5.
Arquitetura do MobiWfMS

Este capítulo apresenta a arquitetura proposta o MobiWfMS, o sistema de gerência de *workflow* com a utilização de dispositivos móveis. A seção 5.1 mostra a arquitetura descrevendo seus componentes e suas funcionalidades. A seção 5.2 descreve os pontos específicos da arquitetura, como particionamento*, triggers* de emergência e controle de perdas de mensagem. A seção 5.3 apresenta um modelo para coreografia. Por fim, a seção 5.4 resume o capítulo.

5.1.
Descrição da arquitetura

A arquitetura proposta neste trabalho abrange os dois tipos de coordenação: orquestração e coreografia. A coreografia pode ser vista como uma extensão da orquestração. Ela é basicamente a arquitetura da orquestração sem o controlador central e com a comunicação diretamente entre os parceiros. Desta forma, todas as características da arquitetura para a orquestração valerão também para a coreografia. As características específicas da coreografia na arquitetura serão discutidas na seção 5.3.

A arquitetura geral do sistema é apresentada na Figura 12. Cada um dos componentes da arquitetura é descrito a seguir:

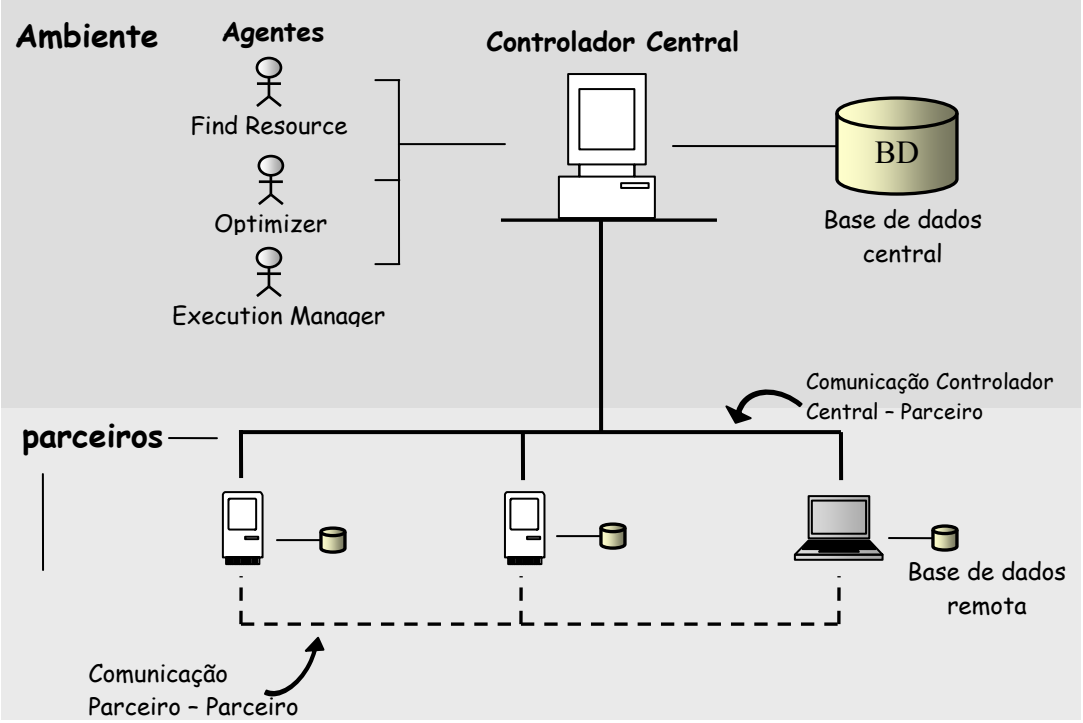

Figura 12 – Arquitetura geral do sistema

Participantes

Para os dois tipos de coordenação de *workflow* existem duas funções possíveis de serem desempenhadas pelos participantes:

- **Controlador Central**: um microcomputador com conexão confiável e permanente à Internet que funcionará como o servidor central. Este será o ponto centralizador da definição e execução do *workflow*.

- **Parceiros**: PDAs que participarão do *workflow*. Além dos dispositivos móveis, este papel poderá também ser desempenhado por microcomputadores. Esta é uma abordagem possível, como já dito no capítulo 3, que descreve os requisitos, mas que só será discutida na arquitetura, não sendo implementada.

Bases de dados

As bases de dados são necessárias para manter o registro dos dados adicionados ao sistema (requisitos 1, 5 e 6), bem como os dados gerados através da execução do *workflow*. As bases estão associadas aos dois tipos de participantes do processo.

Existe uma base de dados central gerenciada pelo controlador central. Esta base contém todos os dados do processo e deve sempre estar sincronizada com as outras bases de dados da arquitetura.

Além da base de dados central, existem bases de dados remotas associadas aos parceiros. Cada parceiro possui uma base de dados que armazena os dados pertinentes à sua parte do *workflow*.

O modelo de dados detalhado das bases de dados será descrito no capítulo 6.

Canais de comunicação

Um canal de comunicação é o canal pelo qual será feita a troca de informações entre os processos para que os participantes possam desempenhar suas tarefas (Figura 13).

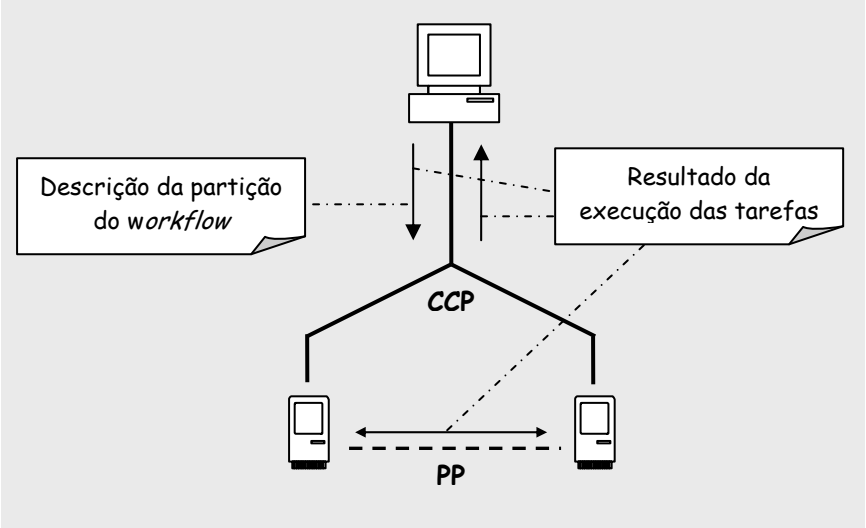

Figura 13 – Tipos de canais de comunicações e mensagens trocadas

Existem dois tipos de canais de comunicação entre os participantes na arquitetura:

- Canal de comunicação Controlador Central – Parceiro (CCP): este canal de comunicação é o mais comum e é utilizado sempre que há uma comunicação do tipo cliente-servidor. Ela acontece tanto na orquestração quanto na coreografia.

- Canal de comunicação Parceiro – Parceiro (PP): este é um tipo de comunicação que é estabelecida entre dois parceiros do processo sem a intermediação

do servidor. Ela está presente unicamente na coreografia. A comunicação PP pode também ser chamada de ponto-a-ponto (do inglês *peer-to-peer*).

As informações do processo que são transmitidas através destes canais de comunicação estabelecidas entre os participantes são:

- Descrição da partição do *workflow* enviada do controlador central para o parceiro. Nesta descrição contém todos os dados necessários para o correto entendimento do processo por parte do parceiro.

- Resultado da execução das tarefas: ocorre nos dois sentidos da comunicação Controlador Central – Parceiro. O parceiro pode informar ao servidor a finalização da tarefa, ou solicitar dele o estado de uma tarefa que está sendo executada por outro parceiro. Exclusivamente na coreografia, um parceiro pode informar outro diretamente sem o intermédio do servidor, o que caracteriza o segundo tipo de comunicação. Diante disso, é necessário que o parceiro tenha conhecimento das informações necessárias para conversar diretamente com outro parceiro.

O correto gerenciamento dessa troca de informações através desses canais de comunicação é fundamental para o perfeito funcionamento da arquitetura proposta. Perdas de mensagens, decorridas de falhas em um ou mais parceiros, devem ser controladas para evitar inconsistência de dados e garantir a correta execução do processo. O mecanismo de controle de perdas de mensagens proposto nessa dissertação é baseado na ação do agente gerente de execução e dos mecanismos de gerência exceção, os quais serão descritos no decorrer deste capítulo.

Agentes

A arquitetura sugere a criação de três agentes para auxiliar no processo de construção e execução do *workflow*. Estes agentes compõem a inteligência do controlador central e atua na definição e execução do *workflow*. No caso da coreografia, os agentes podem ser transferidos para dentro dos parceiros.

***Agente* Execution Manager**

O agente *Execution Manager* é o principal agente desta arquitetura. Ele auxilia o sistema no correto gerenciamento da execução do *workflow*. Este agente vigia a

execução do *workflow* e, a qualquer momento, pode tomar decisões para manter a consistência do sistema. O objetivo dele é controlar as perdas de mensagens ocorridas pela desconexão permanente dos parceiros. Quando, por exemplo, um parceiro perde o contato com o controlador central e não informa o resultado da execução da sua partição, este agente entra em ação e dispara os mecanismos de exceção associados à falha ocorrida. O agente *Execution Manager* entra em ação quando o *timeout* definido para a tarefa é atingido.

Os mecanismos de exceção, bem como a forma como eles são inicializados pelo agente, são discutidos na seção 5.2 a seguir, mais especificamente no item "Mecanismos para gerência de falhas".

Agente Find Resource

Este agente é o responsável pela procura de recursos disponíveis na rede. A procura é baseada nos metadados cadastrados no sistema. Quando o agente *Find Resource* encontra um recurso possível de ser utilizado que seja relevante ao MobiWfMS ele o cadastra na base de dados (Figura 14). A ação isolada desse agente não tem relevância. Ela surtirá efeito quando outro agente, o *Optimizer*, entrar em ação para desempenhar seu papel na arquitetura.

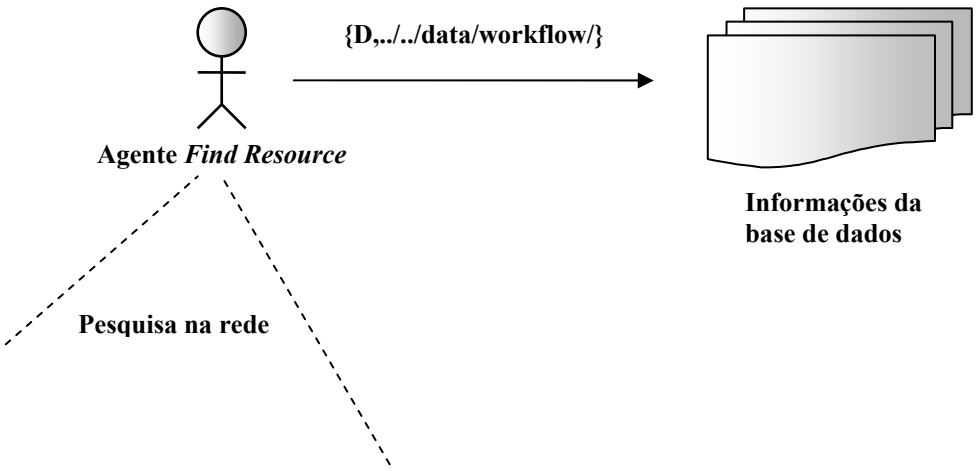

Figura 14 – Ação do agente procura recurso

Agente Optimizer

Este agente é o responsável pela otimização do *workflow* criado. Diversos trabalhos sobre otimização de *workflow* podem ser encontrados na literatura (Minglun et al., 2000; Singh et al., 2005; Lican et al., 2005). Este é um tema importante, pois *workflows*, principalmente os científicos, são complexos e seu processamento pode durar dias. Qualquer eliminação de parte do *workflow* pode atenuar o tempo total de processamento.

Neste trabalho, os dispositivos móveis de baixo poder de processamento funcionam como outra justificativa para a otimização dos *workflow*. Diante disso, qualquer eliminação de trabalho desnecessário será bem aproveitada.

A otimização que o agente *Optimizer* faz está relacionada aos dados materializados no sistema. Os dados materializados podem ser cadastrados pelo usuário ou através da ação do agente *Find Resource*.

Considere que uma tarefa T1 produz um dado D para uma tarefa T2. Se T1 não possui outra atribuição dentro do *workflow* que não seja produzir o dado para T2 e o dado D já existe materializado em algum lugar da rede, D pode ser utilizado diretamente e T1 pode ser eliminado do *workflow* (Figura 15).

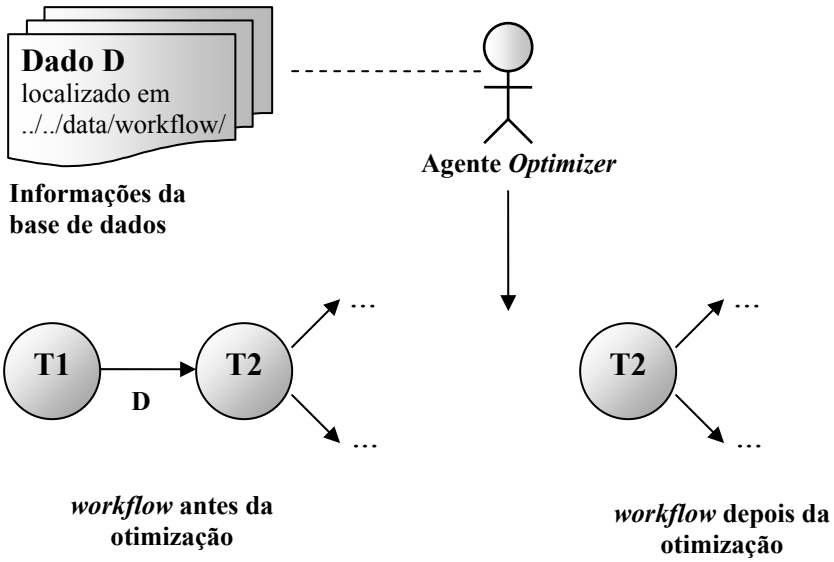

Figura 15 – Ação do agente otimizador

Estes agentes apresentados compõem basicamente a inteligência do coordenador central na orquestração. No caso da coreografia, estes agentes devem estar implementados dentro dos parceiros.

5.2. Características adicionais da arquitetura

Esta seção apresenta as características adicionais da arquitetura que cobrem o restante dos requisitos previstos no capítulo 3. Serão discutidos nessa seção três itens: particionamento (requisito 2), para permitir a divisão e distribuição do *workflow*; *triggers* de emergência (requisito 3) e mecanismos de gerência de falhas que são necessários para a correta realização da execução (requisitos 4 e 7).

Particionamento

O particionamento do *workflow* pode ser visto como uma premissa para o funcionamento em ambientes distribuídos. Caso o *workflow* seja considerado apenas como um único bloco de tarefas, certas vantagens, como a execução paralela, não poderiam ser obtidas da estrutura distribuída da rede. A arquitetura proposta neste trabalho supõe o funcionamento em um ambiente distribuído com diversos parceiros trabalhando em paralelo. Desta forma, o particionamento é um requisito que deve ser suportado nesse trabalho.

A forma como o *workflow* deve ser particionado depende da semântica do negócio ao qual o *workflow* está associado. Muitas vezes, existem regras bem definidas que permitem que o *workflow* seja particionado de forma automática. Citamos como exemplo de particionamento automático o Pegasus, um *framework* flexível que permite o mapeamento de complexos *workflow* científicos em um *grid* (Pegasus, 2005) e que possui suporte à divisão automática de *workflow* (Deelman et al., 2005). Em adição, o trabalho efetuado pelo MobiWork (Hackmann et al., 2006), citado anteriormente nos trabalhos relacionamentos, pode também ser visto como um tipo de divisão do *workflow*, ao passo que ele classifica as tarefas e as envia para serem efetuadas por um parceiro que possua os requisitos necessários para executá-la.

É importante observar que estes sistemas possuem regras específicas para o particionamento automático dos *workflows*. Muitas vezes, fazem sentido apenas no domínio da aplicação ao qual ele está associado. Automatizar o particionamento do *workflow* não faz parte do foco deste trabalho. Entretanto, para que haja um processamento distribuído entre os parceiros, é necessário permitir que o usuário possa efetuar esta tarefa, ainda que de forma manual.

A Figura 16 apresenta um modelo de *workflow* que auxiliará no entendimento de como será feito o particionamento neste trabalho além das suas implicações.

Neste *workflow* há onze tarefas nomeadas de A a K. Suponha que, por algum motivo, o usuário dividiu o *workflow* em três partes *P1* = {A, B, C, D}, *P2* ={H, I, J, K} e *P3* ={E, F, G}, como mostrado na Figura 16. Uma partição é a unidade básica que é enviada para um parceiro, podendo conter apenas uma tarefa. Cada parte especificada pelo usuário poderá ser enviada para qualquer parceiro que possua características suficientes para executar tal partição. Uma partição pode conter tarefas de diferentes tipos. Diante disso, um parceiro só poderá executar uma partição se ele puder executar todos os tipos das tarefas pertencentes à esta partição. Isto é similar ao funcionamento das partições automáticas. A tarefa não é atribuída a um parceiro que não satisfaça os requisitos necessários para realizá-la.

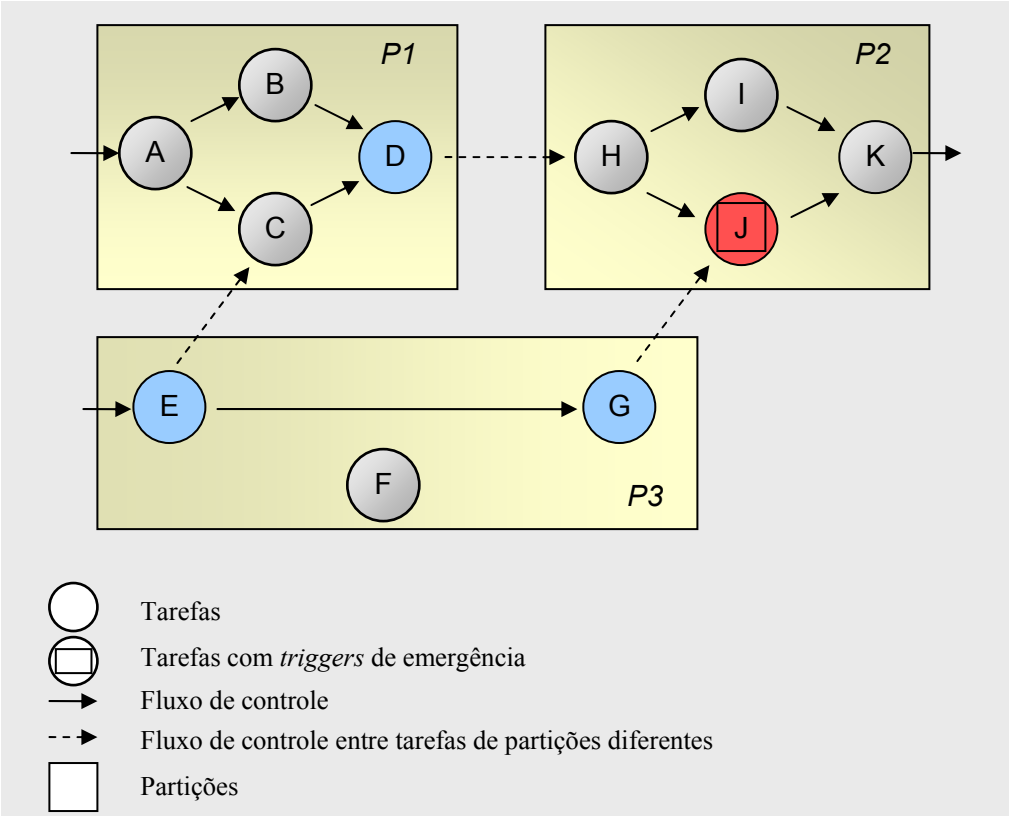

Figura 16 – Exemplo de um *workflow* simples com particionamentos

Ainda no exemplo, é possível observar a relação de dependência entre as tarefas definida pelos fluxos de controle, de forma que algumas tarefas só podem ser executadas após outras: K após I e J; G após E; B após A; etc. Esta dependência é conhecida pelo controlador central e deve também ser de conhecimento do parceiro responsável pela execução. No caso em que duas tarefas que possuem dependência entre si estiverem na mesma partição, o parceiro cuidará de informar à tarefa sucessora que sua precedente já foi finalizada. Já para os casos em que uma tarefa é pré-requisito para outra tarefa que está em outra partição (tarefas D, E e G), há a necessidade de se informar o servidor quando essa tarefa terminar ou, para o caso da coreografia, informar o parceiro diretamente.

Uma solução seria efetuar uma comunicação Parceiro-Controlador central a todo o momento que uma tarefa fosse concluída no parceiro. Isto seria uma solução aceitável não fossem as características dos parceiros. Os parceiros possuem processamento e conexões com a Internet limitados. Processamento desnecessário

nesses parceiros pode prejudicar o desempenho da aplicação, podendo tornar a solução impraticável.

Outra solução, visando evitar estes problemas, seria só conectar com o servidor ao finalizar toda a realização da partição. Suponha por exemplo que a tarefa E seja rapidamente executada. Entretanto, este fato só será informado ao servidor após G e F terem sido finalizados. Caso essas duas últimas tarefas possuam processamento demorado, C ficaria esperando tempo desnecessário prejudicando assim a realização da partição *P1*.

A abordagem adotada nessa arquitetura, que evita os problemas citados, consiste em fazer com que o servidor marque as tarefas que possuem sucessoras em outra partição. Para isso foi criado um atributo *responseImmediate* associado a uma tarefa (Quadro 4):

responseImmediate = 0: não possui sucessor em outra partição *responseImmediate* = 1: caso contrário. O parceiro deve conectar com o controlador central, tão logo que a tarefa seja finalizada.

Quadro 4 – Valores do atributo *responseImmediate*

Sempre que o parceiro finalizar uma tarefa marcada como resposta imediata, ele deve informar o controlador central ou o parceiro responsável pela execução da tarefa sucessora, para os casos de coreografia.

Triggers de emergência

Para ilustrar este ponto da arquitetura, suponha a seguinte situação:

"O corpo de bombeiros de uma pequena cidade recebeu um chamado para uma ocorrência classificada como grave pela tropa. Uma criança estava em situação de perigo, e para tal tipo de situação o corpo de bombeiro possuía dois tipos de procedimentos para tentar solucionar o caso. O procedimento 1, o mais comum, que deve ser usado sempre em primeiro lugar, e o procedimento 2, um procedimento de contingência, de alto custo, que deveria ser usado única e exclusivamente em situações de extrema emergência e quando o procedimento 1 não solucionasse o problema a tempo. Desta forma, o comandante preparou a seguinte estratégia. O

procedimento 1 é iniciado e se após uma hora (tempo plausível de ser esperado dentro das normas de segurança para tal situação) a solução não for alcançada, os operadores do procedimento 1 devem entrar em contato com a central e informar o ocorrido. Neste caso, o procedimento 2 será disparado e a equipe de contingência entrará em ação.

Na execução do chamado, o procedimento 1 não conseguiu solucionar o problema e, por falha de comunicação, também não conseguiu entrar em contato com a central. A equipe de contingência, desatenta à situação, só se deu conta da possível falha algum tempo depois. Tempo suficiente para que uma fatalidade ocorresse com a criança."

Veja que essa história, fictícia, mas possível de se acontecer, pode ser modelada como um *workflow*. O *workflow* contém duas tarefas (procedimentos 1 e 2) que possuem uma dependência entre si. Além disso, elas são efetuadas por equipes distintas, o que configura um caso de particionamento de *workflow*. Ainda na Figura 16, admita que o procedimento 1 esteja representado por G e que o procedimento 2 seja representado por J.

A tarefa J, representando o procedimento 2, é um caso típico de tarefa de emergência, que possui dependências, além de possuir um tempo limite de espera pela finalização das antecessoras. Nada impede que as tarefas estejam no mesmo particionamento, mas o exemplo foi dado com tarefas em partições diferentes porque é mais fácil de visualizar a situação e de aceitar uma falha. Se tratando de PDAs, eles podem facilmente se desconectar e perder o contato com o restante do processo.

A arquitetura proposta suporta este tipo de tarefa através de *triggers* de emergência, por analogia aos *triggers* de banco de dados, que disparam após uma ação no banco de dados. A ação aqui é disparada pelo tempo. Para isso, a tarefa deve conter um atributo *timeToWait*, que representa o tempo de espera máximo que uma tarefa deve esperar pelas suas antecessoras, logo (Quadro 5):

timeToWait = 0: Não é tarefa de emergência. Deve esperar pela finalização de todas suas precedentes. timeToWait > 0: Neste caso, a tarefa deve esperar por **no máximo** o tempo especificado em *timeToWait*.

Quadro 5 – Valores do atributo de *timeToWait*

Mecanismos para gerência de falhas

Sistemas com suporte a desconexão precisam prover um mecanismo de gerência de falhas que possa vir a ocorrer pela desconexão do parceiro da rede. Quando um parceiro é um dispositivo com características dos PDAs atuais, a possibilidade de falhas pode ser maior ainda. Estes dispositivos não possuem conexão estável e confiável principalmente quando comparada com computadores conectados a uma rede a cabo. Algumas possíveis falhas são:

- A operadora de telefonia celular, que normalmente é o canal pelo qual um parceiro se conecta à Internet, perde o conexão com o dispositivo;

- O dispositivo descarrega a bateria;

- Qualquer outra falha de *hardware* ou *software* que impossibilite o parceiro de dar procedimento à realização das tarefas;

A falha decorrente de uma desconexão permanente do parceiro é admitida aqui em dois níveis:

- **Nível da tarefa**: o parceiro se desconecta e o controlador central consegue detectar precisamente quais tarefas realmente falharam. Isto pode ocorrer de duas formas:

1. A partição enviada para o parceiro que falhou possuía apenas uma tarefa. Logo, a tarefa falhada é a associada à partição;
2. A partição do *workflow* contém mais de uma tarefa, entretanto o parceiro já havia informado ao controlador central da realização de algumas delas. Neste caso, as tarefas não informadas são consideradas como tendo falhado.

Uma tarefa é considerada como falhada quando o tempo de resposta da sua execução ultrapassa o tempo máximo de espera (*timeOut*) especificado para a tarefa.

- **Nível da partição**: o parceiro se desconecta sem ter retornado qualquer tipo de informação para o controlador central. Como suposições não podem ser feitas, o controlador central deve assumir que todas as tarefas da partição falharam.

Uma partição é considerada em falha quando o tempo de resposta da sua execução ultrapassa o tempo máximo de espera (*timeOut*) especificado para a partição. O *timeOut* da partição pode ser calculado como o maior somatório de todas as tarefas entre todos caminhos possíveis do grafo representativo do *workflow*.

Essas falhas geram inconsistência no *workflow*. Por exemplo, considere uma tarefa T1 e suponha que T1 é uma pré-condição para outra tarefa T2 (Figura 17). Além disso, admita também que T2 não possui *triggers* de emergência. Se T1 falha, a pré-condição de T2 não é nunca satisfeita e T2 e todo o fluxo de execução sucessor a T2 não poderão ser executados.

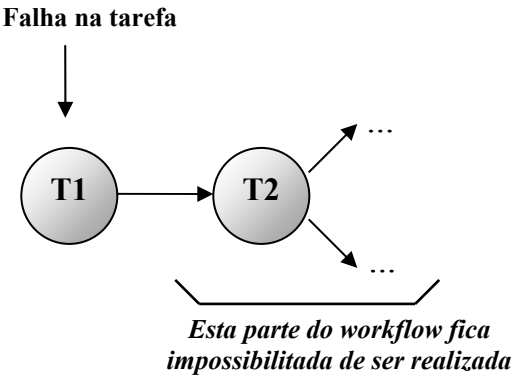

Figura 17 – Falha em uma tarefa do *workflow*

Frente a esta realidade, a arquitetura apresentada aqui propõe dois mecanismos para gerência dessas falhas dentro do *workflow*. A escolha do mecanismo é feita pelo usuário que define o *workflow* no momento da definição de cada tarefa.

1. Abortar a tarefa T1 que falhou, podendo:
    a. Não liberar a sucessora T2;
    b. Liberar a tarefa sucessora T2;

2. Executar ações de contingência (Figura 18). Este mecanismo já foi introduzido na descrição da linguagem e é ratificado a seguir. A execução das ações de contingência acontece de tal forma que:

    a. Cada tarefa do *workflow* T pode possuir uma tarefa de contingência T' (*contingencyActivityId*). T' funcionará como uma alternativa para a tarefa T. Quando T falhar, T' deve ser enviada para um parceiro disponível para executá-la

    b. Cada partição do *workflow* P pode possuir uma outra partição de contingência P'. Esta partição de contingência funciona como um sub-*workflow* alternativo a partição original e é construído a partir do conjunto de tarefas de contingência da partição. Ou seja, sendo P uma partição com o conjunto de tarefas CT[1..n] e CT'[1..m], com $m \leq n$, o conjunto de tarefas de contingência associado a CT, a partição de contingência P' será construída a partir de CT'. m é menor ou igual a n porque algumas tarefas de CT podem abortar após a falha e não conter tarefa de contingência.

A arquitetura admite como padrão o mecanismo de abortar a tarefa com bloqueio das sucessoras (1.a) para os casos em que outro mecanismo não for definido.

As associações de contingência para uma tarefa podem ser feitas de forma recursiva, de forma que uma tarefa de contingência T' pode também ter outra tarefa T'' de contingência. Desta forma, se uma contingência falhar nada impede que outra tarefa de contingência seja executada na tentativa de resolver a inconsistência.

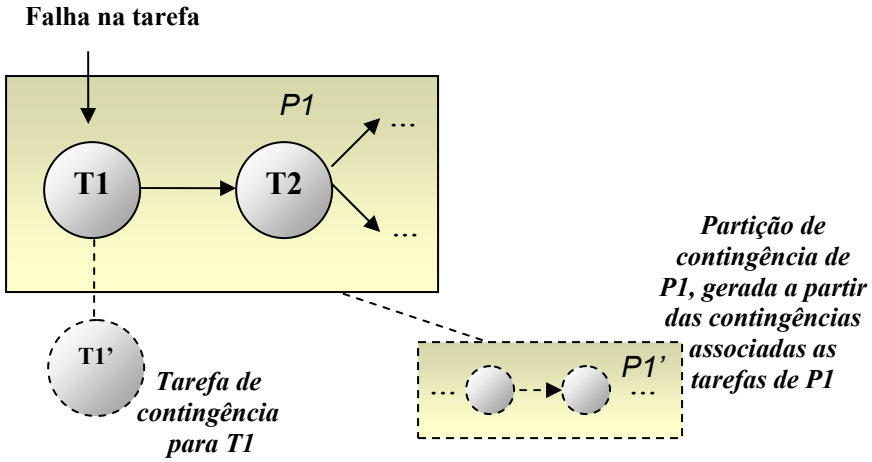

Figura 18 – Tarefas e partições de contingência

Com base nos mecanismos de gerência de falhas apresentados anteriormente é possível enumerar possíveis ações que serão executadas quando ocorrer uma falha. As ações aceitas nessa arquitetura são descritas a seguir. Elas são exemplificadas com base em duas tarefas T1 e T2, onde T1 é uma pré-condição para T2 e T1 é a tarefa que falha. Desta forma, tem-se:

1. Abortar T1 e NÃO liberar T2.

Todas as tarefas pertencentes a todos os caminhos do grafo representativo do *workflow* acessíveis a partir de T2 não serão mais executadas. Estas tarefas serão bloqueadas. Além disso, todos os fluxos de entrada e saída das tarefas T1, T2 e das acessíveis a partir de T2 serão excluídos do *workflow*. Na Figura 19 pode ser visto um exemplo desta ação.

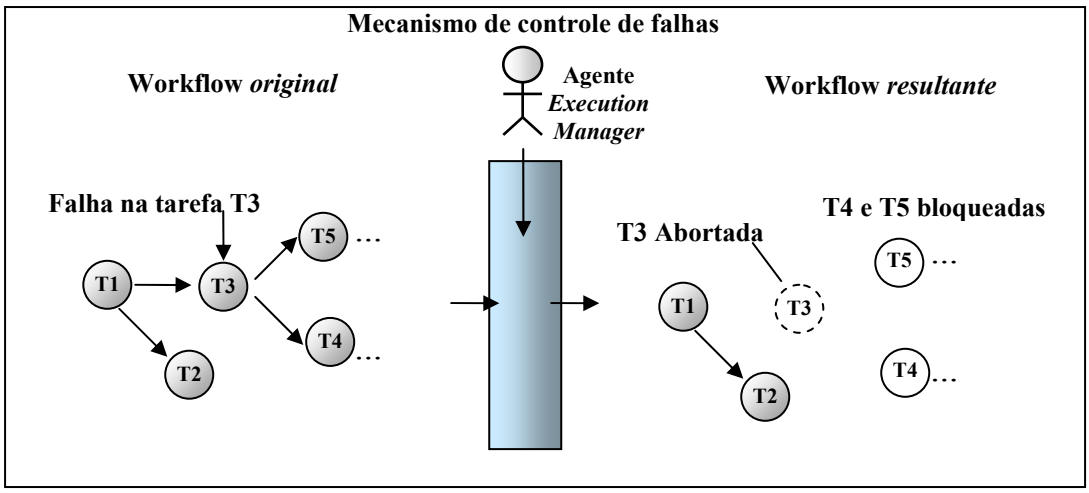

Figura 19 – Aborta tarefa e bloqueia as sucessoras

2. Abortar T1 e liberar T2.

O que ocorre aqui é uma quebra de pré-requisito. T2 será executada sem que a tarefa T1, antes definida como pré-condição, tenha sido efetuada. Todas as tarefas acessíveis diretamente a partir de T1 serão liberadas em relação à pré-condição geradas por T1. Na Figura 20 pode ser visto um exemplo desta ação.

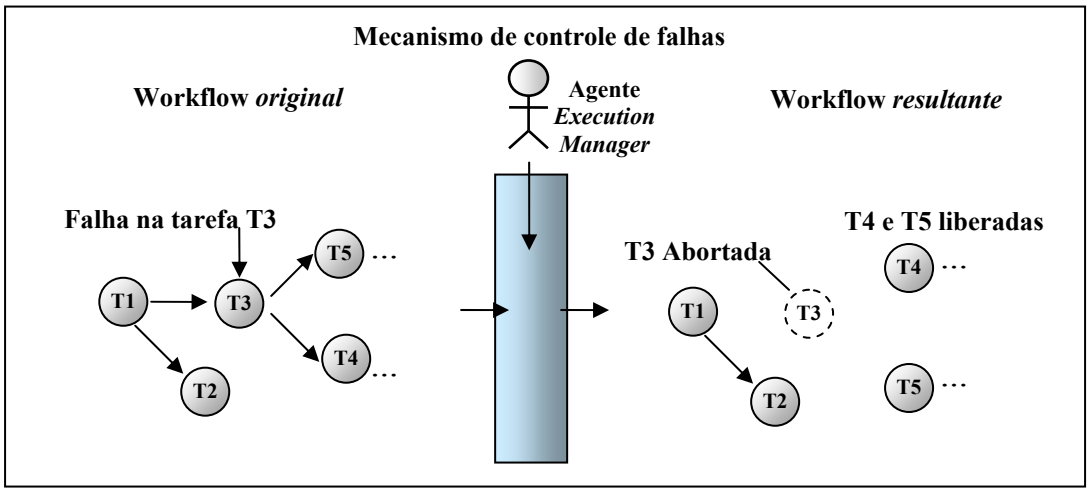

Figura 20 – Aborta tarefa e libera as sucessoras

3. Disparar a tarefa de contingência de T1.

Neste caso, nada é alterado em relação a T2. Esta tarefa espera pela execução da contingência T1' como se estivesse esperando por T1. O exemplo dessa ação pode ser observado na Figura 21.

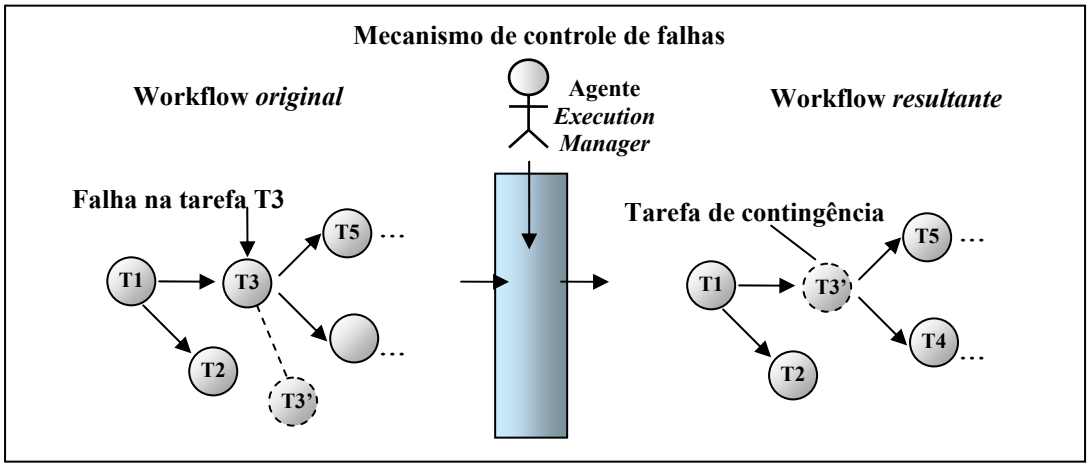

Figura 21 – Dispara a tarefa de contingência

4. Disparar a partição de contingência de P1.

Quando uma partição inteira deve ser trocada pela partição de contingência todas as 3 ações anteriores podem ser usadas. Todas as tarefas serão trocadas pelas suas contingências, a não ser que elas devam ser abortadas ou bloqueadas. A partição de contingência será construída de acordo com a definição de controle de falhas especificada para cada tarefa. O conjunto dessas definições pode resultar em um outro *sub-workflow* completamente diferente. Por exemplo, se uma tarefa T1 falha e aborta T2 que tem uma contingência T2', então a ação de abortar prevalecerá sobre a ação de usar a contingência T2'. Um exemplo dessa ação pode ser observado na Figura 22.

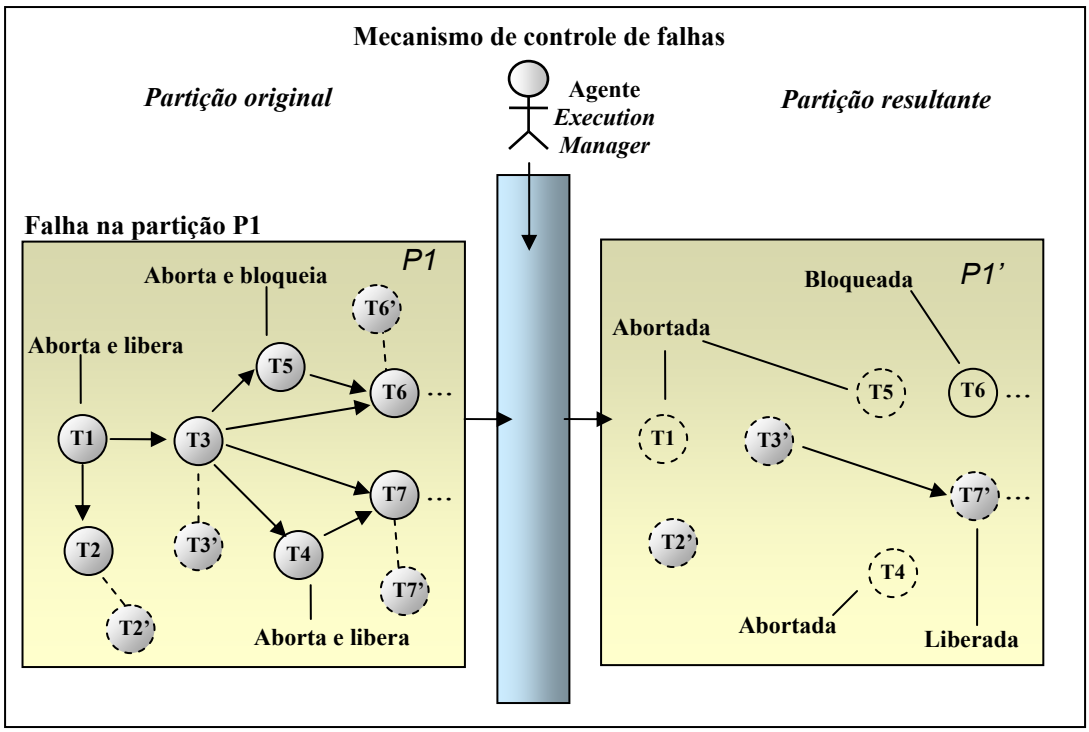

Figura 22 – Dispara a partição de contingência

Os mecanismos de gerência de falhas propostos nessa arquitetura permitem ao usuário especificar ações a serem tomadas caso ocorram falhas no sistema, fornecendo, como medida de correção, uma saída alternativa para as possíveis inconsistências que viriam a ocorrer caso um parceiro se desconecte permanentemente.

5.3. Uma modelagem para coreografia

O objetivo dessa seção é levantar os principais problemas e possíveis soluções associados aos sistemas de gerência de *workflow* em ambientes com desconexão com suporte à coreografia. Como já citado (seção 5.1), a modelagem da coreografia pode ser vista como uma extensão da modelagem projetada para a orquestração. A arquitetura da coreografia é a soma da arquitetura para a orquestração mais a comunicação parceiro-parceiro, que é especifico deste tipo de coordenação.

Todos os componentes citados na seção anterior estão também presentes aqui. Além disso, as características adicionais "particionamento" e "*triggers* de

emergência" apresentadas devem funcionar da mesma forma. A mudança está mais associada aos mecanismos de controle de gerência de falhas, que agora levam em conta a comunicação direta entre parceiros.

Problemas de inconsistência de dados tornam-se maiores com a comunicação parceiro-parceiro. A arquitetura deixa de conter um ponto central responsável por toda comunicação, o que dificulta o controle das trocas de mensagens. Mensagens entregues podem ser dadas como perdidas mais facilmente.

Veja que agora o caminho das mensagens não segue mais o sentido parceiro - controlador central – parceiro. O controlador central centralizava todas as informações e também provia todos os serviços de comunicação. Para facilitar, foi assumido que o controlador central não era um ponto de falha (Suposição C). Agora um parceiro deve também funcionar como um servidor na rede, para que outro parceiro possa se conectar a ele e transmitir as informações necessárias. Fazer um parceiro funcionar como um servidor é o primeiro ponto que dificulta a implementação dessa arquitetura.

Diante desse novo cenário, surgem situações que podem gerar outros problemas como, por exemplo:

- Um parceiro P1 envia uma mensagem para um parceiro P2 que não responde por está desconectado. Entretanto P2 se reconecta logo em seguida. P1 pode considerar que P2 não faz mais parte do processo.

- Um parceiro P1 possui as configurações da rede para acessar P2. P2 se re-conecta com novas configurações. Em seguida, P1 tenta envia uma mensagem para P2 e não obtém sucesso, apesar dele está conectado.

Para contornar problemas como estes, é sugerido aqui que o servidor mantenha uma lista de parceiros ativos no sistema. Essa lista deve ser atualizada de tempos em tempos, especificado através de um parâmetro de tempo denominado de *connectionTime*. Cada parceiro tem conhecimento do *connectionTime* e deve sempre se re-conectar com o servidor após este tempo para revalidar sua participação no processo, caso contrário, ele deve ser desconsiderado do processo (Figura 23).

No momento da reconexão, novas configurações de rede de um parceiro são atualizadas no controlador central. O controlador central, com sua lista atualizada, deve informar todos os parceiros através de uma mensagem (*broadcast*) de atualização da lista de parceiros.

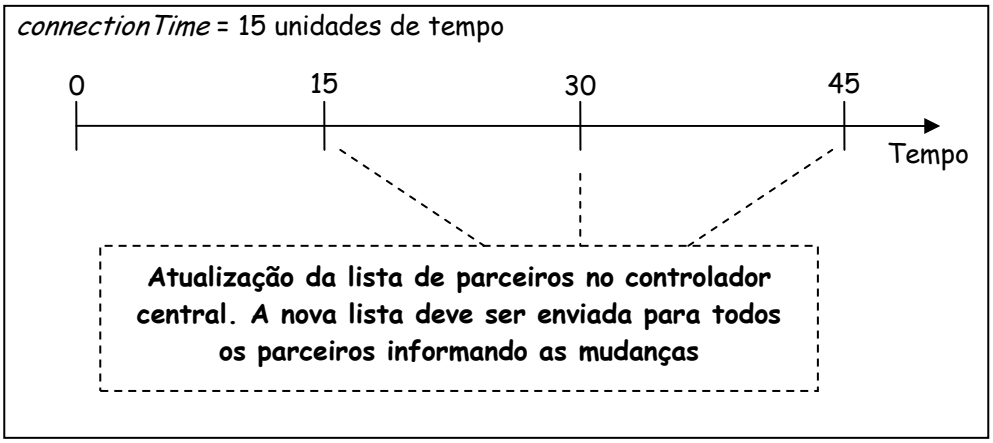

Figura 23 – Atualização da lista de parceiros a cada *connectionTime*

Caso um parceiro realmente se desconecte, não revalidando sua participação junto ao controlador central num determinado intervalo de *connectionTime*, o controlador central deve aplicar os mesmos mecanismos de gerência de falhas apresentadas na seção anterior, abortando ou substituindo tarefas pelas suas contingências. Quando necessário, o controlador central deve reconstruir toda a partição.

Outro controle deve ser feito nessa abordagem, desta vez diretamente nos parceiros. Quando um parceiro P1 envia uma mensagem para outro sucessor P2, o parceiro P2 deve responder a P1 que recebeu a mensagem. P1 só considera que a mensagem foi entregue quando ele receber a confirmação de recebimento. Quando o parceiro P2 não estiver conectado, o parceiro P1 não deve considerá-lo como fora do processo de imediato, visto que sua desconexão pode ser temporária. Desta forma, P1 deve tentar a comunicação com P2 novamente, repetindo sucessivamente até receber uma resposta positiva ou até que o servidor atualize a lista e informe que P2 não faz mais parte do processo. Se P2 possuir novas configurações de rede, P1 deve utilizá-las repetindo o processo de tentativa de comunicação.

Estes problemas são decorrentes principalmente do tipo de conexão, ainda bastante instável, que os PDAs possuem atualmente. Diante dessa realidade, o uso de um servidor faz-se necessário para uma melhor gerência do processo. Com o avanço dessa tecnologia e, consequentemente, uma melhor confiabilidade na conexão dos PDAs, novas abordagens poderão ser sugeridas diminuindo em parte o papel do servidor.

5.4. Conclusão

Este capítulo apresentou uma arquitetura para coordenação de *workflows* em ambientes com suporte à desconexão. A arquitetura, que tem como foco a utilização de dispositivos móveis com fortes características de desconexão, aborda, dentre outros itens, mecanismos para controle de falhas e mecanismos de *triggers* de emergência dentro do processo.

A arquitetura com todos seus componentes e suas características adicionais cobre todos os requisitos especificados anteriormente.