em poucas palavras*, 3<sup>a</sup> edição. O'Reilly & Associates, Sebastopol, CA, 1999.
46. J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, E. Sink e L. Stewart. Autenticação HTTP: Autenticação de Acesso Básico e Digest. *Internet RFC 2617*, junho de 1999.
47. N. Freed e N. Borenstein. MIME (Multipurpose Internet Mail Extensions) Parte Um: Formato dos Corpos de Mensagens da Internet. *Internet RFC 2045*, novembro de 1996.
48. N. Freed, J. Klensin e J. Postel. Multipurpose Internet Mail Extensions (MIME) Parte Quatro: Procedimentos de Registro. *Internet RFC 2048*, novembro de 1996.
49. M. Fridrich e W. Older. Helix: A arquitetura do arquivo distribuído XMS sistema. *Software IEEE*, 2, maio de 1985, pp. 21–29.
50. A. Fuggetta, GP Picco e G. Vigna. Entendendo a mobilidade do código. *IEEE Transactions on Software Engineering*, 24(5), maio de 1998, pp. 342–361.

51. E. Gamma, R. Helm, R. Johnson e J. Vlissides. *Padrões de Projeto: Elementos de Software Reutilizável Orientado a Objetos*. Addison-Wesley, Reading, Massachusetts, 1995.
52. D. Garlan e E. Ilias. Políticas de integração de ferramentas adaptáveis e de baixo custo para ambientes integrados. In *Proceedings of the ACM SIGSOFT '90: Quarto Simpósio sobre Ambientes de Desenvolvimento de Software*, dezembro de 1990, pp. 1–10.
53. D. Garlan e M. Shaw. Uma introdução à arquitetura de software. Ambriola & Tortola (eds.), *Advances in Software Engineering & Knowledge Engineering*, vol. II, World Scientific Pub Co., Cingapura, 1993, pp. 1–39.
54. D. Garlan, R. Allen e J. Ockerbloom. Explorando o estilo em ambientes de projeto arquitetônico. In *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'94)*, Nova Orleans, dezembro de 1994, pp. 175–188.
55. D. Garlan e DE Perry. Introdução à edição especial sobre arquitetura de software. *IEEE Transactions on Software Engineering*, 21(4), abril de 1995, pp. 269–274.
56. D. Garlan, R. Allen e J. Ockerbloom. Incompatibilidade arquitetônica, ou Por que é difícil construir sistemas a partir de peças existentes. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, WA, 1995. Também aparece como: Incompatibilidade de arquitetura: Por que a reutilização é tão difícil. *Software IEEE*, 12(6), novembro de 1995, pp. 17–26.
57. D. Garlan, R. Monroe e D. Wile. ACME: Uma linguagem de descrição de arquitetura. Em *Anais do CASCON'97*, novembro de 1997.
58. C. Ghezzi, M. Jazayeri e D. Mandrioli. *Fundamentos de Engenharia de Software*. Prentice-Hall, 1991.
59. S. Glassman. Um relé de cache para a World Wide Web. *Computer Networks and ISDN Systems*, 27(2), novembro de 1994, pp. 165–173.
60. Y. Goland, EJ Whitehead, Jr., A. Faizi, S. Carter e D. Jensen. Extensões HTTP para autoria distribuída — WEBDAV. *Internet RFC 2518*, fevereiro de 1999.
61. K. Grønbæk e RH Trigg. Problemas de design para um sistema de hipermídia baseado em Dexter. *Comunicações da ACM*, 37(2), fevereiro de 1994, pp. 41–49.
62. B. Hayes-Roth, K. Pfleger, P. Lalanda, P. Morigot e M. Balabanovic. Uma arquitetura de software específica de domínio para sistemas inteligentes adaptativos. *IEEE Transactions on Software Engineering*, 21(4), abril de 1995, pp. 288–301.

63. J. Heidemann, K. Obraczka e J. Touch. Modelando o desempenho do HTTP em vários protocolos de transporte. *IEEE/ACM Transactions on Networking*, 5(5), outubro de 1997, pp. 616–630.
64. K. Holtman e A. Mutz. Negociação de conteúdo transparente em HTTP. *Internet RFC* 2295, março de 1998.
65. P. Inverardi e AL Wolf. Especificação formal e análise de arquiteturas de software usando o modelo de máquina abstrata química. *IEEE Transactions on Software Engineering*, 21(4), abril de 1995, pp. 373–386.
66. ISO/IEC JTC1/SC21/WG7. *Modelo de Referência de Processamento Distribuído Aberto*. ITU-T X.901: ISO/IEC 10746-1, 07 de junho de 1995.
67. M. Jackson. Problemas, métodos e especialização. *IEEE Software*, 11(6), [condensado de Software Engineering Journal], novembro de 1994. pp. 57–62.
68. R. Kazman, L. Bass, G. Abowd e M. Webb. SAAM: Um método para analisar as propriedades de arquiteturas de software. In *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Itália, maio de 1994, pp. 81–90.
69. R. Kazman, M. Barbacci, M. Klein, SJ Carrière e SG Woods. Experiência com execução de análise de tradeoff de arquitetura. In *Proceedings of the 1999 International Conference on Software Engineering*, Los Angeles, 16–22 de maio de 1999, pp. 54–63.
70. NL Kerth e W. Cunningham. Usando padrões para melhorar nossa visão arquitetônica. *Software IEEE*, 14(1), janeiro de 1997, pp. 53–59.
71. R. Khare e S. Lawrence. Atualizando para TLS em HTTP/1.1. *Internet RFC* 2817, maio de 2000.
72. GE Krasner e ST Pope. Um livro de receitas para usar o paradigma de interface de usuário Model-View-Controller em Smalltalk-80. *Journal of Object Oriented Programming*, 1(3), agosto-setembro. 1988, pp. 26-49.
73. D. Kristol e L. Montulli. Mecanismo de gerenciamento de estado HTTP. *Internet RFC* 2109, fevereiro de 1997.
74. PB Kruchten. O modelo de arquitetura 4+1 View. *Software IEEE*, 12(6), novembro de 1995, pp. 42–50.
75. D. Le Métayer. Descrevendo estilos de arquitetura de software usando gramáticas de grafos. *IEEE Transactions on Software Engineering*, 24(7), julho de 1998, pp. 521–533.

76. WC Loerke. Sobre o estilo na arquitetura. F. Wilson, *Arquitetura: Questões Fundamentais*, Van Nostrand Reinhold, Nova York, 1990, pp. 203-218.
77. DC Luckham, JJ Kenney, LM Augustin, J. Vera, D. Bryan e W. Mann. Especificação e análise de arquitetura de sistemas utilizando Rapide. *IEEE Transactions on Software Engineering*, 21(4), abril de 1995, pp. 336–355.
78. DC Luckham e J. Vera. Uma linguagem de definição de arquitetura baseada em eventos. *IEEE Transactions on Software Engineering*, 21(9), set. 1995, pp. 717–734.
79. A. Luotonen e K. Altis. Proxies da World Wide Web. *Computer Networks and ISDN Systems*, 27(2), novembro de 1994, pp. 147–154.
80. P. Maes. Conceitos e experiências em reflexão computacional. In *Proceedings of OOPSLA '87*, Orlando, Flórida, outubro de 1987, pp. 147–155.
81. J. Magee, N. Dulay, S. Eisenbach e J. Kramer. Especificando arquiteturas de software distribuído. In *Proceedings of the 5th European Software Engineering Conference (ESEC'95)*, Sitges, Espanha, setembro de 1995, pp. 137–153.
82. J. Magee e J. Kramer. Estrutura dinâmica em arquiteturas de software. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'96)*, San Francisco, outubro de 1996, pp. 3–14.
83. F. Manola. Tecnologias para um modelo de objeto da Web. *IEEE Internet Computing*, 3(1), Jan.–Fev. 1999, pp. 38-47.
84. H. Maurer. *HyperWave: A Solução Web de Próxima Geração*. Addison-Wesley, Harlow, Inglaterra, 1996.
85. MJ Maybee, DH Heimbigner e LJ Osterweil. Multi Idiomas interoperabilidade em sistemas distribuídos: relato de experiência. In *Proceedings 18th International Conference on Software Engineering*, Berlim, Alemanha, março de 1996.
86. N. Medvidovic e RN Taylor. Um framework para classificar e comparar linguagens de descrição de arquitetura. In *Proceedings of the 6th European Software Engineering Conference realizado em conjunto com o 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurique, Suíça, setembro de 1997, pp. 60–76.
87. N. Medvidovic. *Evolução de software em tempo de especificação baseada em arquitetura*. Ph.D. Dissertação, Universidade da Califórnia, Irvine, dezembro de 1998.
88. N. Medvidovic, DS Rosenblum e RN Taylor. Uma linguagem e ambiente para desenvolvimento e evolução de software baseado em arquitetura. Em *Atos do*

- 1999 *Conferência Internacional de Engenharia de Software*, Los Angeles, 16–22 de maio de 1999, pp. 44–53.
89. A. Mockus, RT Fielding e J. Herbsleb. Um estudo de caso de desenvolvimento de software de código aberto: O servidor Apache. In *Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000)*, Limerick, Irlanda, junho de 2000, pp. 263–272.
90. J. Mogul, R. Fielding, J. Gettys e H. Frystyk. Uso e interpretação de números de versão HTTP. *Internet RFC 2145*, maio de 1997.
91. RT Monroe, A. Kompanek, R. Melton e D. Garlan. Estilos arquitetônicos, padrões de design e objetos. *Software IEEE*, 14(1), janeiro de 1997, pp. 43–52.
92. M. Moriconi, X. Qian e RA Riemenscheider. Refinamento correto da arquitetura. *IEEE Transactions on Software Engineering*, 21(4), abril de 1995, pp. 356–372.
93. HF Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. Lie e C. Lilley. Efeitos de desempenho de rede de HTTP/1.1, CSS1 e PNG. *Anais da ACM SIGCOMM '97*, Cannes, França, setembro de 1997.
94. HF Nielsen, P. Leach e S. Lawrence. Estrutura de extensão HTTP, *Internet RFC 2774*, fevereiro de 2000.
95. H. Penny Nii. Sistemas de quadro-negro. *AI Magazine*, 7(3):38-53 e 7(4):82-107, 1986.
96. Grupo de gerenciamento de objetos. *Guia de Arquitetura de Gerenciamento de Objetos, Rev. 3.0*. Soley & Stone (eds.), Nova York: J. Wiley, 3<sup>a</sup> ed., 1995.
97. Grupo de gerenciamento de objetos. *O Common Object Request Broker: Arquitetura e Especificação (CORBA 2.1)*. <<http://www.omg.org/>>, agosto de 1997.
98. P. Oreizy, N. Medvidovic e RN Taylor. Evolução do software de tempo de execução baseado em arquitetura. In *Proceedings of the 1998 International Conference on Software Engineering*, Kyoto, Japão, abril de 1998.
99. P. Oreizy. Evolução de software descentralizada. Manuscrito não publicado (Phase II Survey Paper), dezembro de 1998.
100. VN Padmanabhan e JC Mogul. Melhorando a latência HTTP. *Computer Networks and ISDN Systems*, 28, dezembro de 1995, pp. 25–35.
101. DL Parnas. Aspectos de distribuição de informação da metodologia de projeto. In *Proceedings of IFIP Congress 71*, Ljubljana, agosto de 1971, pp. 339–344.

102. DL Parnas. Sobre os critérios a serem utilizados na decomposição de sistemas em módulos. *Comunicações da ACM*, 15(12), dezembro de 1972, pp. 1053–1058.
103. DL Parnas. Projetando software para facilidade de extensão e contração. *Transações IEEE em Engenharia de Software*, SE-5(3), março de 1979.
104. DL Parnas, PC Clements e DM Weiss. A estrutura modular do complexo sistemas. *Transações IEEE em Engenharia de Software*, SE-11(3), 1985, pp. 259–266.
105. DE Perry e AL Wolf. Fundamentos para o estudo da arquitetura de software. *ACM SIGSOFT Software Engineering Notes*, 17(4), outubro de 1992, pp. 40–52.
106. J. Postel e J. Reynolds. Especificação do protocolo TELNET. *Internet STD 8, RFC 854*, maio de 1983.
107. J. Postel e J. Reynolds. Protocolo de transferência de arquivos. *Internet STD 9, RFC 959*, Outubro de 1985.
108. D. Pountain e C. Szyperski. Sistemas de software extensíveis. *Byte*, maio de 1994, págs. 57-62.
109. R. Prieto-Diaz e JM Vizinhos. Linguagens de interconexão de módulos. *Diário de Systems and Software*, 6(4), novembro de 1986, pp. 307-334.
110. JM Putilo. O barramento de software Polylith. *ACM Transactions on Programming Languages and Systems*, 16(1), Jan. 1994, pp. 151–174.
111. M. Python. O esboço dos arquitetos. *Programa de TV Flying Circus de Monty Python, Episódio 17*, setembro de 1970. Transcrição em <<http://www.stone-dead.asn.au/sketches/architec.htm>>.
112. J. Rasure e M. Young. Ambiente aberto para processamento de imagens e desenvolvimento de software. In *Proceedings of the 1992 SPIE/IS&T Symposium on Electronic Imaging*, Vol. 1659, fevereiro de 1992.
113. SP Reiss. Ferramentas de conexão usando passagem de mensagens no ambiente Field. *IEEE Software*, 7(4), julho de 1990, pp. 57–67.
114. DS Rosenblum e AL Wolf. Uma estrutura de design para observação e notificação de eventos em escala da Internet. In *Proceedings of the 6th European Software Engineering Conference realizado em conjunto com o 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurique, Suíça, setembro de 1997, pp. 344–360.

115. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh e B. Lyon. Projeto e implementação do sistema de arquivos de rede da Sun. In *Proceedings of the Usenix Conference*, junho de 1985, pp. 119–130.
116. M. Shapiro. Estrutura e encapsulamento em sistemas distribuídos: O proxy princípio. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, Cambridge, MA, maio de 1986, pp. 198–204.
117. M. Shaw. Para abstrações de alto nível para sistemas de software. *Data & Knowledge Engineering*, 5, 1990, pp. 119-128.
118. M. Shaw, R. DeLine, DV Klein, TL Ross, DM Young e G. Zelesnick. Abstrações para arquitetura de software e ferramentas para apoiá-las. *IEEE Transactions on Software Engineering*, 21(4), abril de 1995, pp. 314–335.
119. M. Shaw. Comparando estilos de projeto arquitetônico. *Software IEEE*, 12(6), novembro de 1995, pp. 27–41.
120. M. Shaw. Alguns padrões para arquitetura de software. Vlissides, Coplien & Kerth (eds.), *Pattern Languages of Program Design, Vol. 2*, Addison-Wesley, 1996, pp. 255-269.
121. M. Shaw e D. Garlan. *Arquitetura de Software: Perspectivas de uma Empresa Emergente Disciplina*. Prentice-Hall, 1996.
122. M. Shaw e P. Clements. Um guia de campo para boxology: classificação preliminar de estilos de arquitetura para sistemas de software. In *Proceedings of the Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, Washington, DC, agosto de 1997, pp. 6–13.
123. A. Sinhá. Computação cliente-servidor. *Comunicações da ACM*, 35(7), julho de 1992, pp. 77-98.
124. K. Sollins e L. Masinter. Requisitos funcionais para nomes de recursos uniformes. *Internet RFC 1737*, dezembro de 1994.
125. SE Spero. Análise de problemas de desempenho HTTP. Publicado na Web, <<http://metalab.unc.edu/mdma-release/http-prob.html>>, 1994.
126. KJ Sullivan e D. Notkin. Conciliando integração de ambiente e evolução de software. *ACM Transactions on Software Engineering and Methodology*, 1(3), julho de 1992, pp. 229–268.
127. AS Tanenbaum e R. van Renesse. Sistemas operacionais distribuídos. *ACM Computing Surveys*, 17(4), dezembro de 1985, pp. 419–470.

128. RN Taylor, N. Medvidovic, KM Anderson, EJ Whitehead Jr., JE Robbins, KA Nies, P. Oreizy e DL Dubrow. Um estilo de arquitetura baseado em componentes e mensagens para software GUI. *IEEE Transactions on Software Engineering*, 22(6), junho de 1996, pp. 390–406.
129. W. Tepenhart e JJ Cusick. Uma topologia de objeto unificada. *Software IEEE*, 14(1), janeiro de 1997, pp. 31–35.
130. W. Tracz. Exemplo pedagógico de DSSA (arquitetura de software específica de domínio). *Notas de Engenharia de Software*, 20(3), julho de 1995, pp. 49–62.
131. A. Umar. *Ambientes de Internet Cliente/Servidor Orientados a Objetos*. Prentice Hall PT, 1997.
132. S. Vestal. Manual do programador MetaH, versão 1.09. *Relatório Técnico*, Honeywell Technology Center, abril de 1996.
133. J. Waldo, G. Wyant, A. Wollrath e S. Kendall. Uma nota sobre a distribuição Informática. *Relatório Técnico SMLI TR-94-29*, Sun Microsystems Laboratories, Inc., novembro de 1994.
134. L. Wall, T. Christiansen e RL Schwartz. *Programação Perl*, 2<sup>a</sup> ed. O'Reilly & Associates, 1996.
135. EJ Whitehead, Jr., RT Fielding e KM Anderson. Fusão da tecnologia WWW e do servidor de link: uma abordagem. In *Proceedings of the 2nd Workshop on Open Hypermedia Systems, Hypertext'96*, Washington, DC, março de 1996, pp. 81-86.
136. A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin e H. Levy. Análise baseada em organização de compartilhamento e armazenamento em cache de objetos da Web. In *Proceedings of the 2nd USENIX Conference on Internet Technologies and Systems (USITS)*, outubro de 1999.
137. W. Zimmer. Relações entre padrões de projeto. Coplien e Schmidt (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, pp. 345-364.
138. H. Zimmerman. Modelo de referência OSI — O modelo ISO de arquitetura para interconexão de sistemas abertos. *IEEE Transactions on Communications*, 28 de abril de 1980, pp. 425–432.