2
Trabalhos Relacionados

Neste capítulo procuramos resumir alguns trabalhos que apresentam abordagens para auxiliar o desenvolvedor na análise de aplicações distribuídas, através da compreensão do seu comportamento em tempo de execução e na identificação de falhas. Nesses trabalhos apresentados destaca-se a necessidade de um mecanismo de instrumentação da aplicação para coletar dados em tempo de execução, que vão alimentar as ferramentas propostas.

O trabalho de Moe e Carr [11] descreve um método para utilizar os dados de operação para aprimorar a compreensão de um sistema distribuído. O objetivo é apresentar os dados coletados de forma que o analista possa melhor e mais facilmente entender o comportamento do sistema em execução, principalmente quando o analista não é o desenvolvedor do sistema.

O método utilizado captura a comunicação cliente/servidor utilizando o mecanismo de interceptadores CORBA [10], e envia os dados para um servidor central de *logs* visando minimizar o impacto no desempenho. A visualização é feita pela ferramenta SpotFire.net, que permitiu a construção rápida de um gráfico em duas e três dimensões a partir dos dados obtidos.

Weinreich e Kurshl [12] apresentam um ambiente de monitoramento utilizando a biblioteca de comunicação *ObjectWire*, com o objetivo de oferecer suporte à análise dinâmica de aplicações distribuídas orientadas a objetos. O ambiente oferece uma arquitetura adaptável e extensível, tanto para a etapa de aquisição de dados quanto para a etapa de visualização. A arquitetura permite a adesão dinâmica de diferentes ferramentas de análise, suporta eventos definidos pelo usuário e a observação de aspectos específicos da arquitetura cliente/servidor, focando na semântica de comunicação, na frequência e no volume de comunicação de dados e no gerenciamento de erro. Embora a arquitetura não apresente grande impacto no desempenho, essa não é uma prioridade dos autores.

Segundo o trabalho de Weinreich e Kurshl [12], a análise do fluxo de comunicação pode auxiliar a identificação de decisões errôneas de projeto. Como exemplo, um grande número de mensagens recebidas mas não processadas pode indicar sobrecarga de algum nó da aplicação ou de um serviço, ou ainda, ali-

ada à informação sobre o tempo de processamento das mensagens, pode ajudar a determinar a melhor estratégia de processamento para um servidor. Além disso, as informações sobre a frequência e o volume de transmissão de dados podem ajudar a determinar de que forma mapear os diversos componentes da aplicação nos recursos de hardware disponíveis. Componentes que se comunicam com grande frequência, ou que trocam grandes volumes de dados, podem ser alocados no mesmo nó físico ou até mesmo dentro do mesmo processo. Também é possível, através do monitoramento de exceções lançadas, detectar pontos de grande ocorrência de exceções que, mesmo que corretamente tratadas, podem sinalizar a necessidade de mudanças no projeto de uma aplicação.

O *framework* de monitoramento utilizado por Weinreich e Kurshl gera eventos que são enviados para as ferramentas de visualização e análise semântica por meio de plug-ins de distribuição de eventos (EDP), que podem ser dinamicamente conectados à aplicação. Um EDP simples apenas encaminha os eventos, enquanto outros, mais complexos, podem filtrar os eventos, combinando eventos em eventos de mais alto nível, não só reduzindo o número de eventos encaminhados, mas também permitindo um maior nível de abstração na observação de aspectos relacionados à semântica de comunicação. Por exemplo, os eventos de enviar e receber mensagem podem ser combinados em um evento transmissão bem sucedida, o que também ajuda a reduzir o número de eventos transmitidos para a ferramenta de análise.

Com esse desacoplamento, o ambiente de monitoramento pode ser configurado para cada componente. Dependendo do aspecto a ser monitorado, diferentes EDPs podem ser usados para preprocessar os eventos gerados antes de enviá-los para uma unidade de notificação de eventos, onde serão condensados e semanticamente enriquecidos. Por sua vez, a unidade de notificação de eventos se comunica, usando o padrão *Observer*, com as unidades de processamento de eventos que vão apresentar as informações de maneiras diferentes para atender requisitos específicos da análise. Cada unidade de processamento de eventos registra os eventos na qual está interessada junto à unidade de notificação apropriada que, quando receber o evento do EDP, vai encaminhá-lo àquela unidade de processamento que registrou interesse no evento.

Ju Li [13] propõe um mecanismo para manter e correlacionar as informações globais de causalidade em aplicações baseada em componentes, e usá-las para caracterizar as cadeias de chamadas de funções e seus respectivos comportamentos. A abordagem utilizada se baseia na construção de um túnel virtual que permite que a informação de causalidade possa ser propagada através das fronteiras das *threads*, processos e processadores. Esse túnel é construído por meio de código de instrumentação injetado nos *stubs* e *skeletons*. A

idéia básica é fazer com que um identificador único global se propague pelo sistema através do túnel encadeando as funções subsequentes, de *thread* para *thread*, de processo para processo e de processador para processador, estabelecendo a ligação causal de uma coleção de *threads* em tempo de execução.

O túnel virtual consiste de um canal privado entre *stubs* e *skeletons* e do transporte de dados usando variáveis específicas de linha de execução (*Thread specific storage*). Os dados necessários para determinar a estrutura hierárquica das funções chamadas são transferidos em uma estrutura de dados que é incluída como um parâmetro adicional na interface da função. Essa inclusão é totalmente transparente para a aplicação já que é feita pelos *stubs* e *skeletons*. Esse trabalho não oferece suporte à DII (Dynamic Invocation Interface) ou DSI (Dynamic Skeleton Interface).

Os trabalhos apresentados abordam diferentes técnicas para a análise dos traços de execução, propondo ferramentas que vão auxiliar o analista na compreensão do comportamento do sistema e na depuração de problemas.

No trabalho de Moe e Carr [11] o analista pode ter um melhor entendimento do sistema correlacionando as informações por meio de gráficos. Nesse trabalho destaca-se a divisão do método empregado em três fases: captura dos traços de execução, processamento dos dados coletados e apresentação visual dos dados. Está divisão também é usada na arquitetura de nossa ferramenta.

O trabalho de Weinreich e Kurshl [12] também busca melhor compreensão do sistema e oferece uma arquitetura mais extensível permitindo a que a visualização possa ser adaptada, de acordo com os requisitos da análise. A partir desse trabalho pudemos perceber a necessidade de que nossa ferramenta oferecesse a possibilidade de exibição dos dados analisados de diferentes formas, a fim de que diferentes aspectos da análise pudessem ser explorados pelo desenvolvedor.

O trabalho de Ju Li [13] utiliza um mecanismo para capturar a informação de causalidade ao longo do sistema e construir um grafo de chamadas do sistema (*dynamic system call graph*). Esse trabalho apresenta como principal dificuldade o estabelecimento da relação de causalidade global e também apresenta uma abordagem baseada na instrumentação dos *stubs* e *skeletons* para contornar tal dificuldade fazendo com que as informações sobre a causalidade global se propaguem pelo sistema. Neste trabalho utilizaremos uma abordagem ligeiramente diferente, fazendo a instrumentação da aplicação com o auxílio do mecanismo de interceptadores definidos no CORBA e oferecendo suporte para DII e DSI.

Em geral, pudemos notar a necessidade de oferecer ao desenvolvedor um conjunto de ferramentas para auxiliar na compreensão de diversos aspectos do

sistema e facilitar a interpretação das informações obtidas durante a execução.