4
Trabalhos Relacionados

A adaptação e implantação dinâmicas de componentes são requisitos de sistemas distribuídos em diversos domínios. Existem diversos sistemas tradicionais, como [53], [54], [55], [56], que tratam deste assunto. Neste capítulo, são apresentados alguns sistemas adaptativos que possuem características em comum com o Kaluana. Estes sistemas serão analisados e comparados com o trabalho descrito nesta dissertação. A abordagem utilizada por cada sistema para realizar adaptações será detalhada e refletida nos resultados obtidos nas comparações.

4.1.
CASA

O framework CASA [10] fornece uma plataforma que possibilita a adaptação de aplicações pervasivas executando em ambientes dinâmicos. Assim como no Kaluana, cada nó adaptativo deve estar executando uma instância do CSR (CASA Runtime System). O CSR monitora as mudanças no ambiente de execução das aplicações, e em caso de mudanças significativas, realiza a adaptação dinâmica das aplicações. A política de adaptação de todas as aplicações é definida no que é chamado de contrato de reconfiguração.

Os três pontos que o *framework* CASA possui em comum com o Kaluana são: ele trabalha com a alteração dinâmica de serviços de baixo nível usados pelas aplicações, mudança dinâmica dos atributos da aplicação e recomposição dinâmica de componentes. Exemplos de serviços de baixo nível incluem a modificação do nível de qualidade de compressão dos dados sendo transmitidos em um canal de comunicação em resposta à modificação da banda disponível. O CASA possui ainda o *weaving e unweaving* dinâmico de aspectos, que não é uma característica do Kaluana. Para realizar a integração com aspectos computacionais, o CASA se baseia na flexibilidade do sistema PROSE [57], que é desenvolvido especificamente para este propósito.

O trabalho citado pode ser integrado a qualquer *middleware* adaptativo que possua suporte a regulações externas. Neste aspecto, ele se assemelha ao Kaluana de modo que ambos podem se conectar a sistemas provendo a reconfiguração de seus serviços em tempo de execução. Por exemplo, o *middleware* Kaluana foi integrado, para este trabalho, com a plataforma Mobilis [49], transferindo a responsabilidade de ciência ao contexto para os serviços do Mobilis.

Outro ponto em comum entre o Kaluana e o *framework* CASA é o fato de ambos poderem realizar a recomposição dinâmica de componentes. No entanto, o processo de recomposição destes dois trabalhos é feito de uma maneira diferente. Neste *framework*, optou-se por realizar a substituição de componentes usando a estratégia *eager* (onde os componentes são substituídos durante a sua execução) em contraste com a estratégia *lazy*. Para isto, é utilizado um *proxy* que armazena o estado de execução do componente a ser substituído. Realiza-se a conexão com o novo componente, e continua a execução deste novo componente a partir do estado em que ele se encontrava antes da reconfiguração. No *middleware* Kaluana, foi adotado método *lazy* de substituição (onde o *middleware* aguarda o término da execução do componente, para em seguida realizar sua substituição). Estes métodos de reconfiguração são representados na Figura 5. Na figura, as barras preenchidas representam os componentes antes utilizados pela aplicação, antes da adaptação, enquanto as barras brancas representam os componentes novos que foram acoplados após a reconfiguração. Realizar o salvamento e recuperação do estado de execução de um componente não é um trabalho trivial e necessitaria de uma pesquisa e implementações mais extensas. Por este motivo, este ponto não foi tratado neste trabalho.

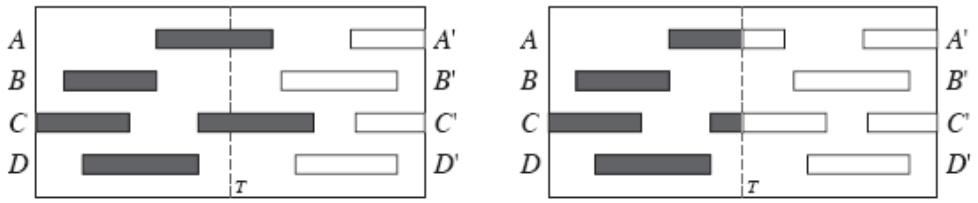

Figura 5. Métodos de reconfiguração de componentes [10].

O *framework* também pode fazer a troca dos atributos de uma aplicação de maneira dinâmica, de maneira similar ao *middleware* Kaluana. Ambos os trabalhos necessitam receber, durante a requisição da reconfiguração, uma

callback, que será chamada ao final do processo de adaptação, informando à aplicação sobre o sucesso ou falha deste processo.

O contrato de reconfiguração do *framework* é realizado em um arquivo XML a parte, onde são descritos os atributos do componente como seu contexto de execução e seus pontos de interesse para as aplicações. No Kaluana, este contrato é realizado por anotações Java que informam os serviços, receptáculos e restrições de um componente.

No entanto, o trabalho citado possibilita a inserção e remoção de aspectos de forma dinâmica. Para isto, é utilizada a plataforma PROSE [57], que fica responsável por fazer o trabalho de manipulação destes aspectos. O *middleware* Kaluana não faz uso de programação orientada a aspectos em nenhuma parte de seu sistema pois até o momento da implementação, a orientação a aspectos não era suportada pela plataforma Android.

4.2. IAM

Outro trabalho bastante relacionado com o *middleware* Kaluana é o *An Infrastructure for Adaptable Middlware* [58], o qual será citado neste texto como IAM, para facilitar sua leitura. Assim como o trabalho aqui apresentado, o IAM também possui um módulo chamado de "*core"*, que monitora e detecta mudanças no estado dos recursos de execução do nó de execução, como é mostrado na Figura 6. Quando uma mudança significativa é detectada, a *engine* de adaptação é notificada. Este componente é encarregado de decidir qual modificação é necessária para adaptar o sistema, baseado em políticas de adaptação.

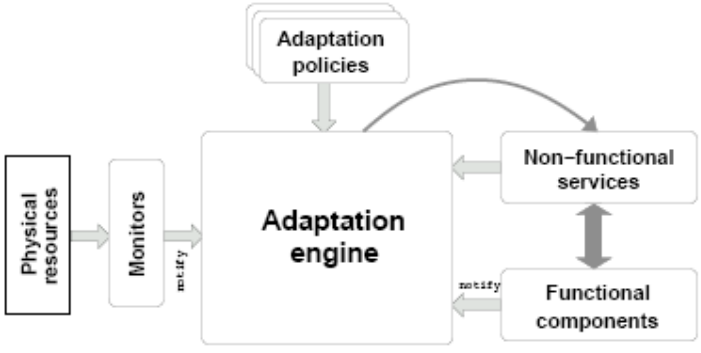

Figura 6. Arquitetura global do IAM [58]

No entanto, este *framework* propõe uma separação clara entre componentes funcionais e não funcionais. Para o IAM, o desenvolvedor da aplicação adaptável, não tem a necessidade de seguir qualquer regra/norma de programação como implementar interfaces ou seguir padrões de projeto, diferente do Kaluana, onde a aplicação deve explicitamente possuir uma referência para o gerenciador de adaptações, o qual permite à aplicação requisitar a qualquer momento, a troca de um componente, usando essa referência para o gerenciador de adaptação. Isto é descrito em mais detalhes em [12].

O IAM faz a associação de cada componente funcional a um controlador de metanível, que intercepta e interpreta todas as criações de objetos, recepção de métodos e acessos a campos. Para isto, a infra-estrutura de metanível usa o protocolo RAM de metaobjetos [59], o qual faz o controle dos metaobjetos através da pré-compilação do código usando AspectJ [60]. A Figura 7 mostra um exemplo de interface de um metaobjeto no IAM. Os métodos básicos a serem implementados, proporcionam o controle do objeto através de reflexão computacional [61].

```
public class MetaObject implements Serializable {
    public Object create(ReflectiveObject creationRequestor,
                         Constructor constructor, Object[] initArgs);
    public void initialize(ReflectiveObject base, Object[] parameters);
    public Object invokeMethod(ReflectiveObject receiver, Method method, Object[] args);
    public Object getField(ReflectiveObject target, String fieldName);
    public Object setField(ReflectiveObject target, String fieldName,
                           Object oldValue, Object newValue);
    public void deserialize(ReflectiveObject obj, ObjectInputStream in);
    public void serialize(ReflectiveObject obj, ObjectOutputStream out);
}
```

Figura 7. Exemplo de um metaobjeto no IAM, usando o *RAM MetaObject protocol* [58]

Como foi dito na seção anterior, o Kaluana não faz uso de aspectos em seu código. Esta opção foi descartada durante a fase de projeto do *middleware*, pois a plataforma Android provê um ambiente com reflexão computacional e até a fase de projeto do *middleware*, esta plataforma não suportava o uso de aspectos.

Os requisitos não-funcionais são tratados pelo IAM através serviços e papéis (*roles*). Isto permite que múltiplas implementações do mesmo serviço, com diferentes propriedades, possam ser implementados. Esta camada, a qual é mostrada na Figura 8, também é implementada no Kaluana, porém de uma maneira ligeiramente diferente. Um componente *A*, pode implementar o serviço *s1*, que pode ser do mesmo tipo que o serviço *s2*, do componente *B*, porém com uma

implementação diferente. No entanto, o componente *B*, pode possuir outros serviços que o componente *A* não implementa. Sendo assim, o componente *A* seria mais leve de carregar e de executar, fazendo com que a aplicação gastasse menos energia do dispositivo.

Desta maneira, o Kaluana faz um melhor aproveitamento dos componentes e serviços, de maneira a tornar a aplicação mais leve, ou mais completa, de acordo com o contexto computacional do dispositivo.

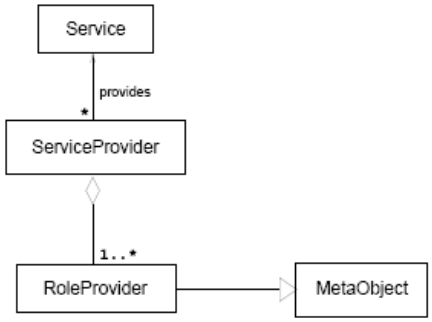

Figura 8. Camada que trata de requisitos não funcionais, no IAM [58]

O *framework* de observação guia o ambiente de execução e o sistema em si através do conhecimento de ambos. Este *framework* possui dois papéis: expor informação suficiente para a *engine* de adaptação permitindo que esta tome decisões de maneira correta e detectar mudanças significativas nas informações do sistema. A importância de uma mudança depende da semântica de cada aplicação. Por isso, este *framework* é configurado de acordo com cada aplicação executada, a partir das políticas de adaptação.

Cada nó do *framework* é descrito através atributos representados por pares *nome/valor*. O valor destes atributos pode mudar dinamicamente para representar o estado do recurso em que estão anexados. A Figura 9 sumariza a organização do *framework* de observação de recursos no IAM.

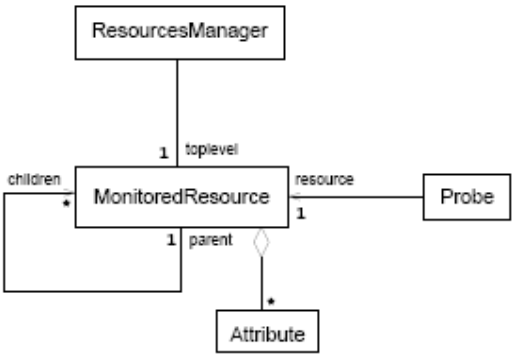

Figura 9. Organização do *framework* de observação de recursos no IAM [58]

Para este trabalho, o Kaluana transfere a responsabilidade de realizar o monitoramento dos recursos do dispositivo para a plataforma Mobilis. Isto é feito através da integração do *middleware* Kaluana com o serviço de gerenciamento de contexto (CMS), que monitora as informações providas pela plataforma Android.

O IAM conta com uma *engine* de adaptação única, que é configurada para atender aos requisitos de cada aplicação de acordo com suas necessidades. Estas configurações são feitas através das políticas de adaptação. Assim como no Chisel (descrito na seção 4.3), o IAM também usa uma linguagem própria para descrever as políticas de adaptação. Esta é uma linguagem usa uma sintaxe *Scheme-like*, que a faz fácil de fazer *parser* e flexível. No entanto, a linguagem não é completamente *scheme*, pois as primitivas disponíveis para definir as condições que são partes da regra incluem apenas operações e comparações numéricas básicas, comparações de strings e operadores lógicos. A Figura 10, mostra um exemplo de uma política de adaptação para o IAM.

```
(def-policy "distribution.server"
  (when #t (ensure-attached "distribution.rpc.server")))

(def-policy "distribution.default-client"
  (when #t (ensure-attached "distribution.rpc.client"
                            ((server . "pollux.info.emn.fr")))))

(def-policy "distribution.wireless-client"
  (when (= (attr "/host/network.connection-type") "wireless")
    (ensure-attached "invocation.asynchronous.future"))
  (when (and (= (attr "/host/network.connection-type") "wireless")
             (< (attr "/host/network.available-bandwidth_kbs") 5)
    (ensure-attached "proxy.caching"))))

(def-policy "distribution.mobile-client"
  (when (and (not (attr "/host/battery.charging"))
             (< (attr "/host/battery.autonomy_s") 300))
        (ensure-attached "persistence")))
```

Figura 10. Políticas de adaptação no IAM [58].

Para o IAM, diferente do Kaluana, existem dois tipos de política de adaptação: Políticas de sistema e políticas de aplicação. A primeira dita as regras de adaptação para camadas mais baixas do sistema. Seu papel é definir as regras de adaptação independentemente da semântica da aplicação. Elas são normalmente criadas por especialistas do sistema e podem ser reusadas em diferentes aplicações. Concretamente, uma política de adaptação do sistema é um conjunto de regras do tipo *condição=>ação.* Estas políticas respondem duas perguntas: "Quando?" e "O que fazer?" No entanto são as políticas de adaptação das aplicações que dizem "a quem". Elas possuem o papel de informar a *engine* de adaptação quais componentes funcionais presentes no sistema, devem ser afetados por qual política do sistema.

Para o Kaluana, existe apenas um tipo de política de adaptação (ou reconfiguração). Esta política é definida através de anotações java, que indicam ao *middleware* o comportamento do componente através de suas restrições de execução. Através destas anotações, o sistema faz uma leitura do componente usando reflexão computacional, para poder prover para a aplicação os serviços necessários para sua execução.

Diferente do IAM, não existem "políticas de sistema" no Kaluana. A própria aplicação toma a decisão de quando é necessário adaptar-se, deixando para o *middleware* a decisão de qual seria a melhor maneira de fazer isso. Este código está intrínseco no sistema e não é um módulo a parte. O código toma suas decisões através de leituras constantes da situação do contexto de execução do

dispositivo. Este contexto é provido ao *middleware* pela própria Plataforma Android, através da integração do *middleware* Kaluana com a plataforma Mobilis.

Desta maneira, o Kaluana possui uma plataforma menos flexível que o IAM, porém torna o código do *middleware* mais simples, pois ainda não se fez necessário modularizar esta parte do sistema, visto que ela é relativamente pequena.

4.3. Chisel

O objetivo principal do projeto Chisel [11] é construir um *framework* para dar suporte a adaptações dinâmicas não antecipadas. Estas adaptações são baseadas em informações contextuais da maior quantidade de origens possível. Estas origens incluem informações de baixo nível sobre a modificação da natureza de execução do ambiente, mas também incluem conhecimento de alto nível e inteligência a respeito da aplicação sendo adaptada e do usuário da aplicação.

O *framework* Chisel usa a arquitetura Iguana/J [62] para tratar, nos níveis mais baixos da aplicação, a associação de meta-objetos e operações com objetos em tempo de carregamento e execução. O *middleware* Kaluana não teve a necessidade de usar uma arquitetura extra para tratar a manipulação de objetos de baixo nível, pois a plataforma Android fornece suporte à reflexão computacional, a qual foi usada para associar e desassociar componentes das aplicações.

Assim como o Kaluana, o *framework* Chisel é baseado em políticas de adaptação que controlam as escolhas de comportamento do sistema controlado. No entanto, estas políticas são explicitadas de maneiras diferentes nos dois sistemas. No Chisel, as políticas são definidas em um arquivo separado (um *protocol declaration file*), que define os MOPs [63] através de uma linguagem própria, como é visto na Figura 11

```
ON WirelessDisconnect:
  NetworkConnectionService.WiredBehaviour
    IF NetworkConnectionService.WiredAvailable == True
          &&
          WirelessDisconnect.IsTemporary == False
```

Figura 11. Linguagem definição de políticas de reconfiguração no Chisel [11]

Para o Kaluana, estas políticas de adaptação são tratadas através dos contratos de reconfiguração, definidos através de estruturas de dados que contém os serviços, receptáculos e restrições de um componente. Para defini-los, o desenvolvedor usa anotações Java. Os contratos de reconfiguração de um componente são descritos em detalhes na seção 5.2 desta dissertação.

O *framework* Chisel faz uma separação do mecanismo de adaptação em diversos módulos. São definidos uma série de módulos distintos dentro do meta-nível do gerenciador de adaptação, como mostrado na Figura 12.

O gerenciador de contexto irá monitorar os recursos para uma mudança contextual apropriada. Quando uma mudança de contexto ocorrer, o gerenciador de contexto (***Context Manager***), em conjunto com o gerenciador de regras (***Rule Manager***), irá verificar o conjunto de políticas para identificar a política mais relevante que desencadeará a adaptação relacionada à mudança de contexto específica.

O gerenciador de comportamento (***Behaviour Manager***) é responsável por realizar a adaptação comportamental através da reassociação dos meta-tipos de um componente, no objeto de um serviço gerenciado.

O gerenciador de políticas (***Policy Manager***) fica responsável por rastrear as mudanças aos arquivos de declaração de políticas de reconfiguração, realizando um *parse* incremental dos arquivos interpretados.

Por fim, o gerenciador de regras fica responsável por avaliar as regras enviadas pelo gerenciador de políticas, em conjunto com a informação de contexto passadas pelo gerenciador de contexto, desencadeando assim a mudança comportamental apropriada, através do gerenciador de comportamento.

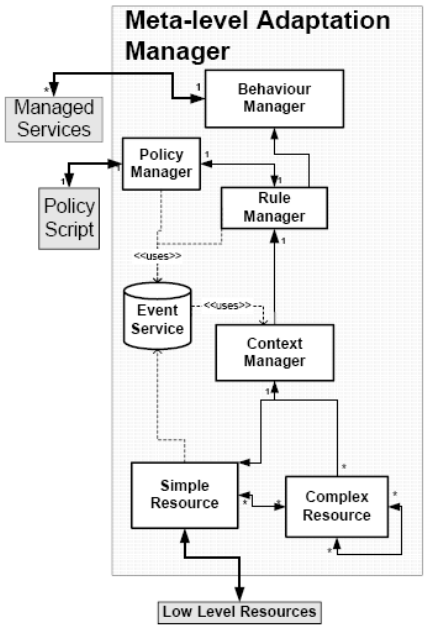

Figura 12. Mecanismo de adaptação do Chisel [11]

Isto torna o sistema mais flexível, no entanto torna seu projeto e desenvolvimento mais complexos por conta do tempo de planejamento necessário para modularizar e componentizar cada parte do sistema. Por este motivo, o *middleware* Kaluana possui apenas um módulo de gerenciamento de adaptação, o *AdaptationManager*, que controla por completo o processo de adaptação, ficando encarregado de fazer a interceptação de requisições de adaptação, assim como monitorar mudanças do contexto do dispositivo.

4.4. DynamicTAO

Outro trabalho relacionado com o Novo Kaluana cria uma extensão para o TAO [64], para aproveitar de suas vantagens de modularizar e organizar *middlewares*. O DynamicTAO [65] é um ORB CORBA reflexivo para dar suporte de reconfiguração em tempo de execução enquanto assegura que o ORB continue consistente, isto é feito através da especialização da estrutura interna do ORB, que possibilita a alteração de estratégias específicas de seu comportamento sem a necessidade de reinicializar sua execução.

Diferente do Kaluana, o DynamicTAO usa um arquivo de configuração que especifica as estratégias usadas pelo ORB para implementar aspectos como concorrência, demultiplexação, agendamento e gerenciamento de conexão inter-

objeto. No Kaluana, não é necessário que o desenvolvedor trate estes aspectos diretamente, pois eles são gerenciados pelo próprio Android, facilitando sua implementação.

O DynamicTAO realiza suas alterações através de uma coleção de entidades ou configuradores de componentes [66]. Um configurador de componente carrega as dependências entre um certo componente e outro sistema de componentes. Cada processo executando um dynamicTAO ORB contém uma instancia de um *component configurator* chamada *DomainConfigurator*. Ela é responsável por manter as referencias para as instancias do ORB e para os *servants* executando nesse processo. No Kaluana Original, foi convencionado que estas configurações seriam feitas através de anotações Java no código, informando as dependências de um componente. Estas anotações facilitam a codificação de um componente, pois não é necessário usar arquivos de configuração ou mesmo outras linguagens para especificações do componente. Para o Novo Kaluana, este aspecto não foi alterado.

Assim como o Novo Kaluana, o TAO *Configurator* contém pontos nos quais a implementação das estratégias é anexada. Estes A Figura 13 mostra o mecanismo de especialização do DynamicTAO. Nota-se que se faz necessário possuir um configurador para especificar as estratégias adotadas, em contraste com o Novo Kaluana, onde esta camada não existe e os componentes são conectados diretamente às aplicações.

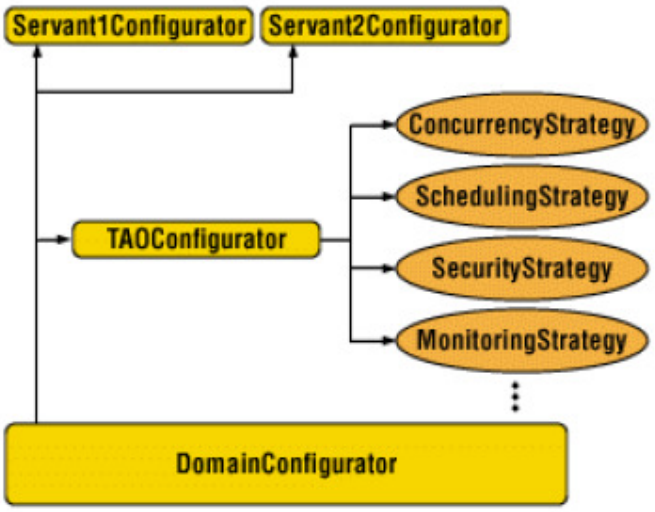

Figura 13 Mecanismo de especialização do DynamicTAO [65]

A arquitetura do DynamicTAO, assim como a arquitetura do Kaluana, possui um repositório, onde as categorias dos componentes são persistidas. A diferença entre estas duas implementações se dá no modo em que estas categorias são persistidas. No DynamicTAO, as categorias são salvas em arquivos locais, enquanto no Novo Kaluana, o repositório local é guardado em memória até que seja necessário persisti-lo, evitando demoradas operações de IO para acessar as especificações de um componente. A Figura 14 mostra a arquitetura do DynamicTAO.

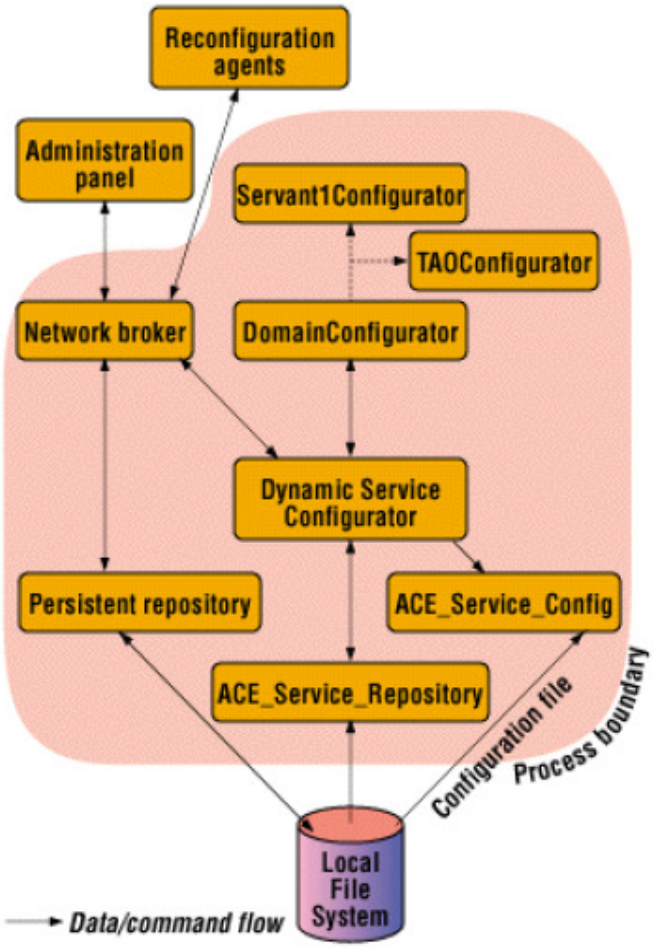

Figura 14 Arquitetura do DynamicTAO [65]

Durante o processo de adaptação, o DynamicTAO determina se a implementação do novo componente é compatível com as implementações já implantadas verificando outros componentes executando no mesmo ORB e, em

alguns casos, avaliando componentes da mesma categoria em outros ORBs. Este processo também é realizado no Novo Kaluana, através da avaliação dos contratos de reconfiguração dos componentes candidatos a implantação de acordo com a compatibilidade de seus tipos. Caso não sejam encontrados componentes compatíveis com a requisição, são buscados componentes em um servidor remoto, evitando que a requisição falhe.

O DynamicTAO trabalha em conjunto com o Universally Interoperable Core [67] para lidar com problemas existentes em plataformas de *middleware* para computação ubíqua. O UIC define um *skeleton* baseado na descrição dos componentes, no qual encapsula os aspectos básicos de funcionalidades mais requisitadas por objetos do tipo *broker* (protocolos de transporte de rede, estabelecimento de conexões, etc). O UIC permite a integração de diferentes dispositivos móveis heterogêneos permitindo interoperação entre plataformas nativas de *middleware*. O Novo Kaluana não possui a necessidade de usar outros *middlewares* para tratar a interconexão de suas aplicações entre dispositivos de heterogêneos. Este trabalho é realizado pelo próprio Android, que se responsabiliza por fazer a interlocução dos dispositivos. No entanto, para que isso ocorra, é necessário que todos os dispositivos (independente do hardware usado) estejam executando a plataforma Android para que esta interoperabilidade seja possível para o Novo Kaluana.

4.5. Síntese dos trabalhos

Nesta seção são listadas as principais semelhanças e diferenças entre os trabalhos citados, e o *middleware* Kaluana.

O principal ponto de comparação entre os trabalhos é o fato de todos usarem conceitos similares a contratos de reconfiguração para definir o comportamento dos componentes utilizados na reconfiguração dinâmica. A pesar disso, cada um possui sua própria maneira de estruturar estes contratos de reconfiguração. Por exemplo, enquanto o Chisel e o IAM definem uma linguagem própria para explicitar os contratos de seus componentes, o CASA e o Kaluana usam estruturas já conhecidas, como XML e anotações Java, para estas definições.

Outro ponto importante a ser comentado é o fato do CASA e do IAM usarem aspectos computacionais para controlar partes do sistema (como anexar políticas de adaptação em tempo de execução). Até a fase de projeto do Kaluana, a plataforma Android não suportava o uso de aspectos computacionais, o que impossibilitava a utilização deste tipo de artifício para realizar controle das adaptações.

Dos cinco trabalhos (incluindo o Kaluana), os únicos que não foram inicialmente concebidos para atuar em um ambiente de mobilidade foram o Chisel e o DynamicTAO. No entanto, em conjunto com o *framework* ALICE (Architecture for Location-Independent Computing Environments) [55], foi criada uma implementação para executar o Chisel em ambientes móveis. O DynamicTAO usa o UIC [67] para criar especializar sua implementação para um ambiente móvel.

Sistema	Aspectos computacionais	Mobilidade	Contrato de reconfiguração
Chisel	Não	Sim, em conjunto com o ALICE	Sim, usando arquivos XML
CASA	Sim, usando PROSE	Sim	Sim, usando linguagem própria
IAM	Sim, usando AspectJ	Sim	Sim, usando linguagem própria
DynamicTAO	Não	Sim, em conjunto com o UIC	Sim, usando CORBA
Kaluana	Não	Sim	Sim, usando anotações Java