UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MODELOS DE CONTROLE DE ACESSO SENSÍVEIS AO CONTEXTO COM USO DE SMART CONTRACTS

MARCOS VINICIUS MENEZES RODRIGUES DE SOUZA

ORIENTADOR: PROF. DR. HÉLIO CRESTANA GUARDIA

São Carlos – SP

Março/2019

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MODELOS DE CONTROLE DE ACESSO SENSÍVEIS AO CONTEXTO COM USO DE SMART CONTRACTS

MARCOS VINICIUS MENEZES RODRIGUES DE SOUZA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Sistemas Distribuídos e Redes de Computadores
Orientador: Prof. Dr. Hélio Crestana Guardia

São Carlos – SP

Março/2019

Souza, Marcos Vinicius Menezes Rodrigues de

Modelos de Controle de Acesso Sensíveis ao Contexto com uso de Smart Contracts / Marcos Vinicius Menezes Rodrigues de Souza. -- 2019.
118 f. : 30 cm.

Dissertação (mestrado)-Universidade Federal de São Carlos, campus São Carlos, São Carlos
Orientador: Hélio Crestana Guardia
Banca examinadora: Hélio Crestana Guardia, Paulo Matias, Julio Cezar Estrella
Bibliografia

1. Controle de Acesso. 2. Sensibilidade ao Contexto. 3. Blockchain. I. Orientador. II. Universidade Federal de São Carlos. III. Título.

Ficha catalográfica elaborada pelo Programa de Geração Automática da Secretaria Geral de Informática (SIn).
DADOS FORNECIDOS PELO(A) AUTOR(A)
Bibliotecário(a) Responsável: Ronildo Santos Prado – CRB/8 7325

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Marcos Vinicius Menezes Rodrigues de Souza, realizada em 22/03/2019:

Prof. Dr. Helio Crestana Guardia
UFSCar

Prof. Dr. Paulo Matias
UFSCar

Prof. Dr. Julio Cezar Estrella
ICMC/USP

AGRADECIMENTOS

Agradeço primeiramente a Deus por me dar a saúde e condições de realizar este trabalho, a meus familiares pelo seu constante apoio e incentivo, e ao meu orientador Prof. Dr. Hélio pelos ensinamentos e auxílio no desenvolvimento deste trabalho.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

RESUMO

O avanço da computação ubíqua e da Internet das Coisas traz novos desafios aos sistemas de controle de acesso, como a não dependência de unidades centrais consideradas seguras, e a necessidade de poder identificar contextos. Estas características são necessárias para viabilizar os serviços de Internet das Coisas como em *smart homes*, *smart cities*, serviços médicos em hospitais, residências, ou até mesmo no monitoramento móvel de condições de saúde. Nesses contextos de computação distribuída, com interações comumente realizadas na forma de invocações de serviço entre os sistemas participantes, autenticidade e confidencialidade são exemplos de aspectos de segurança a serem providos de forma flexível e escalável. Nesse sentido, este trabalho trata do desenvolvimento de um modelo de controle de acesso que atenda estes desafios, baseando-se na infraestrutura de *blockchain*. Para tal, propõe-se a estruturação de informações e contextos usando o modelo de *smart contracts* para inserção de políticas de acesso e técnicas de *blockchain* para o registro distribuído e seguro das informações. No modelo proposto, recursos (*assets*) são associados às permissões, e suas transferências, devidamente registradas na *blockchain*, são usadas para a validação das tentativas de acesso a serviços. Uma implementação de referência do modelo proposto foi realizada sobre a infraestrutura provida pelo sistema *Hyperledger* e os resultados mostram a viabilidade deste modelo.

Palavras-chave: Controle de Acesso, Sensibilidade ao Contexto, Computação Ubíqua, Blockchain.

ABSTRACT

The constant advances in ubiquitous computing and in the Internet of things (IoT) bring new challenges to access control systems, such as building a mechanism not dependent on a centralized trusted party, and the need to account for contextual data. Such characteristics may be required to enable IoT services in smart homes, smart cities, medical services, or even in mobile computer based health monitoring systems. In these distributed computing environments, direct device to device interactions are commonly based on service requests among the participants, which reinforce the need for flexible and scalable authenticity and confidentiality mechanisms. This work deals with the development of an access control model to address such challenges using a blockchain infrastructure. In the proposed method, services and contextual data are defined using smart contracts, and blockchain logs are used for distributed and trustful information management. Blockchain assets are associated with permissions, and their transfers are used to directly verify access control rights in service invocation requests. A reference implementation of the the proposed model has been developed using the Hyperledger system and the results show the viability of this methodology.

Keywords: Access Control, Context Awareness, Ubiquitous Computing, Blockchain.

LISTA DE FIGURAS

2.1 Esquema de como deve ocorrer a geração do *hash* de um bloco. 25

2.2 Encadeamento dos blocos assegurando o PoW, conforme Nakamoto (2008). . . 26

2.3 *Framework* OM-AM, conforme Sandhu (2000). 28

2.4 *Framework* OM-AM para sistemas RBAC, segundo Sandhu (2000). 28

2.5 Modelo referência de OM-AM conforme Ouaddah, Elkalam e Ouahman (2016). 29

2.6 Relacionamentos no RBAC. 38

2.7 RBAC com restrições. 39

2.8 Relacionamentos nos CAAC. 40

2.9 *Middleware* UbiCOSM. 41

2.10 prox-RBAC conforme Gupta, Kirkpatrick e Bertino (2012). 42

2.11 Controle de acesso com uso de *blockchain*, conforme Zyskind, Nathan e Pentland (2015). 43

2.12 Diagrama de classes do FairAccess, conforme Ouaddah, Elkalam e Ouahman (2016). 44

3.1 Diagrama UML para o CAAC usando smart contracts. 49

3.2 Protocolo da transação *AddResource*, para registros de recursos. 52

3.3 Protocolo da transação *ComposeContext*, para atualização de contexto. 53

3.4 Protocolo da transação *RequestAccess*, para requisição de permissão de acesso. 54

3.5 Uso de JSON para modelagem de dados. 56

3.6 Definição de um *concept* representando localização. 56

3.7 Diferença entre os modelos. 57

3.8 Instalação de *chaincode* através de ferramenta da IBM. 58
3.9 Documento de configuração de peers. 59
3.10 Documento de configuração de ordering peers. 60
3.11 Documento de configuração inicial da rede com o *genesis block*. 62
3.12 Propriedades necessárias para instalar um *chaincode*. 64
3.13 Exemplo de função em um chaincode. 65
3.14 Exemplo de estrutura lógica de uma rede. 65
3.15 Definição de política de endosso na instanciação de chaincode, através de ferramenta da IBM. 67
3.16 Política de endosso definida através de JSON. 67
3.17 Proposta de transação encaminhada aos *peers* pelo cliente. 69
3.18 Resposta dos *peers* quando a transação é endossada. 70
3.19 Resposta dos *peers* quando a transação é inválida. 71
3.20 Propagação do endosso. 72
3.21 *Ordering service* reentregando as transações para os *peers*. 72
3.22 Fluxo comum de uma transação. 74
3.23 Diferença entre o *PeerLedger* e o *ValidatedLedger*. 76
4.1 Implementação da modelagem proposta. 79
4.2 Código *JavaScript* para a função *AddResource*. 81
4.3 Código *JavaScript* para a função *ComposeContext*. 82
4.4 Código *JavaScript* para a função *RequestAccess*. 84
4.5 Criação de participante proprietário de recursos. 86
4.6 Registro dos participantes proprietários de recursos. 86
4.7 Criação de participante requerente. 87
4.8 Registro de participantes requerentes. 88
4.9 Cradastro de recurso. 88

4.10 Registro de membro atualizado. 89
4.11 Registro da operação *AddResource* efetuada. 90
4.12 Atualização de contexto. 91
4.13 Registro da operação ComposeContext efetuada. 92
4.14 Requisição para acesso. 93
4.15 Registro da operação RequestAcess efetuada. 94
4.16 Fluxo de comunicações proposto. 96
4.17 Função *JavaScript* para cadastro de recurso. 97
4.18 Função *JavaScript* para criação de *Assets* de permissão. 98
4.19 Função *JavaScript* para transferência de *Assets* de permissão. 98
4.20 Função *JavaScript* para obtenção de acesso. 99
4.21 Cadastro de recurso/serviço. 101
4.22 Registro do cadastro de recurso/serviço efetuado. 102
4.23 Criação de *Asset* para permissão. 103
4.24 Registro da criação de Asset efetuada. 104
4.25 Transferência de *Asset* de permissão. 105
4.26 Registro da transferência efetuada. 106
4.27 Requisição de acesso. 107
4.28 Registro da requisição efetuada. 108
4.29 *Query* de verificação de tranferência. 110

LISTA DE TABELAS

2.1 Comparativo de trabalhos relacionados. 36

LISTA DE ABREVIATURAS E SIGLAS

ABAC Attribute Based Access Control

AMP Authorization Manager Point

API Application Programming Interface

BFT Byzantine Fault Tolerance

blob Binary Large Object

blockno Block Number

BLONDiE Blockchain Ontology with Dynamic Extensibility

CAAC Context-Aware Access Control

CARMEN Context-Aware Resource Management

chaincodeID Identificador de Chaincode

clientID Identificador de Cliente

clientSig Assinatura de Cliente

CPU Central Processing Unit

CTO Linguagem de modelagem baseada no framework Concerto

D2D Device to Device

DAC Discretionary Access Control

DHT Distributed Hashtable

DNS Domain Name Service

EHR Eletronic Health Records

epID Identificador de Endorsing Peer

eTRON Entity TRON (The Real-time Operating system Nucleus)

FM Feature Monitor

GEO-RBAC Geo-spatial RBAC

IBM International Business Machines

ID Identificador

IoT Internet of Things

IP Internet Protocol

JSON JavaScript Object Notation

KB Kilo Bytes

LBAC Location-Based Access Control

LD Lógica Descritiva

LRBAC Location-aware Role-Based Access Control

MAC Media Access Control

MB Mega Bytes

MSP Membership Service Provider

NS Namespace

OM-AM Objetivos, Modelos, Arquitetura e Mecanismos

OPM Object-Process Methodology

OWL Web Ontology Language

PA Permission Assignment

peerSig Assinatura do Peer

PO-SAAC Purpose-Oriented Situation-Aware Access Control

PoW Proof of Work

prevhash Previous Hash

Prox-RBAC Proximity-RBAC

RBAC Role Based Access Control

RDF Resource Description Framework

RMEs Registros Médicos Eletrônicos

SDK Software Development Kit

seqno Sequence Number

SitBAC Situation-Based Access Control

SMR State Machine Replication

SoD Separation of Duty

STAC Spatial-Temporal Access Control

tx Transação

UA User Assignment

UbiCOSM Ubiquitous Context-based Security Middleware

UML Unified Modeling Language

vBlock Validated Block

VLedger Validated Ledger

XML eXtensible Markup Language

SUMÁRIO

CAPÍTULO 1 – INTRODUÇÃO **16**

1.1 Problema 17

1.2 Objetivos 18

1.3 Hipótese 18

1.4 Organização 18

CAPÍTULO 2 – TECNOLOGIAS E CONCEITOS ENVOLVIDOS **20**

2.1 Controle de Acesso 20

2.2 Ontologias e OWL 23

2.3 Blockchain 24

2.3.1 Smart Contracts 25

2.3.2 Proof-of-Work 25

2.3.3 Tolerância a Falha Bizantina 26

2.4 Camadas OM-AM 27

2.5 Trabalhos Relacionados 29

2.6 Discussão sobre controle de acesso sensível ao contexto 37

CAPÍTULO 3 – MODELO PARA CONTROLE DE ACESSO USANDO HYPERLEDGER **45**

3.1 Arquitetura de um mecanismo de controle de acesso usando Hyperledger . . . 46

3.2 CAAC usando smart contracts 48

3.2.1 User 50
3.2.2 Resource 50
3.2.3 AccessPolicy 50
3.2.4 Context 51
3.2.5 Access 51
3.2.6 Transactions 51
3.3 CAAC com smart contracts baseado em transferências de recursos 55
3.4 Estruturas do sistema Hyperledger 58
3.4.1 Nós 58
3.4.2 Estado 61
3.4.3 Ledger 61
3.4.4 Chaincodes 63
3.5 Transações e Registros usando Hyperledger 68
3.5.1 Formato de proposta de transação 68
3.5.2 Simulação de transação e assinatura de endosso 69
3.5.3 Coleta de assinaturas de endosso e propagação pela rede 71
3.5.4 Entrega da transação aos peers 72
3.5.5 Especificação de políticas 74
3.5.6 Avaliação das transações sob as políticas do endosso 75
3.5.7 Livro de registros válidos 75
3.5.8 Checagem do PeerLedger 76

CAPÍTULO 4 – RESULTADOS E DISCUSSÕES **78**

4.1 Metodologia 78
4.2 Implementação do CAAC usando smart contracts 78
4.2.1 Execuções e resultados 85

4.3 Implementação do CAAC com smart contracts baseado em transferências de recursos 94

4.3.1 Execuções e resultados 99

4.4 Aspectos operacionais das propostas usando CAAC 108

CAPÍTULO 5 – CONCLUSÕES **111**

5.1 Trabalhos futuros 114

REFERÊNCIAS **115**

Capítulo 1

INTRODUÇÃO

O acesso à informação deve ser controlado, a fim de garantir privacidade e segurança. Diversas podem ser as formas adotadas para o controle de acesso e como serão suas políticas e regras.

Alguns modelos de controle de acesso destacam-se, como o RBAC (*Role-Based Access Control*), baseado em papéis e o ABAC (*Attribute-Based Access Control*), baseado em atributos.

RBAC (FERRAIOLO; KUHN, 1992) é um modelo que utiliza a relação entre papéis assumidos por um usuário e os recursos que ele poderá acessar. É bastante comum de ser aplicado pela sua fácil associação aos modelos de cargos em instituições diversas. Ou seja, ele se enquadra em situações em que as permissões de acesso sejam definidas pelo cargo ocupado pelo profissional. Como exemplo, podemos tomar uma loja em que todo funcionário com cargo de vendedor pode acessar as informações do estoque de produtos.

Já no ABAC (HU et al., 2014), o acesso é concedido àqueles que possuírem em seu perfil os atributos necessários para acessar um determinado recurso. Para cada recurso, pode-se solicitar um ou mais atributos para que o acesso seja garantido. Esse modelo propicia uma granularidade mais fina nas regras de acesso, mas acarreta em maior complexidade de gerenciamento dessas regras.

Apesar de o ABAC permitir uma granularidade mais fina, a sua complexidade de implementação e manutenção, dificulta sua adoção. Assim, o RBAC é mais adotado, surgindo extensões para atender necessidades adicionais.

O conceito de contexto e suas implicações são explorados em diversas áreas de conhecimento. Em computação, contexto é qualquer informação que possa ser usada para caracterizar o estado, situação, ou atividade de uma entidade, ou ainda do ambiente no qual a entidade opera

(DEY, 2001; TONINELLI et al., 2006)

A sensibilidade ao contexto (*context-awareness*) é considerada como essência da computação ubíqua (KUHN; COYNE; WEIL, 2010). Uma característica importante do contexto é a sua relação com a segurança, sendo que algo considerado seguro em um contexto, pode não ser seguro em outro contexto.

Muitos trabalhos foram feitos para identificação de informações que podem ser usadas para contexto, qual a estrutura dessas informações, como representá-las e como explorá-las em aplicações específicas. Informações como tempo, localização, condições gerais do ambiente, intenções do usuário, estado da rede e estado de execução, entre outras, são possíveis de serem utilizadas para contexto.

Com o desenvolvimento das tecnologias *mobile*, da internet das coisas e da computação ubíqua de forma geral, houve um aumento na quantidade de formas e locais possíveis para acessar informações, consequentemente, aumentando a variedade de situações nas quais as informações serão acessadas. Mas, uma informação não estará segura se não for visualizada apenas pela pessoa certa, no momento certo e nas circunstâncias certas.

A habilidade computacional de perceber contextos, seja por processamento de informações de sensores, ou por técnicas de inferência, possibilita uma melhor representação de situações do mundo físico. Dessa forma, pode-se utilizar o contexto para auxiliar nas tomadas de decisões e no gerenciamento de informações, como nos procedimentos de controle de acesso.

1.1 Problema

O controle de acesso em diversos sistemas foca em aplicar um nível de granularidade de acesso, de acordo com as políticas definidas. O maior desafio enfrentado nesses sistemas é prover autorização, conectando vários sistemas heterogêneos que são integrados em ambientes de computação ubíqua.

Parece claro que o aumento do número de ambientes com interações diretas entre dispositivos, como D2D e IoT, através de invocações de serviços, trouxe necessidade de adequação dos serviços prestados.

Esses ambientes precisam da capacidade de identificação mútua entre os nós que interagem, além da confiança de que os serviços oferecidos realizam o que se espera.

Também é necessária a garantia de que as informações envolvidas na comunicação sobre os estados e recursos são confiáveis.

A confirmação de localização pelo dispositivo de acesso também deve se tornar mais comum para autenticação multifator, e para decisões de novas políticas necessárias com o crescimento da Internet das Coisas.

Não apenas a localização, mas também outras variáveis de contexto, precisam ser levadas em consideração no funcionamento desses sistemas. É preciso garantir que estas informações sejam confiáveis para satisfazer as condições de operação sem comprometer a segurança.

Modelos de controle de acesso que não sejam dependentes de uma unidade central, assumida como segura, também devem se tornar mais comuns. Isso deve ser feito de forma a garantir a privacidade e autenticidade dos dados.

Portanto, se faz necessário o desenvolvimento de um modelo de controle de acesso sensível ao contexto e adaptado aos desafios presentes em ambientes de computação ubíqua.

1.2 Objetivos

A partir da demanda identificada, o trabalho busca identificar ferramentas e técnicas que poderão ser utilizadas na solução do problema.

Ao encontrar as ferramentas e técnicas que atendam às necessidades apresentadas anteriormente, desenvolveu-se um modelo de controle de acesso sensível ao contexto.

O modelo conta com estruturação das informações, modelagem de contextos e definição das políticas e uma implementação como prova de conceito.

1.3 Hipótese

O modelo de confiança distribuío provido pelas plataformas de blockchain pode ser aplicado no controle de acesso em ambientes de computação ubíqua, e smart contracts podem apoiar a validação do cumprimento de políticas e requisitos para acesso a recursos sensível ao contexto.

1.4 Organização

Essa dissertação está organizada da seguinte forma: no Capítulo 2, apresenta-se as tecnologias e conceitos envolvidos no tema desta pesquisa, bem como a apresentação de trabalhos

relacionados e uma breve discussão sobre eles; no Capítulo 3, encontra-se a apresentação teórica das propostas elaboradas, como também uma apresentação de aspéctos técnicos específicos da plataforma *blockchain* utilizada; no Capítulo 4, apresentam-se implementações e testes para validação das funcionalidades abordadas, além de uma discussão sobre aspectos operacionais das propostas; no Capítulo 5, apresentam-se as conclusões obtidas neste trabalho, bem como a recomendação de trabalhos futuros.

Capítulo 2

TECNOLOGIAS E CONCEITOS ENVOLVIDOS

Neste capítulo são apresentadas as tecnologias e conceitos envolvidos no tema desta pesquisa, bem como trabalhos relacionados e uma breve discussão sobre Controle de Acesso Sensível ao Contexto.

2.1 Controle de Acesso

O mecanismo primário de proteção à privacidade das informações é o controle de acesso. Sistemas de registro eletrônico de saúde (EHR - *Electronic Health Records*), são um exemplo da necessidade do controle de acesso para proteger informações críticas/sensíveis dos pacientes (JAYABALAN; O'DANIEL, 2016)

O metamodelo mais usual para controle de acesso é o baseado em papéis (RBAC - *Role-based access control*), com extensões de acordo com a necessidade. Para contornar problemas de granularidade e capturar informações de contexto para privilégios de acesso, utilizam-se extensões com aspectos espaciais, temporais, probabilísticos, dinâmicos, e semânticos. Dados como localização e tempo são comumente utilizados em autenticação multifator (JAYABALAN; O'DANIEL, 2016).

Abordagens que utilizam informações de localização são bastante exploradas, recebendo nomes como: GEO-RBAC, LRBAC, LBAC, Prox-RBAC, entre outros. Informações de localização não são definidas apenas pela localização absoluta do usuário que requer o acesso, mas também podem envolver a sua posição relativa a outras pessoas, usuários ou não, dependendo do caso (GUPTA; KIRKPATRICK; BERTINO, 2012).

Este uso de posições relativas, entre usuário e outras pessoas, é o que possibilita o chamado princípio da separação de deveres (*principle of separation of duty*), classificado como uma das

cinco metas dos controles de acesso baseados em localização. As outras metas são: princípio do privilégio mínimo (*principle of least privilege*), capacidade de responsabilização (*accountability*), capacidade de manutenção (*maintainability*) e usabilidade (*usability*) (CLEEFF; PIETERS; WIERINGA, 2010). Estas metas são importantes e poderiam ser consideradas em outros modelos de controle de acesso.

O princípio da separação de deveres é a maneira de controlar as permissões de acesso e determinadas ações de um usuário, baseando-se na ausência ou presença de outras pessoas. Em outras palavras, a responsabilidade sobre uma ação é dividida por ao menos dois usuários, aumentando a confiabilidade por verificação (CLEEFF; PIETERS; WIERINGA, 2010). Neste princípio, a localização exata do usuário pode não ser tão importante, mas a sua posição relativa aos outros é essencial (GUPTA; KIRKPATRICK; BERTINO, 2012).

Como exemplo de permissão associada à ausência, pode-se considerar o cenário com dois fiscais que não poderiam estar acessando os mesmos documentos, no mesmo local, para impedir que um influencie na decisão do outro. Ainda por ausência, há os cenários em que funcionários de alguma organização não podem acessar documentos sigilosos se houver outras pessoas no mesmo local.

Já nos casos de permissão associada à presença, existem os cenários onde usuários só podem realizar determinadas ações no sistema se um supervisor estiver presente.

O princípio do privilégio mínimo retrata a importância de manter os usuários apenas com as permissões realmente necessárias. Isso pode evitar problemas com erros de acessos indevidos, pelo excesso de autorizações; prevenir ações abusivas de agentes maliciosos; e diminui o prejuízo causado em possíveis invasões, como por roubo de senha (CLEEFF; PIETERS; WIERINGA, 2010).

Capacidade de responsabilização é o termo usado para a capacidade de manter a responsabilidade dos usuários sobre suas ações, por *logging* e monitoramento. Isso pode inibir ações indevidas e, em alguns casos, permite a recuperação após ações ilegais (CLEEFF; PIETERS; WIERINGA, 2010).

Capacidade de manutenção refere-se à necessidade de manter as autorizações dos usuários atualizadas. Deve haver sincronismo entre a descrição dos trabalhos, cargos, ou papeis, dos usuários e as autorizações que eles recebem no sistema. Se o sincronismo não for feito de maneira eficiente, o princípio do privilégio mínimo pode ser rompido (CLEEFF; PIETERS; WIERINGA, 2010).

Usabilidade é a necessidade de não sobrecarregar os usuários pelas medidas de segurança

do sistema. Um sistema que exija trocas de papéis excessivas para autorizações ou muitas senhas, não será bem recebido pelos usuários, podendo ter seu uso evitado (CLEEFF; PIETERS; WIERINGA, 2010). Portanto, o sistema deve ter usabilidade próxima ao intuitivo, com exigências coerentes, para ser bem recebido pelos usuários e atingir seu objetivo.

Estas metas foram idealizadas para satisfazer as necessidades de sistemas de controle de acesso, critérios de qualidade e sistemas pervasivos (CLEEFF; PIETERS; WIERINGA, 2010).

Nota-se, também, que estes sistemas de controle de acesso remetem aos conceitos, requisitos e características da computação ubíqua.

ElShafee (2016) destaca requisitos da computação ubíqua, como a necessidade da interação e participação do sistema com os parâmetros do ambiente e a sensibilidade ao contexto para otimização de operações. Também discorre sobre os dispositivos envolvidos, como computadores, dispositivos móveis e vestíveis, acessórios e implantes computadorizados, sendo que estes devem estar em rede, distribuídos e acessíveis de forma transparente.

Sistemas ubíquos em geral são orientados a serviços, utilizam sensores e dispositivos em rede sem fio, além de terem natureza distribuída e baseados em semântica. Para poder reconhecer os estados do usuário e do ambiente, é necessário que um sistema ubíquo seja minimamente intrusivo, sendo capaz então de se comportar de acordo com a situação (ELSHAFEE, 2016).

Qualquer informação que possa ser utilizada para caracterizar o estado de uma entidade, ou do ambiente onde opera, é considerada informação de contexto (TONINELLI et al., 2006).

Muitas linguagens já foram desenvolvidas para processamento computacional de semântica (KAYES; HAN; COLMAN, 2014). No entanto, as técnicas de modelagem por ontologias têm se provado como mais adequadas para modelar contextos dinâmicos (BETTINI et al., 2010; RIBONI; BETTINI, 2011).

A modelagem baseada em ontologias permite que outros fatos possam ser inferidos a partir de fatos já conhecidos usando regras de inferência em uma máquina de inferência (KAYES; HAN; COLMAN, 2014).

A linguagem OWL (*Web Ontology Language*) tem sido considerada a escolha mais prática pela sua relação custo-benefício entre a complexidade computacional para inferências e a expressividade proporcionada (RIBONI; BETTINI, 2011).

OWL pode ser usado na definição de políticas de sistemas, tendo vantagem crítica em ambientes distribuídos que envolvam a coordenação entre várias organizações. Outra vantagem é a facilidade de tradução das políticas expressas em OWL para outros formalismos, devido ao

seu embasamento em lógica (FININ et al., 2008).

A partir de 2012, OWL começou a substituir XML (*eXtensible Markup Language*), no uso para o gerenciamento das políticas de acesso, visto que essa linguagem pode modelar representações semânticas e suporta relacionamentos complexos em um maior nível (JAYABALAN; O'DANIEL, 2016).

Ontologias OWL são usadas para definição e processamento de políticas, e definição de papéis em situações de emergência. Existe grande diversidade de parâmetros coletados (contexto) para definição de políticas de acesso, e na forma como os dados são utilizados. Em acessos emergenciais, parâmetros temporais são comumente considerados (JAYABALAN; O'DANIEL, 2016).

As ontologias OWL são usadas na representação de modelos de controle de acesso, como o RBAC (FININ et al., 2008), possibilitando extensões que consideram o contexto das situações, através da especificação de políticas e incorporando atributos dinâmicos (KAYES; HAN; COLMAN, 2014).

Novas abordagens também têm apresentado o uso de *blockchain* no controle de acesso. Zyskind, Nathan e Pentland (2015) apresentam uma forma de administrar o controle de acesso sem a necessidade de confiança em terceiros, usando *blockchain*. Para tanto, implementam um serviço que armazena dados criptografados, enquanto registram-se ponteiros sobre o *blockchain*.

MedRec (AZARIA et al., 2016) e FairAccess (OUADDAH; ELKALAM; OUAHMAN, 2016) são exemplos de *frameworks* desenvolvidos para modelos de controle de acesso baseado em *blockchain*.

Essas abordagens visam a aproveitar princípios e práticas já testados no ambiente das criptomoedas (*crypto currencies*). Com isso, o *blockchain* traz para o controle de acesso a possibilidade de uma arquitetura descentralizada, distribuída, auditável, transparente e robusta, incrementando a segurança e privacidade. Essas características tornam tal tipo de abordagem promissora nos ambientes de Internet das Coisas (IoT - *Internet of Things*) e *Big Data*, com aplicação em serviços de cuidados de saúde e *Smart Cities* (HASHEMI et al., 2016).

2.2 Ontologias e OWL

A definição de políticas de acesso e suas representações são essenciais em um modelo de controle de acesso. *Frameworks* com uso de XML foram e ainda são utilizados nessas

definições de políticas de acesso. Mas nos últimos anos, as linguagens de ontologias, como OWL, passaram a atrair interesse de pesquisadores em algumas áreas de aplicação dos controles de acesso (JAYABALAN; O'DANIEL, 2016).

Linguagens de ontologia permitem uma melhor representação de estruturas complexas, como relacionamentos múltiplos entre objetos. Além disso, a OWL se torna mais atrativa pela capacidade de extração de conhecimento semântico a partir dos dados, facilitando o uso de sensibilidade ao contexto.

Outra aplicação para a OWL é a modelagem de *frameworks* de interoperabilidade. Sua forma de validação de políticas permite a troca de informações entre diferentes entidades, sem violação de privacidade.

Ontologias também são aplicadas nos problemas de Web Semântica, além de surgir em propostas para Web 3.0.

Ugarte (2017) apresenta uma ontologia para descrever a estrutura de *blockchain*, do Bitcoin e do Ethereum, nomeada como BLONDiE (*Blockchain Ontology with Dynamic Extensibility*). Discorre também sobre a necessidade do desenvolvimento do que chama de *Semantic Blockchains*, como plataforma para criação da Web 3.0. Tal necessidade seria devido aos benefícios de obter raciocínio semântico nos registros distribuídos do blockchain, junto a confiabilidade nas transações, ou interações pelos *smart contracts*, em uma plataforma ser homogênea, contrária a *web* atual.

Como exemplo do que o *blockchain* pode oferecer para a web, pode-se citar o Namecoin. O Namecoin é um sistema de registro de informações, baseado no *blockchain* do Bitcoin, que permite a desintermediação de provedores DNS (*Domain Name Service*), que são um dos últimos elementos centralizadores na *web* moderna (ENGLISH; AUER; DOMINGUE, 2016).

2.3 Blockchain

O uso de criptografia de chave pública para criação de uma cadeia de registros datados, que aceita apenas adição de novos blocos e não permite alterações nos já existentes, é chamado de *blockchain*.

O *blockchain* foi primeiramente apresentado como um registro público e descentralizado para transações financeiras, no artigo do Bitcoin(NAKAMOTO, 2008). Mas, sua capacidade começou a atrair interesse para aplicação em outras áreas, pois traz a possibilidade de uma arquitetura descentralizada, distribuída, auditável, transparente e robusta, incrementando a segurança

e privacidade.

Uma das áreas onde o *blockchain* vem sendo aplicado é no armazenamento e controle de acesso de informações.

2.3.1 Smart Contracts

A rede *blockchain* apresentada com o Bitcoin trabalha com um conceito simples de transações, focadas na transferência de valores entre contas. Outras plataformas blockchain expandiram o conceito de transações de acordo com suas aplicações.

A Ethereum apresentou uma plataforma *blockchain* mais generalizada expandindo as transações para contratos complexos que passaram a ser chamados de *smart contracts*, em forma de programas Turing-completos. Dessa forma, as transações passam a ter descrições algorítmicas do *smart contract*, representando as operações computacionais que serão feitas no blockchain(WOOD, 2014; ENGLISH; AUER; DOMINGUE, 2016).

Smart contracts passaram a ser implementados em outras plataformas *blockchain*, como no Hyperledger Fabric, do projeto Hyperledger(CACHIN et al., 2016).

2.3.2 Proof-of-Work

Quando o Bitcoin foi apresentado em 2008, propôs o uso de um sistema de *proof-of-work* (PoW) na implementação de um servidor de marca temporal (ou estampa de tempo, do inglês *timestamp*), adotando uma versão similar ao Hashcash apresentado por Back (2002). O procedimento envolve a busca de um valor (chamado de nonce) que, quando passado por uma função *hash* junto ao bloco, o *hash* gerado comece com um número específico de *bits* zero.

A Figura 2.1 mostra como deve ser gerado o *hash* do bloco. O valor *nonce* será alterado até que a função *hash* retorne um valor *hash* com os *bits* zero necessários.

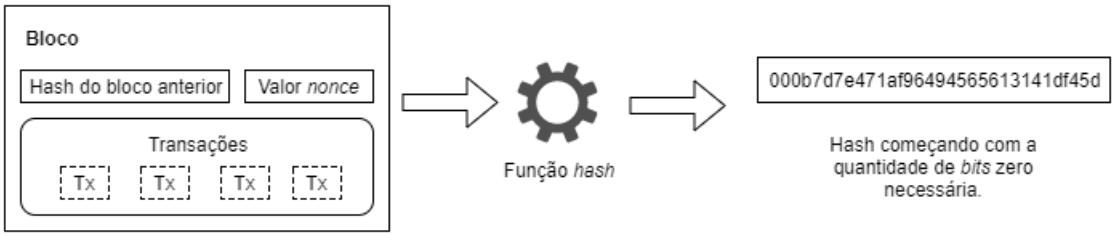

Figura 2.1: Esquema de como deve ocorrer a geração do *hash* de um bloco.

O trabalho necessário para encontrar o valor correto para composição do *hash* do bloco, com

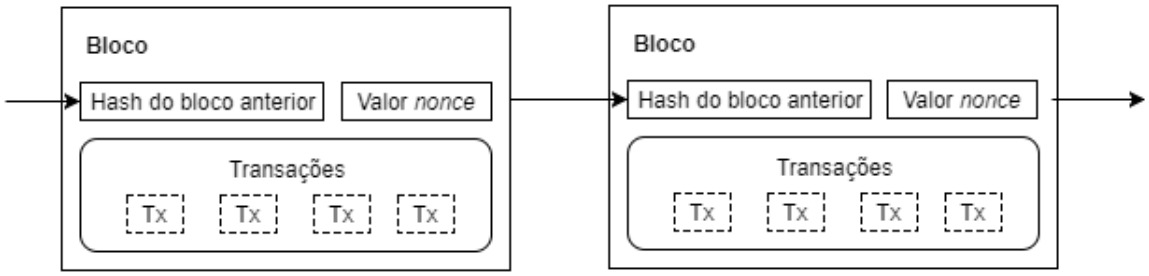

Figura 2.2: Encadeamento dos blocos assegurando o PoW, conforme Nakamoto (2008).

o número de *bits* zero solicitados, é exponencial em relação a esse número de zeros. No entanto, pode ser verificado executando um único *hash*. Após o valor que gera um *hash* satisfatório ser encontrado, o bloco pode ser colocado no registro e novos blocos poderão ser encadeados depois dele (NAKAMOTO, 2008). A Figura 2.2 mostra o esquema de encadeamento dos blocos.

Considerando que qualquer alteração no bloco altera o valor do *hash*, seria necessário um retrabalho de cálculo do valor que gera o *hash* com a quantidade de *bits* zero necessária. Como os blocos são encadeados, a alteração do bloco exige o retrabalho no *hash* de todos blocos seguintes para que seja mantida a consistência do *blockchain*.

No caso de existência de mais de uma cadeia de blocos, por consenso considera-se a maior como a correta. Isso ocorre pelo fato de que um agente mal intencionado que altere algum bloco passado, terá que refazer o cálculo deste bloco e de todos os seguintes. Sendo assim, tal agente só conseguiria alcançar a cadeia original caso possua poder em CPUs maior que o restante da rede operando com o *blockchain*, já que a chance de um atacante mais lento alcançar, diminui exponencialmente a cada novo bloco adicionado(NAKAMOTO, 2008).

Outras plataformas *blockchain*, além do bitcoin, também utilizam alguma forma de PoW, como é o caso da plataforma Ethereum (WOOD, 2014).

2.3.3 Tolerância a Falha Bizantina

A confiança cada vez maior em serviços de informação online traz consigo um aumento na complexidade dos serviços oferecidos e dos softwares envolvidos. Consequentemente, o número de erros em *softwares* também aumenta. Tais erros podem fazer que nós falhos apresentem comportamento arbitrário, também chamado de Bizantino. Ataques maliciosos também podem gerar comportamento Bizantino, tornando algoritmos de tolerância a falha bizantina (BFT) cada vez mais importantes.

Castro e Liskov (2002) apresentaram um algoritmo de replicação de máquinas de estado

(SMR - *State Machine Replication*) para tolerância a falhas bizantinas, que foi o primeiro a ser seguro em sistemas assíncronos como a internet.

Uma característica de SMR é fazer que toda requisição de um cliente seja atendida através de réplicas. Isso requer a implementação de um protocolo *broadcast* com ordenação total, que é equivalente ao problema de consenso. Dessa forma, a solução de problemas de consenso está no núcleo de qualquer protocolo distribuído de SMR (SOUSA; BESSANI, 2012).

Tendo isto em vista, a BFT se mostra uma opção interessante para a aplicação em sistemas distribuídos, promovendo segurança quanto a falhas e quanto a consenso. Sendo assim, a BFT se mostrou atrativa para plataformas *blockchain*.

Como exemplo da aplicação de BFT em *blockchain*, há o Hyperledger Fabric, que executa o protocolo BFT em seus peers de validação, através de uma SMR que aceita três tipos de operações como transações(CACHIN et al., 2016).

Outro exemplo desta aplicação ocorre na plataforma de *smart contracts* Chainspace(AL-BASSAM et al., 2017).

2.4 Camadas OM-AM

OM-AM é um *framework* para segurança, constituído pelas camadas Objetivos, Modelos, Arquitetura e Mecanismos. As camadas de Objetivos e Modelos indicam quais (*what*) são os objetivos e decisões de segurança, enquanto as camadas de Arquitetura e Mecanismo indicam como (*how*) atender estes requisitos de segurança. A Figura 2.3 mostra a estrutura do OM-AM (SANDHU, 2000).

OM-AM é melhor compreendido quando se vê os modelos de controle de acesso com grande escopo de objetivos de atuação, e também suporta diversas arquiteturas ou mecanismos para sua implementação. Um exemplo disso é o RBAC. Na Figura 2.4 pode-se observar um exemplo da representação do RBAC com o OM-AM.

(OUADDAH; ELKALAM; OUAHMAN, 2016)apresentam um estudo sobre os requisitos de autorização em ambientes IoT e utilizam o *framework* OM-AM para estruturá-los.

Diversos foram os Objetivos encontrados para o ambiente IoT, podendo variar de acordo com sua aplicação e sua proximidade aos seres humanos (pessoal, social, público, entre outros).

A partir dos Objetivos encontrados, Ouaddah, Elkalam e Ouahman (2016) passam a discutir sobre técnicas e tecnologias que podem ser utilizadas nos processos de autorização desses

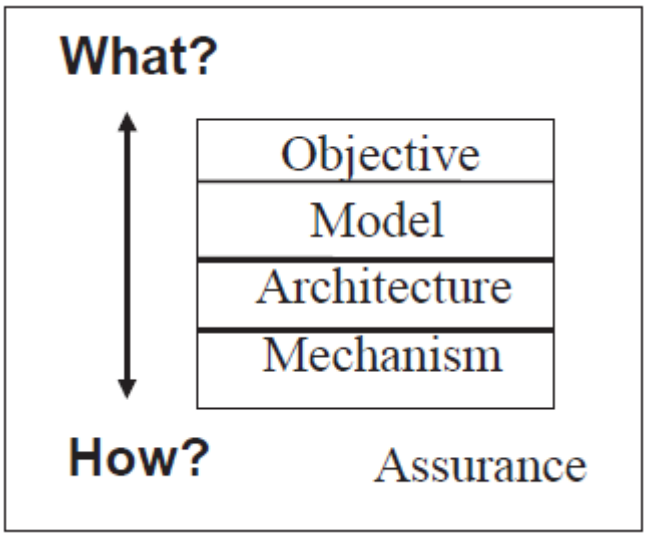

Figura 2.3: *Framework* OM-AM, conforme Sandhu (2000).

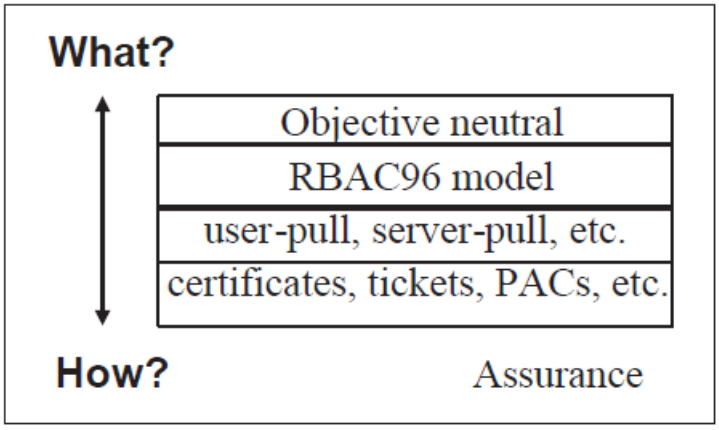

Figura 2.4: *Framework* OM-AM para sistemas RBAC, segundo Sandhu (2000).

ambientes, desenvolvendo abordagens centralizadas e descentralizadas.

Pode-se destacar o uso de endereços Bitcoin para identificação das entidades, além do uso de *smart contracts* através de linguagem de *script* para políticas de acesso com granularidade fina.

O *framework* apresenta também a distribuição de *tokens* de autorização através do *blockchain*, que é cifrado usando o sistema nativo de chaves públicas/privadas das criptomoedas. O uso inapropriado do *token* de autorização pode ser identificado pelo mesmo mecanismo utilizado para impedir o *double spending* das criptomoedas.

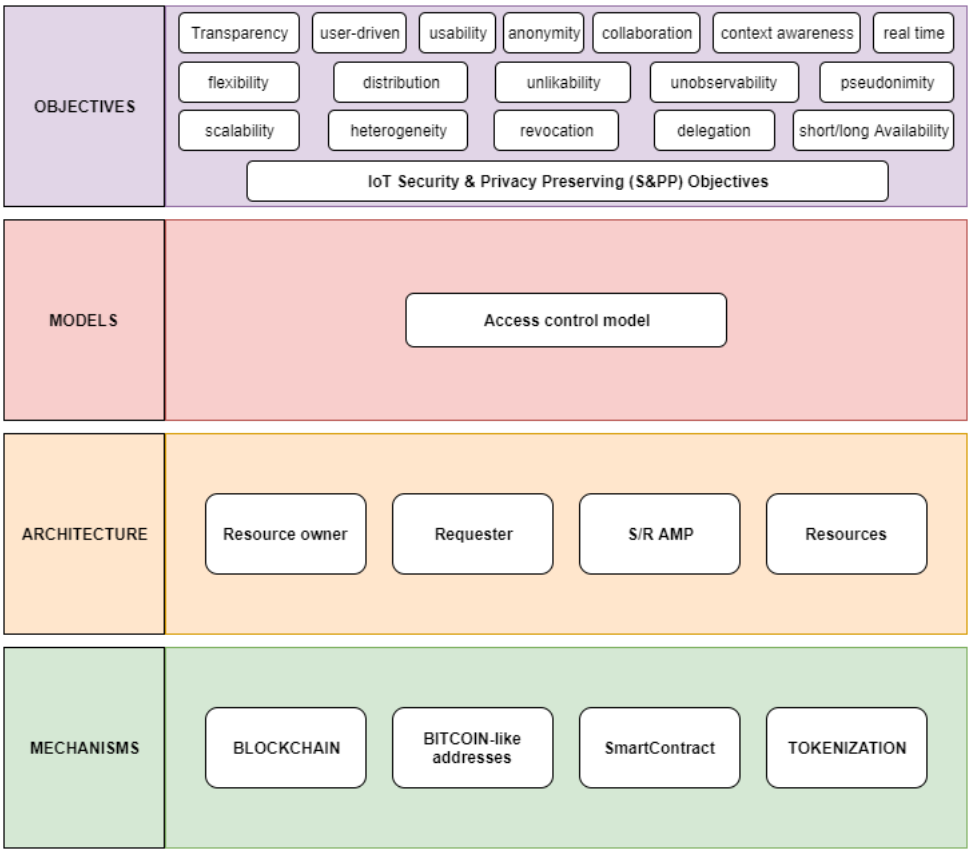

Figura 2.5: Modelo referência de OM-AM conforme Ouaddah, Elkalam e Ouahman (2016).

A Figura 2.5 mostra o modelo de referência do OM-AM para IoT, apresentado por Ouaddah, Elkalam e Ouahman (2016). As partes mencionadas em Objetivos e Modelos apresentam as necessidades dos ambientes IoT, enquanto Arquitetura e Mecanismos apresentam as proposta abordada no artigo.

2.5 Trabalhos Relacionados

Diversas são as abordagens propostas envolvendo o controle de acesso, desde novos modelos até extensões, melhorias ou novas representações dos modelos já existentes. Nesse sentido, é comum o uso de ontologias para a representação dos modelos ou suas políticas de acesso.

Finin et al. (2008) estuda o relacionamento entre o RBAC e OWL, definindo então, ontologias OWL que possam representar o modelo RBAC. Descreve duas abordagens para representação do RBAC em OWL. Na primeira abordagem, os papéis são representados como classes e subclasses. Na segunda abordagem, os papéis são representados como valores. Na abordagem de papéis como classes, a especificação é mais complexa, mas permite que consultas sejam respondidas de maneira eficiente através de Lógica Descritiva (LD). Na de papéis como valores,

a especificação é mais simples, mas não é possível explorar a LD para consultas. Ambas as abordagens possuem problemas no gerenciamento de mudanças de estado, devido à natureza monotônica do RDF (*Resource Description Framework*)/OWL.

Peleg et al. (2008) aborda uma questão frequente nos modelos de controle de acesso, que é a necessidade de identificação da situação na qual o acesso é solicitado, influenciando na decisão de permissão de acesso. A partir do RBAC, aplica OPM (*Object-Process Methodology*) para estruturar cenários e conceber um modelo chamado SitBAC (*Situation-Based Access Control*).

Em 2010, Cleeff, Pieters e Wieringa apresentam um estudo sobre os LBAC (*Location-Based Access Control*). Esses modelos de controle de acesso baseado em localização, têm sido apresentados como uma forma de melhorar a segurança em TI. Ao estabelecer localizações específicas para atuação de sistemas e usuários, atacantes terão mais dificuldade em comprometer um sistema. Fazem uma análise da motivação por trás do LBAC e seus potenciais benefícios, e examinam as metas que o LBAC tem potencial de cumprir, os sistemas específicos de LBAC que realizam essas metas e os contextos dos quais o LBAC é dependente. Entre as contribuições do trabalho, pode-se citar, em primeiro lugar, a proposta de um *framework* teórico para avaliação do LBAC, baseado em metas, sistemas e contexto. Em segundo lugar, formula e aplica critérios para avaliação da utilidade de um sistema LBAC. Terceiro, identifica quatro cenários de uso para LBAC: áreas e sistemas abertos, hospitais, empresas, e por fim *data centers* e instalações militares. Quarto, propõe direções para pesquisas futuras, como avaliar as vantagens e desvantagens entre os controles de acesso baseado em localização, físico e lógico; melhorar a transparência no processo de decisão do LBAC; formular critérios de design para instalações e ambientes de trabalho para otimizar o uso do LBAC.

Cleeff, Pieters e Wieringa (2010), ainda apresentam cinco metas principais para os LBAC, que são o princípio da separação de deveres (*principle of separation of duty*), princípio do privilégio mínimo (*principle of least privilege*), capacidade de responsabilização (*accountability*), capacidade de manutenção (*maintainability*) e usabilidade (*usability*).

O princípio da separação de deveres refere-se ao compartilhamento da responsabilidade sobre uma ação, por dois ou mais usuários, com objetivo de aumentar a confiabilidade por verificação. Quanto ao controle de acesso, as permissões de acesso dependem da ausência ou presença de outras pessoas(CLEEFF; PIETERS; WIERINGA, 2010).

O princípio do privilégio mínimo retrata que o acesso deve ser concedido apenas quando for necessário e possuir objetivos legítimos. Deste modo, evita-se problemas com erros de acessos indevidos; ações abusivas de agentes maliciosos; e diminui o prejuízo causado em possíveis invasões, como por roubo de senha (CLEEFF; PIETERS; WIERINGA, 2010).

Capacidade de responsabilização é a necessidade de manter os usuários responsáveis por suas ações, monitorando e registrando (*logging*) as operações. Deste modo, inibe-se ações indevidas e, em alguns casos, permite a recuperação após ações ilegais (CLEEFF; PIETERS; WIERINGA, 2010).

Capacidade de manutenção refere-se à necessidade de manter o sistema com funcionamento adequado. Deve haver sincronismo entre a descrição dos trabalhos, cargos, ou papéis, dos usuários e as autorizações que eles recebem no sistema. Se houver falha no sincronismo, o princípio do privilégio mínimo pode ser rompido(CLEEFF; PIETERS; WIERINGA, 2010).

Usabilidade é a necessidade de ter fácil utilização, não sobrecarregando os usuários pelas medidas de segurança do sistema. Um sistema não será bem recebido e poderá ser evitado pelos usuários se suas exigências para utilização não forem coerentes(CLEEFF; PIETERS; WIERINGA, 2010). Portanto, o sistema deve ter usabilidade próxima ao intuitivo para ser bem recebido pelos usuários e atingir seu objetivo.

Tais características, chamadas de metas, por Cleeff, Pieters e Wieringa (2010), também podem ser levadas em consideração em outros modelos de controle de acesso, não somente nos LBAC.

Boonyarattaphan et al. (2010) também utilizam informações de localização em sua abordagem de controle de acesso, mas com adição de informações temporais. Apresentam um modelo chamado de STAC (*Spatial-Temporal Access Control*), com objetivo de evitar problemas de segurança em aplicações online. No artigo, os autores utilizam como cenário uma aplicação voltada para serviços de saúde, com controle de acesso às informações de pacientes.

Boonyarattaphan et al. (2010) monitora o momento de login no sistema, obtendo informações como endereço IP e MAC, para avaliar se o usuário está em uma rede segura, além de utilizar o horário como fator adicional. Propõe também que o usuário possa definir as condições de acesso às suas informações.

Bertino e Kirkpatrick (2011) também abordam sistemas de controle de acesso baseados em localização, utilizando o termo GEO-RBAC (*Geo-spatial* RBAC). Discorrem sobre padronizações, requisitos, tecnologias para obtenção de localização e áreas de aplicações, além de formas de utilização do posicionamento físico e lógico no controle de acesso. Apresentam também a possibilidade de utilização de ativação de papéis (*roles*) de acordo com localização. Em outro trabalho (KIRKPATRICK; DAMIANI; BERTINO, 2011), apresentam também a linguagem Prox-RBAC para expressar as políticas de acesso utilizadas nestes modelos.

Gupta, Kirkpatrick e Bertino (2012) ressaltam que, em muitos casos, a localização relativa

de um usuário pode ser mais importante do que sua localização absoluta. Alegam que muitos modelos de controle de acesso, ou suas extensões, mesmo tomando diversas informações de contexto incorporadas ao processo de decisão, ainda falham por não considerar tal localização relativa. Isso ocorre pelo fato de a posição do usuário em relação à outras pessoas ser uma ameaça a segurança das informações. Levando em consideração o princípio da separação de deveres, por exemplo, é mais importante saber se duas pessoas estão na mesma sala, do que saber se estão em uma localização pré-definida. A partir disso, apresentam uma definição rigorosa de proximidade, entre usuários, baseada em relações topológicas formais. Mostram alguns domínios nos quais esta definição pode ser aplicada, como em redes sociais, canais de comunicação, e modelos baseados em atributos e tempo. Apresentam também resultados teóricos para estes sistemas, como análise de complexidade, modelos para protocolo de criptografia, e provas de características de segurança.

Khan e Sakamura (2012) tratam a necessidade da sensibilidade de contexto dentro da computação ubíqua, seu uso para facilitar aplicações melhoradas e a capacidade de servir como parâmetro de segurança significante. Declaram que seu uso como parâmetros de segurança pode assegurar que serviços não sejam providos a qualquer um, em qualquer momento e local, mas apenas à pessoa certa no momento e local certos. Sendo assim, propõem um modelo de controle de acesso sensível ao contexto aplicado a serviços ubíquos de cuidados de saúde. O artigo apresenta ainda uma implementação de sensibilidade ao contexto adicionada ao RBAC, utilizando cartões eTRON para identificação única de cada usuário através de um ID de 128 *bits*. O domínio de cuidados de saúde foi escolhido pela quantidade de requisitos não triviais de controle de acesso sensíveis ao contexto. Requisitos estes que podem ser utilizados, direta ou indiretamente, em outras aplicações de computação ubíqua.

Em 2013, Choi et al. propõem um modelo de controle de acesso baseado em proximidade, com uso de serviços baseados em localização, através de dispositivos *Bluetooth*. O domínio do estudo são os registros médicos portáteis e dispositivos médicos implantados ou vestíveis. Na abordagem apresentada, são utilizadas chaves e assinaturas parciais. O leitor precisa de todas chaves parciais e da composição das assinaturas para obter acesso. Para isto, ele deve estar na área determinada como confiável. O modelo também previne riscos de ataques como *replay attacks*, *collusion attacks* e *distance spoofing attacks*.

Khan e McKillop (2013) propõem um modelo de controle de acesso centrado em privacidade, considerando o consentimento dos pacientes. Este modelo utiliza a linguagem OWL para representação de conhecimento, de forma que as preferências de consentimento dos pacientes sejam armazenadas em forma de políticas, que serão utilizadas nas decisões de controle

de acesso. O modelo também prevê que as políticas de preferências possam ser compartilhadas para diferentes sistemas, sem gerar ambiguidade na interpretação. No artigo, os autores apresentam também um protocolo (*handshake protocol*) para troca segura de informações entre diferentes setores ou instituições.

Yarmand, Sartipi e Down (2013) apresentam uma abordagem de controle de acesso customizável que captura o comportamento dinâmico do usuário e o utiliza para determinar os direitos de acesso. O modelo é flexível e genérico, recebendo parâmetros de segurança dinamicamente a partir do usuário. Para tanto, utiliza formatos de dados padronizados e ontologias para prover compatibilidade e interoperabilidade ao modelo.

Li, Chu e Yao (2014) utilizam ontologias e tecnologias de *web* semântica para propor um modelo de autorização semântico flexível, capaz de atuar com fontes heterogêneas de dados, com contextos dinâmicos. Apresentam juntamente ao modelo, estratégias de resolução de conflitos.

Kayes, Han e Colman (2014) propõem um *framework* de controle de acesso sensível a situações e orientado a propósitos. O *framework* utiliza um modelo de políticas considerando os tipos de situações possíveis, levando em consideração o estado das entidades, o relacionamento entre elas e a intenção do usuário. Os autores apresentam ainda uma plataforma baseada em ontologias, para identificar e modelar as situações, e implementar as políticas de acesso. Para isso, utilizam a linguagem de ontologias OWL, estendida com SWRL. Implementam e avaliam o *framework* através de um ambiente simulado de cuidados de saúde.

Em 2015, Khan e Sakamura apresentam um modelo de controle de acesso com granularidade fina e sensível ao contexto, projetado sobre o RBAC. Utilizam a arquitetura eTRON para obter autenticação mútua e comunicação encriptada.

Amato et al. (2015) propõem um *framework* híbrido fornecendo, em primeiro lugar, um método baseado em semântica para dar forma semiestruturada a textos narrativos inseridos em um EHR; segundo, uma extensão do RBAC para autorização, controlando diferentes seções de um EHR, através de *web* semântica com a ontologia OWL, sobre as informações semiestruturadas; e terceiro, uma linguagem procedural de políticas, para codificar o controle de acesso no formato de regras *"if-then"*.

Zyskind, Nathan e Pentland (2015) afirmam que o recente aumento nos incidentes reportados de vigilância e brechas de segurança, comprometendo a privacidade de usuários, colocou em questionamento o modelo corrente, onde terceiros coletam e controlam grandes quantidade de dados pessoais. Por outro lado, Bitcoin demonstrou no campo financeiro que computação

confiável e auditável é possível, usando uma rede descentralizada de *peers* acompanhada de um registro público. O artigo aproveita essa ideia e descreve um sistema de gerenciamento de dados pessoais descentralizado que certifica que o usuário tenha posse e controle sobre seus dados. Nesse sentido, implementam um protocolo que transforma um *blockchain* em um gerenciador de controle de acesso automatizado que não requer confiança em terceiros. Diferente do Bitcoin, as transações no sistema não são financeiras, elas são usadas para carregar instruções, como armazenar, consultar e compartilhar dados. Os autores discutem ainda possíveis futuras extensões ao *blockchain* que podem aproveitá-lo como solução para problemas de confiança em computação na sociedade.

Para Hashemi et al. (2016), na implantação da IoT, sensores e atuadores são possuídos, acessados e ativados por uma infinidade de indivíduos e organizações. Acessar os dados produzidos pode ser tanto benéfico como desvantajoso para a sociedade. Estes dados podem representar as atividades de milhões de indivíduos e suas posses coletados por bilhões de "coisas". Agregações desses dados podem ser analisadas através da Internet e *Clouds*. Isso eleva os possíveis desafios de privacidade, de segurança, morais e éticos, dos quais as soluções irão requerer mecanismos de proteção flexíveis. O desafio é adquirir e distribuir dados na escala mundial da IoT, mantendo direitos dos indivíduos e organizações de proteger, usar e compartilhar seus dados. Claramente, um mecanismo e controle bem definidos precisam regular o acesso aos dados e suas agregações. O artigo descreve um mecanismo de granularidade múltipla, multinível e centrado ao usuário para compartilhar dados de dispositivos para pessoas e organizações. A solução usa listas de acesso e direitos de acesso seguindo noções formais de raciocínio sobre acesso. Descreve-se um mecanismo robusto auditável, transparente, distribuído, descentralizado e baseado em publicação-subscrição, e a automação dessas ideias no domínio da IoT pareado à atual geração de *clouds*. A estratégia adotada pelos autores é baseada em princípios e práticas de *blockchains* de transações já testados e usados nas criptomoedas e coloca o usuário no controle sobre o acesso a sua coleção de dados de sensores. O artigo descreve ainda a implantação dessas ideias para cuidados de saúde, *smart cities*, e carros autônomos.

Azaria et al. (2016) citam anos de regulamentação pesada e ineficiência burocrática como responsável pela lenta inovação em registros médicos eletrônicos. Apontam uma necessidade crítica por inovação, como personalização e ciência de dados para levar os pacientes a se envolver nos detalhes do seu cuidado de saúde e restaurar agência sobre seus dados médicos. Neste artigo é proposto o MedRec, um sistema de gerenciamento de registros descentralizado para manusear RMEs (Registros Médicos Eletrônicos), usando tecnologia *blockchain*. O sistema dá aos pacientes um registro compreensível e imutável e fácil acesso às suas informações médicas através de provedores e locais de tratamento. A estratégia aproveita propriedades úni-

cas de blockchain, fazendo o MedRec gerenciar autenticação, confidencialidade, capacidade de responsabilização e compartilhamento de dados.

MedRec ainda possui um design modular que o integra com provedores existentes e soluções de armazenamento local de dados, facilitando interoperabilidade e tornando o sistema adaptável e conveniente.

Azaria et al. (2016) incentivam as partes médicas interessadas a participar da rede como "mineradores" de *blockchain*. Isso lhes fornece acesso aos dados agregados e anônimos como recompensa pela mineração, por sustentar e assegurar a rede por *Proof of Work*. Dessa forma, os dados fornecidos como recompensa da mineração seriam semelhantes a um *big data*, fortalecendo os pesquisadores, enquanto mantém pacientes e provedores engajados na escolha de liberar os metadados.

English, Auer e Domingue (2016) afirmam que conceito de aplicações *peer-to-peer* não é novo, o conceito de tabelas *hash* distribuídas também não. Sendo assim, o que emergiu em 2008 com a publicação do Bitcoin foi uma estrutura de incentivo que uniu os dois paradigmas de software com um conjunto de estímulos econômicos, para motivar a criação de uma rede computacional dedicada mais poderosa que os supercomputadores mais rápidos do mundo. De acordo com os autores, além de possibilitar as moedas digitais, a tecnologia *blockchain* é um novo paradigma computacional com grandes implicações para o desenvolvimento futuro da *Web* e, consequentemente, o crescimento da *Semantic Web* e do *Linked Data*. Os autores demonstram no trabalho como tecnologias *blockchain* podem contribuir na realização de uma *Semantic Web* mais robusta, e apresentam um *framework* em que a *Semantic Web* é utilizada para melhorar a própria tecnologia *blockchain*.

Segundo Ouaddah, Elkalam e Ouahman (2016), segurança e privacidade são grandes problemas em ambientes IoT, mas a harmonização dos padrões e protocolos relacionados a IoT demoram e dificilmente são difundidas de forma generalizada. Propõem, então, um novo *framework* de controle de acesso em IoT baseado em tecnologia *blockchain*. A primeira contribuição consiste em fornecer um modelo de referência para o *framework* proposto dentro da especificação IoT de Objetivos, Modelos, Arquitetura e Mecanismo. Assim, apresentam o FairAccess como um framework de gerenciamento de autorizações completamente descentralizado e com pseudônimos e preservação da privacidade, que permite aos usuários possuírem e controlarem seus dados. Então, adaptam o *blockchain* como um gerenciador de controle de acesso descentralizado. Diferente das transações financeiras do bitcoin, o FairAccess apresenta novos tipos de transações que são usadas para conceder, receber, delegar e revogar acesso. Por fim, estabelecem uma implementação inicial, como prova de conceito, com Raspberry Pi e *blockchain*

local, discutem as limitações e propõem oportunidades futuras.

A Tabela 2.1 mostra uma comparação entre alguns dos trabalhos relacionados, quanto ao tipo de modelo, tipo de política de acesso, variáveis de contexto e uso de *blockchain*.

	Modelo de controle de acesso	**Tipo de política de acesso**	**Variáveis de contexto**	**Blockchain**
Corradi, Montanari e Tibaldi (2004)	Centrado em contexto	Políticas de contexto	Conceitual	-
Finin et al. (2008)	RBAC	Ontologia	-	-
Peleg et al. (2008)	Baseado em Situações	Situation schema	Localização e tempo	-
Cleeff, Pieters e Wieringa (2010)	Baseado em Localização	Políticas de localização	Localização	-
Boonyarattaphan et al. (2010)	Autenticação multifator + RBAC	Regras de autenticação	Espaço, tempo e informações de rede	-
Bertino e Kirkpatrick (2011)	Baseado em Localização	Políticas de localização	Localização	-
Kirkpatrick, Damiani e Bertino (2011)	RBAC + proximidade	Políticas de proximidade	Proximidade	-
Gupta, Kirkpatrick e Bertino (2012)	RBAC + proximidade	Políticas de contexto	Proximidade	-
Khan e Sakamura (2012)	RBAC + políticas de contexto	Ontologia + tokens	Localização e tempo	-
Choi et al. (2013)	Chaves e assinaturas	Dispositivos bluetooth e assinaturas	Proximidade	-
Khan e McKillop (2013)	Centrado em Privacidade	Ontologia	Regras de consentimento	-
Yarmand, Sartipi e Down (2013)	Baseado em Comportamento	Políticas de contexto	Localização, tempo e comportamento	-
Li, Chu e Yao (2014)	Baseado em regras	Ontologia e Web Semântica	Conceitual	-
Kayes, Han e Colman (2014)	PO-SAAC (baseado em RBAC)	Ontologia	Situação e intenção	-
Khan e Sakamura (2015)	DAC + RBAC	Ontologia + tokens	Localização e tempo	-
Amato et al. (2015)	RBAC customizado	"if-then" baseado em ontologia	Conteúdo	-
Zyskind, Nathan e Pentland (2015)	Blockchain como gerenciador de acesso	Políticas armazenadas	Conceitual	Bitcoin
Hashemi et al. (2016)	Protocolo de gerenciamento de dados	Troca de mensagens	-	Conceitual
Azaria et al. (2016)	Blockchain como gerenciador de acesso	Permissões de busca	-	Ethereum
English, Auer e Domingue (2016)	Discussão: atribuição semântica a blockchains	Web Semântica	Conceitual	Ethereum
Ouaddah, Elkalam e Ouahman (2016)	Blockchain como gerenciador de acesso	Políticas armazenadas + tokens	Conceitual	Bitcoin

Tabela 2.1: Comparativo de trabalhos relacionados.

A partir dos trabalhos mostrados nesta seção, é possível observar a evolução dos modelos propostos pela comunidade científica, assim como o crescimento da demanda por segurança com o surgimento de novas tecnologias e novas áreas para aplicação do controle de acesso. Nota-se também a constante busca por novas tecnologias para apoiar e avançar este campo. E é neste ambiente que a proposta deste trabalho será desenvolvida.

2.6 Discussão sobre controle de acesso sensível ao contexto

Os modelos de controle de acesso sensíveis ao contexto (CAAC - *Context Aware Access Control*), usualmente são extensões do tradicional modelo RBAC (*Role Based Access Control*). Tais extensões visam a adição de informações de contexto do usuário na tomada de decisão sobre o acesso a recursos.

O núcleo básico de um RBAC se dá por dois tipos de atribuições que se relacionam com os papéis (*roles*): as atribuições de usuários (UA - *User Assignment*) e as atribuições de permissões (PA - *Permission Assignment*).

As UAs se referem ao conjunto de papéis para os quais um usuário pode ser mapeado, de acordo com suas obrigações dentro de um sistema ou organização. Em outras palavras, é o relacionamento usuário-papel (*user-role*) sendo do tipo muitos-para-muitos (*many-to-many*).

Em uma universidade, por exemplo, o conjunto de papéis na UA poderia ser *{Professor, Orientador, Técnico Administrativo, Secretário, Chefe de Departamento, Coordenador de Graduação, ...}*. Sendo que o mesmo usuário pode receber mais de um papel, como *{Professor, Orientador, Coordenador de Graduação}* e estes papéis podem ser atribuídos a mais de um usuário.

Já as PAs se referem ao conjunto de ações permitidas para determinado papel, sendo que o relacionamento entre papéis e permissões de acesso, também é do tipo muitos-para-muitos.

Usando novamente o exemplo da universidade, ao papel de *Professor* poderia ser atribuído um conjunto de permissões como *{Acesso à salas de aula, acesso a notas de alunos, acesso a frequência de alunos}*. E as mesmas permissões também podem estar disponíveis para outros papéis.

Há também o conceito de sessão que contém o conjunto de papéis ativos do usuário em determinado período. Por tanto, as PAs dos papéis ativos do usuário compõem o conjunto dos direitos de acesso durante a sessão. Um usuário pode ter várias sessões (relacionamento *one-to-many*).

Dessa forma, os relacionamentos no RBAC são como na Figura 2.6.

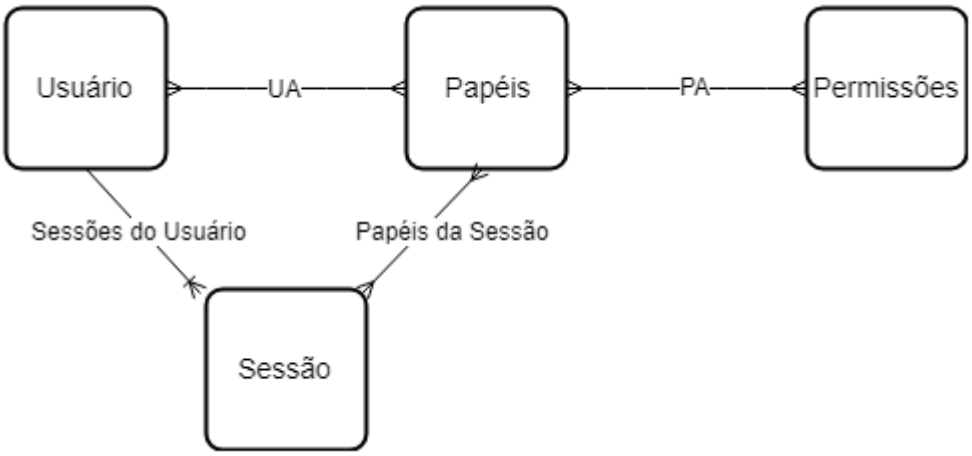

Figura 2.6: Relacionamentos no RBAC.

Outros modelos que usam o RBAC como base, alteram ou adicionam novas propriedades a este relacionamento, de acordo com o objetivo a ser alcançado, ou as limitações a serem eliminadas.

Para obter um controle mais fino sobre os direitos de acesso, surgiu o conceito *Separation of Duty* (SoD), para atender o princípio do privilégio mínimo. O SoD busca impor restrições às permissões de acesso que um usuário pode receber, de forma a evitar problemas como acessos indevidos.

Um exemplo de SoD é impor uma restrição na qual um usuário com um papel ativo não possa ativar um segundo papel cujo conjunto de permissões seja mutuamente disjunto (*mutually disjoint*) ao conjunto de permissões do primeiro papel. Ou seja, o segundo papel não poderá ser ativo, se o seu conjunto de permissões não possuir algum elemento em comum com o conjunto de permissões do primeiro papel. Isto evita que o mesmo usuário esteja com papéis ativos que o permitam realizar tarefas de diferentes interesses e permissões ao mesmo tempo.

As restrições de SoD podem atuar sobre os relacionamentos no modelo do RBAC, como nas atribuições do usuário, nas atribuições das permissões e nas sessões de usuários, como mostrado na Figura 2.7.

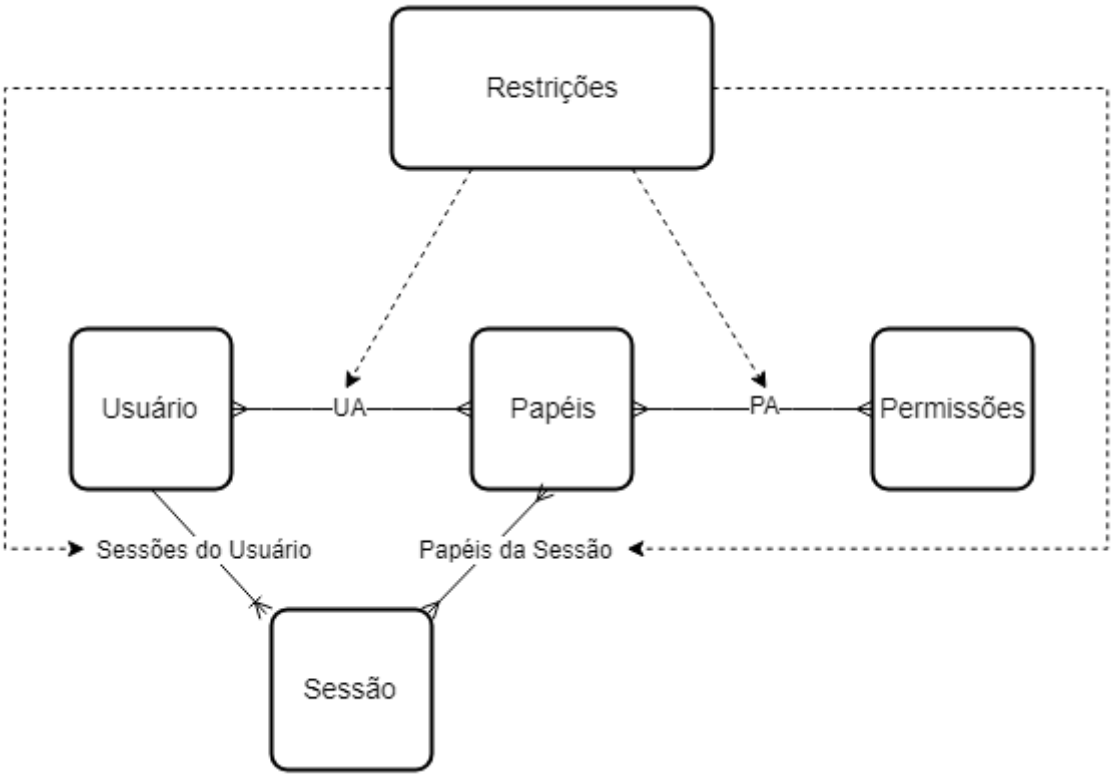

Figura 2.7: RBAC com restrições.

Outra das limitações presentes no RBAC tradicional envolvem, por exemplo, atividades que requeiram mais de um usuário, para que possam ser realizadas.

Além disso, outra limitação do RBAC ocorre em ambientes de computação ubíqua, onde as decisões podem ocorrer dinamicamente e as permissões de acesso não dependem apenas do papel do usuário, mas também do contexto em que a requisição de acesso é feita.

Assim, o interesse pela adaptação dos controles de acesso em ambientes dinâmicos, como de computação ubíqua, trouxe o desenvolvimento dos modelos de controle de acesso sensíveis ao contexto, CAAC.

Dessa forma, o diagrama de relacionamentos nos modelos de CAAC, costumam ser semelhantes aos do RBAC, com algumas alterações dependendo da proposta, mas podendo ser generalizados como representado na Figura 2.8.

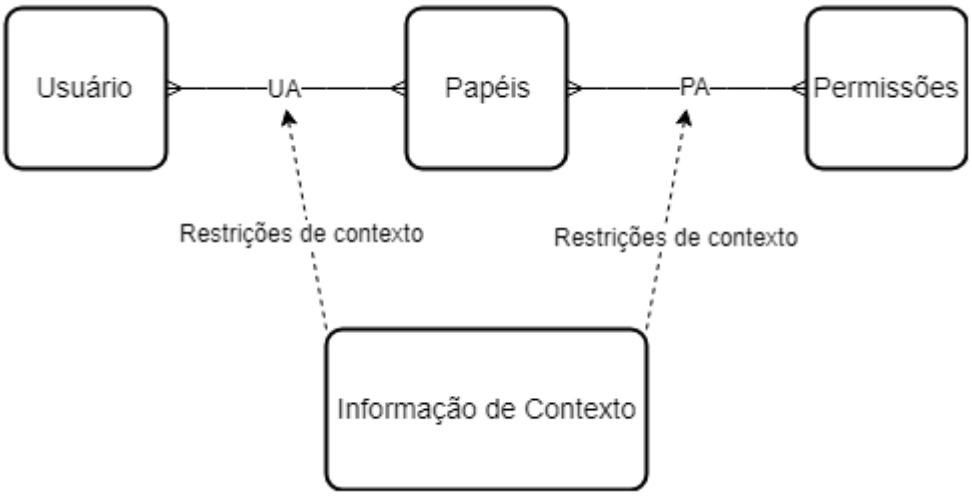

Figura 2.8: Relacionamentos nos CAAC.

As abordagens utilizadas nos modelos CAAC tendem a utilizar informações de contexto para alterar as permissões de acesso de um determinado papel. O contexto utilizado pode envolver variáveis como o estado do usuário, o estado do objeto a ser acessado, ou variáveis do ambiente.

Os CAAC não são exatamente um modelo único, mas sim uma série de propostas diferentes para abordar variáveis de contexto no controle de acesso. As propostas mais apresentadas, são as conhecidas como modelos espaço-temporais.

Modelos espaço-temporais abordam principalmente variáveis de localização, ou tempo, ou ambos. Estas variáveis são inseridas na especificação das políticas de acesso como condições adicionais para obter as permissões.

Uma alteração possível neste sentido é realizar a restrição do papel que um usuário pode assumir, de acordo com o tempo. Ou ainda, restringir as permissões atribuídas a um papel de acordo com o tempo.

Por exemplo, pode-se impedir que um usuário assuma o papel *Professor* enquanto estiver fora do horário de expediente na universidade. Ou ainda, restringir o conjunto de permissões de todos com papel *Professor*, enquanto estiver fora do horário de expediente na universidade.

De forma semelhante, é possível restringir o acesso de acordo com a localização, sendo que um usuário só teria permissão de acesso, caso esteja dentro de uma zona espacial específica.

Como exemplo, um usuário que só possa assumir seu papel de *Professor*, ou suas permissões, enquanto estiver dentro das dependências da instituição de ensino em que trabalha.

Outro tipo de alteração pode envolver a restrição sobre um objeto dependendo de seu estado. Por exemplo, um arquivo que possa ser classificado como confidencial perante algum evento no sistema.

Em uma possível generalização de um modelo CAAC, para um usuário *U* ter acesso ao objeto *O*, o modelo levaria em conta:

- Se *U* tem um papel *R*.
- Se *R* tem uma permissão *P*.
- Se *R* e *P* estão disponíveis no contexto do usuário.
- Se *R* e *P* estão disponíveis no contexto do objeto.

No entanto, é preciso a obtenção das informações de contexto e, por isso, existem propostas de infraestruturas para alimentar estas informações de contexto no sistema, auxiliando o processo de decisão.

Em Corradi, Montanari e Tibaldi (2004), por exemplo, os autores apresentam um *middleware* para auxiliar na obtenção do estado de contexto de um usuário móvel. O *middleware*, chamado UbiCOSM, ainda conta com arquitetura como a CARMEN (BELLAVISTA et al., 2003) para auxiliar no gerenciamento de contexto e identificação de identidades. O *middleware* UbiCOSM é representado como na Figura 2.9.

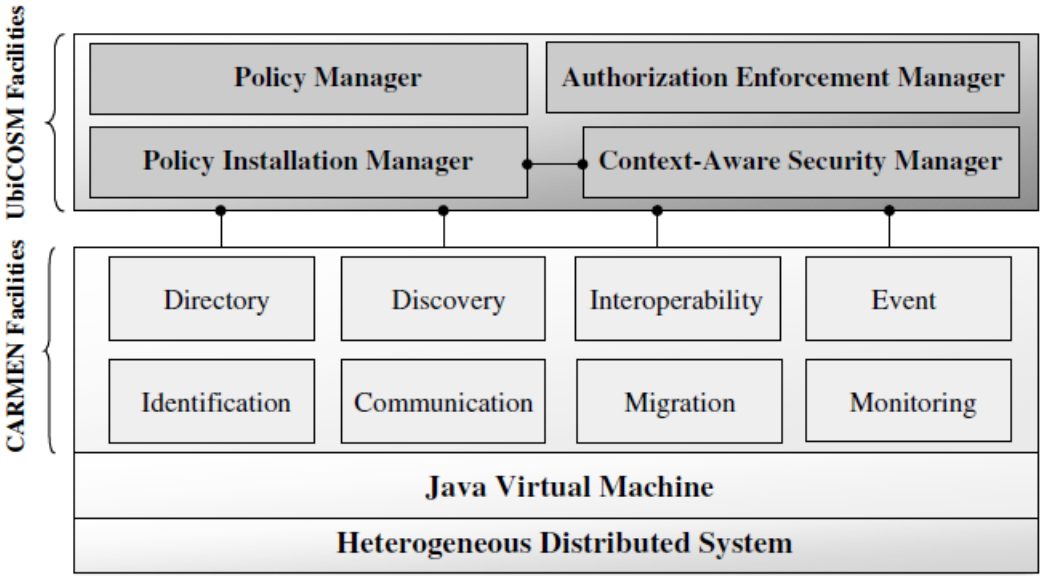

Figura 2.9: *Middleware* UbiCOSM.

O prox-RBAC (*proximity*-RBAC) proposto por Kirkpatrick, Damiani e Bertino (2011) trabalhou também com o conceito de proximidade entre usuários como variável de contexto adicional ao seu modelo de acesso.

Ainda envolvendo o prox-RBAC, Gupta, Kirkpatrick e Bertino (2012) apresentaram uma proposta de arquitetura com um componente opcional, chamado de *feature monitor* (FM), para auxiliar no mapeamento das informações do usuário no decorrer do tempo. O *feature monitor* ainda auxilia na emissão de um certificado digital, ou credencial, sobre suas informações. O prox-RBAC apresentado é representado como na Figura 2.10, sendo (a) opção sem FM, (b) com FM independente e (c) com FM em comunicação direta.

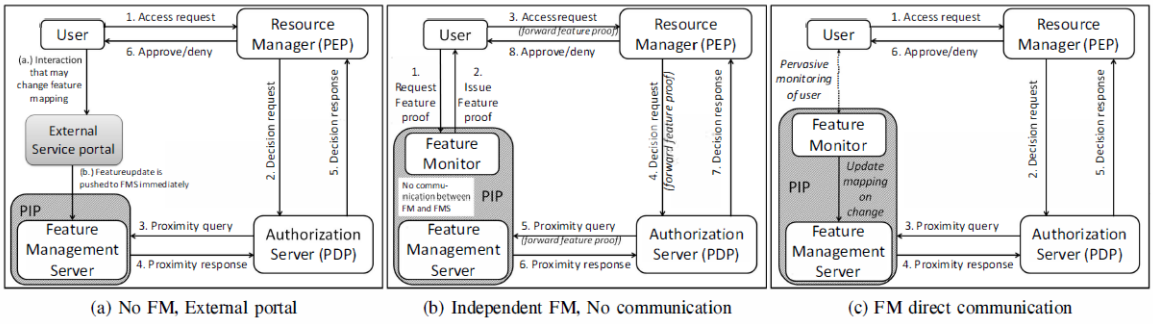

Figura 2.10: prox-RBAC conforme Gupta, Kirkpatrick e Bertino (2012).

O uso da proximidade como variável de contexto também é defendido no trabalho de Choi et al. (2013), que prega a utilização de dispositivos *Bluetooth* sobre uma área de interesse, cada um propagando parte de uma chave. O usuário, para provar sua proximidade da área, deve compor a chave a partir das mensagens propagadas com auxílio dos dispositivos *Bluetooth* e sua assinatura.

Khan e Sakamura (2012) já passa a propor o uso de ontologias OWL na estruturação do modelo e um repositório de políticas de contexto, alimentando um gerenciador de acesso, além de utilizar *tokens* na delegação de acesso.

Diversas outras propostas também defendem o uso de ontologias, como OWL, para a estruturação de informações de contexto, ou até a definição de políticas. Estas propostas sempre destacam uma melhor possibilidade de extrair semântica através desta estruturação.

Por volta de 2015, com a popularização das redes *blockchain*, novas propostas usando estas plataformas começaram a chamar atenção. Estas propostas apontam uma necessidade presente nos sistemas ubíquos, que é a descentralização dos sistemas de decisão.

Zyskind, Nathan e Pentland (2015) apresentam a utilização da rede *blockchain* Bitcoin para controle de acesso e armazenamento. Eles propõem a implementação de dois novos tipos de

transação: *Taccess* para o gerenciamento de controle de acesso e *Tdata* para o armazenamento e recuperação de dados. Ainda conta com um *hashtable* distribuída (DHT) mantida pelos nós responsáveis pelas transações de leitura e escrita. A proposta de Zyskind, Nathan e Pentland (2015) é representada na Figura 2.11.

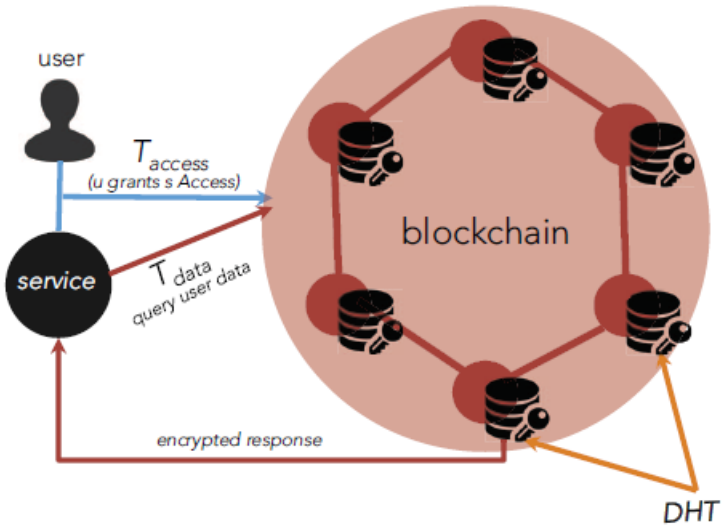

Figura 2.11: Controle de acesso com uso de *blockchain*, conforme Zyskind, Nathan e Pentland (2015).

Zyskind, Nathan e Pentland (2015) ainda ressaltam a importância de não necessitar confiar à terceiros a gestão de dados sensíveis do usuário.

No mesmo sentido, Azaria et al. (2016) também buscam a descentralização dos controle de acesso, mas apresenta uma implementação utilizando a plataforma *blockchain* Ethereum. No artigo, os autores também defendem que as organizações interessadas em prestar serviços que envolvam este controle de acesso, participem da rede blockchain como mineradores.

Ouaddah, Elkalam e Ouahman (2016) apresentaram um *framework* de controle de acesso baseado em blockchain chamado de FairAccess. Esse *framework* promove a adaptação das transações na plataforma Bitcoin para transferir *tokens* de acesso no lugar da criptomoeda bitcoin e é representado na Figura 2.12.

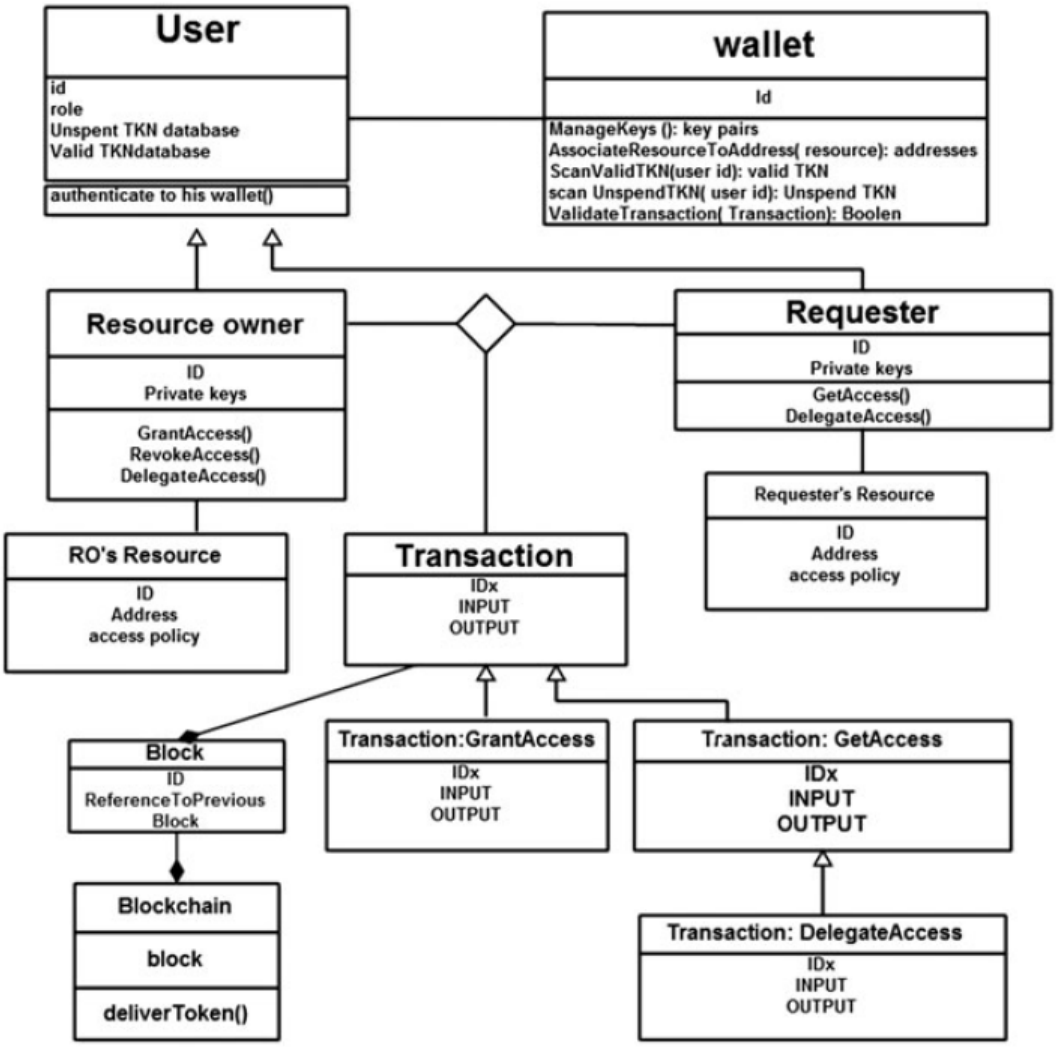

Figura 2.12: Diagrama de classes do FairAccess, conforme Ouaddah, Elkalam e Ouahman (2016).

Ouaddah, Elkalam e Ouahman (2016) propõem três tipos de transações base para serem implementadas no *blockchain*: A transação *GrantAccess*, em que um proprietário define uma política de acesso para seu recurso e cria um *token* de acesso; a transação *DelegateAccess*, na qual um usuário transfere um *token* de acesso; e a transação *GetAccess*, em que um requerente gasta seu *token* de acesso.

Trabalhos e sistemas existentes, como Ouaddah, Elkalam e Ouahman (2016), já exploram os mecanismos de autenticação e confiança distribuída providos pelos sistemas *blockchain*. Entretanto, o uso de *smart contracts* pode ser melhor explorado nos sistemas *blockchain*, atribuindo também maior flexibilidade aos modelos de controle de acesso.

Capítulo 3

MODELO PARA CONTROLE DE ACESSO USANDO HYPERLEDGER

O crescimento da utilização de ambientes com interação de dispositivos por invocação de serviços impulsionou a pesquisa sobre esse cenário e as suas necessidades.

No cenário adotado neste trabalho, considera-se a comunicação entre nós que queiram compartilhar recursos perante certas condições e outros nós que queiram ter acesso a estes recursos e que, para tal, precisam satisfazer as condições estabelecidas.

Tendo em vista as flexibilidades do modelo de controle de acesso baseado em papéis e usando dados de contexto em cenários de IoT, ou em qualquer situação de interação direta entre dispositivos, bem como o modelo de autenticação distribuída provido pelos sistemas de *blockchain*, esse trabalho propõe a criação de um sistema de autenticação que integre essas plataformas.

Neste cenário adotou-se o uso de plataforma *blockchain* pela sua capacidade de permitir estabelecer uma relação de confiança distribuída, entre nós que não possuem uma confiança pré-estabelecida entre si, nem com uma terceira parte centralizada. O *blockchain* também permite registrar serviços e operações que se deseja realizar de forma verificável. Estes serviços podem ser criados, armazenados e acessados de maneira distribuída, com o uso de uma plataforma *blockchain*.

Para tratar dados de contexto e a autenticação de forma distribuída, propõe-se ainda o uso de *smart contracts* para a especificação dos códigos dos serviços de forma confiável entre múltiplos pares, sem o uso de uma entidade central confiável. *Smart Contract* é uma estrutura presente em algumas plataformas *blockchain* que permite especificar código verificável em função de dados de entrada e do estado da rede *blockchain*.

As operações previamente indicadas nos *smart contracts* e solicitadas pelos participantes são armazenadas no *blockchain* e podem ser verificadas. As verificações sobre as operações para registro e alteração do estado atual da rede são feitas por meio de consenso entre os nós participantes.

Desta forma, este capítulo apresenta uma proposta de mecanismos sobre essa infraestrutura que permitam verificar variáveis de contexto dos nós em comunicação para satisfazer condições necessárias para as transações e acessos.

A viabilidade da proposta é analisada através de um modelo de sistema criado sobre o *Hyperledger Fabric*. Sobre esse sistema, propõe-se um conjunto de atores específicos, que exercem diferentes papéis num sistema distribuído / IoT / D2D.

A seção *Arquitetura de um mecanismo de controle de acesso usando Hyperledger* apresenta os componentes básicos de uma rede *blockchain* no *Hyperledger*. A seção *CAAC usando smart contracts* apresenta a primeira proposta, com classes, objetos e operações, desenvolvida sobre os recursos oferecidos pelo sistema *Hyperledger*. Na seção *CAAC com smart contracts baseado em transferência de recursos*, apresenta-se a segunda proposta que visa a transferência de *Assets* como fator para obtenção de acesso. Na seção *Estruturas do Sistema Hyperledger* mostra-se os recursos do *Hyperledger* que foram utilizados. A seção *Transações e Registros* mostra como as operações do sistema são tratadas pelo Hyperledger, como as transações são aplicadas e discute o que pode ser utilizado para melhorar o desempenho do modelo, além de discutir alguns aspectos de segurança e confiabilidade. A Análise da viabilidade do modelo é apresentada no *Capítulo 4 - Resultados e Discussões* através de uma implementação com um estudo de caso.

3.1 Arquitetura de um mecanismo de controle de acesso usando Hyperledger

Plataformas *blockchain* são redes P2P com um mecanismo de registro de dados, compartilhado entre nós participantes, que armazenam transações processadas em ordem cronológica inviolável. Esta rede é capaz de fornecer uma relação de confiança descentralizada entre os nós.

Para o desenvolvimento do modelo neste trabalho foi utilizada a plataforma *blockchain Hyperledger Fabric*, pela sua versatilidade quanto à construção da rede *blockchain* e transações, sua não dependência de criptomoedas e a disponibilidade de uso de *smart contracts*, chamados de *chaincodes*.

Hyperledger Fabric é um dos projetos *Hyperledger* com desenvolvimento hospedado pela

The Linux Foundation. Trata-se de uma implementação de estrutura *blockchain* com a finalidade de atuar como base para aplicações, com funcionalidades como consenso e serviços de associação entre membros, aos moldes de tecnologia *plug-and-play* (conectar e usar).

Blockchain é utilizado para registro de recursos, de políticas de acesso, de dados de contextos e transações, bem como para obtenção de acesso, atuando também como registro (*logging*) auditável das operações.

No sistema *Hyperledger Fabric* há também a utilização das tecnologias de contêineres, aplicadas principalmente para hospedar a execução de *smart contracts*. No *Hyperledger Fabric* os *smart contracts* são chamados de *chaincodes* e são utilizados para compor toda a lógica da aplicação do sistema e suas transações.

Plataformas *blockchain* são sistemas distribuídos formados por diversos nós que se comunicam. Na plataforma *Hyperledger*, esse sistema distribuído é responsável por executar programas contidos nos *chaincodes*, gravar o estado e os dados de registro, e executar as transações.

Os *chaincodes* são utilizados para determinar como as transações ocorrerão. Suas especificações podem ser implementadas como código executável com funções para manipular os dados de entrada, o estado e os registros na rede.

O elemento central do *Hyperledger* é o *chaincode*, pois todas as transações são operações invocadas a partir dele. Pode haver também um ou mais *chaincodes* responsáveis pelo gerenciamento das funções e parâmetros do sistema e, portanto, chamados de *system chaincodes*.

A rede é composta por nós lógicos nos papéis de **cliente**, **endorsing peer** e **ordering peer**. Nós atuando como clientes são responsáveis por criar propostas de transações, aguardar assinaturas de endosso e propagar ao *ordering service* as transações assinadas. *Endorsing peers* ficam em execução permanente, coletando propostas de transação e simulando a execução dessas propostas, assinando propostas endossadas, realizando *commits* e mantendo seus *PeerLedgers*. *Ordering peers* são responsáveis pelo *ordering service*; eles garantem a ordem cronológica das transações, coletam propostas assinadas, verificam o endosso e redistribuem as transações válidas ordenando *commits*.

O estado do sistema é composto por um armazenamento de versões de pares chave-valor (*key-value*), onde as chaves são nomes e os valores são *blobs* (*binary large objects*) referentes aos nomes.

Transações precisam ser endossadas pelos *peers* e aplicadas para terem efeito sobre o estado do sistema. Há dois tipos de transações, as de implantação (*deploy transactions*) e as de invocação (*invoke transactions*).

Uma transação de implantação é responsável por criar um novo *chaincode* e tem como parâmetro um programa, de forma que quando a execução desta transação é bem sucedida, significa que o *chaincode* foi instalado no sistema.

Uma transação de invocação se refere aos *chaincodes* já instalados e a alguma das funções contidas neles. Quando a transação é bem sucedida, a função referenciada do *chaincode* é executada, podendo gerar uma saída e modificar o estado correspondente.

3.2 CAAC usando smart contracts

O modelo proposto neste trabalho é descrito a partir do diagrama de classes UML, conforme ilustrado na Figura 3.1.

Semelhante ao modelo proposto em Ouaddah, Elkalam e Ouahman (2016), com o FairAccess, este trabalho também utiliza os termos **requerente** (*Requester*) e **proprietário de recurso** (*ResourceOwner*) para identificar os participantes do sistema de acesso. Estes participantes são modelados em classes próprias, extendidas da classe usuário (*User*), com diferentes tipos de atividades e informações.

No entanto, a abordagem utilizando *Hyperledger Fabric* com *smart contracts* (*chaincodes*) permite uma modelagem mais flexível do que a de Ouaddah, Elkalam e Ouahman (2016). Esta plataforma promove um gerenciamento de chaves e certificados menos trabalhoso do que a plataforma Bitcoin utilizada por Ouaddah, Elkalam e Ouahman (2016) e, por ser desvinculada de criptomoedas, não é necessária a adaptação de troca de *tokens* de acesso no lugar de bitcoins.

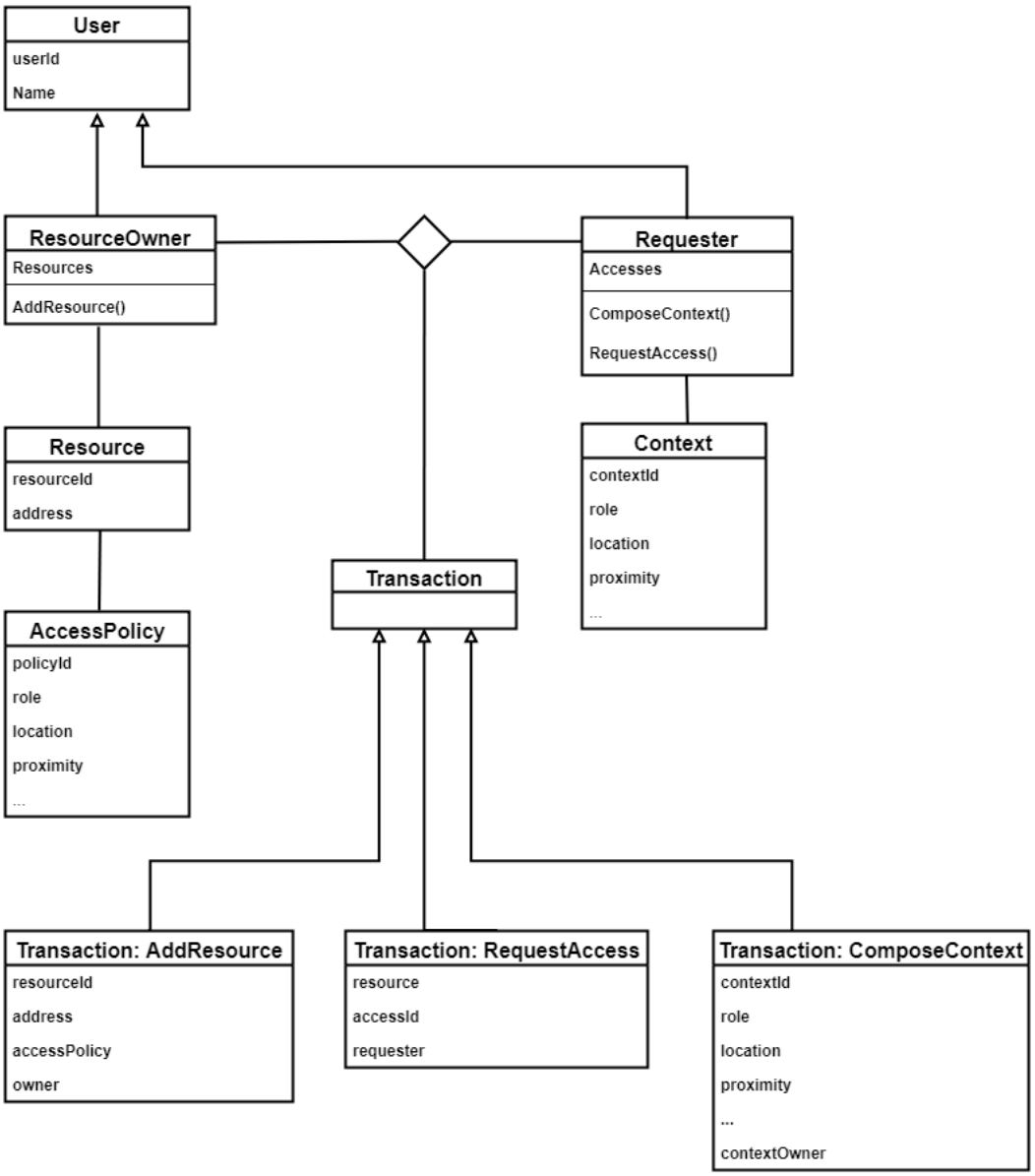

Figura 3.1: Diagrama UML para o CAAC usando smart contracts.

Com o sistema *Hyperledger*, trabalha-se com os conceitos de *Participant* e *Asset*. *Participants* são os atores que interagem com a rede, usados para definir as classes dos participantes na comunicação, no caso desta proposta, *ResourceOwner* e *Requester*. *Assets* são os objetos utilizados na interação dos *participants* com as transações, que podem ser criados, lidos, atualizados ou deletados. Os *assets* definidos no modelo aqui proposto são os recursos (*Resource*), políticas de acesso (*AccessPolicy*), contexto (*Context*) e acesso (*Access*).

A implementação do modelo aqui proposto visa os elementos de sistemas de interação baseados na interação via serviço. Os usuários de serviços são os usuários da rede *blockchain*, enquanto os prestadores de serviços são os *peers* que executam as operações (*chaincodes*). As

transações por sua vez definem os requisitos necessários e as regras de execução do serviço (também definidos nos *chaincodes*).

Considerando um caso comum de obtenção de acesso neste modelo, um usuário de serviço requerindo acesso deve prover seus parâmetros e informações (dados de contexto) ao fazer a solicitação do serviço. Caso os parâmetros do requerente atendam os requisitos definidos (políticas de acesso) o serviço é executado e retorna suas saídas.

3.2.1 User

User é um *Participant* usuário do sistema, é aquele que está conectado na rede com um nó cliente e é composto por identificador e nome. Usuários podem assumir as funções de proprietário de recurso (*ResourceOwner*), ou requerente de acesso (*Requester*).

Proprietários de recursos possuem um conjunto de recursos. Cada recurso é adicionado através de uma transação chamada *AddResource*, definida nessa proposta.

Requerentes possuem um *Context* que armazena as variáveis de contexto do requerente no momento, e um conjunto de acessos aos quais já têm permissão. Informações de contexto são definidas pela chamada *ComposeContext*.

3.2.2 Resource

Resource é modelado como um *Asset* representando um recurso que é registrado por seu proprietário, e que poderá ser acessado pelos requerentes que satisfizerem a política de acesso especificada. Todo recurso é formado por um identificador, um endereço e uma política de acesso que contém as condições de contexto que o requerente deverá possuir para obter acesso.

3.2.3 AccessPolicy

AccessPolicy é modelado como um asset contendo as políticas de acesso necessárias para acessar um *Resource*. As políticas podem conter informações como o papel (*role*) e as variáveis de contexto necessários para um *Requester* obter acesso.

É no *AccessPolicy* que são definidos os requisitos para acesso. É desse *Asset* que serão extraídos os parâmetros para aplicação da lógica de acesso durante a execução.

Por exemplo, caso um recurso cadastrado represente informações de um exame médico, uma política de acesso poderia exigir que o requerente de acesso possua o papel *Médico*, que

esteja em *Hospital* ou *Consultório*, e próximo a outro profissional de saúde, entre outras condições possíveis. Neste modelo proposto, estas condições são registradas na estrutura chamada *AccessPolicy*.

3.2.4 Context

Context é um *asset* que foi definido para representar o contexto atual de um *Requester*. No contexto devem estar presentes o papel ativo (*role*) do *Requester* e suas informações de contexto. As informações de contexto podem conter a localização (*location*) do *Requester*, a proximidade (*proximity*) em relação a outros usuários ou objetos, tempo, intenções de atividades, entre outras.

O contexto de um *Requester* é o parâmetro que será passado ao solicitar o serviço para obtenção de acesso, por isso faz-se necessário que essas informações não possam ser forjadas. Para tal, devem ser implementados mecanismos de verificação sobre a autenticidade dessas informações, como: certificados que assegurem que o usuário possui determinado papel; ou combinações de solução de desafios relacionados a dispositivos de um local (*Bluetooth Beacons*, por exemplo) para comprovar uma localização, entre outros.

Também é importante garantir que esses dados de contexto sejam atualizados frequentemente, ou ao menos antes de realizar requisições, para que uma permissão não seja emitida baseada em um contexto no qual o usuário não se encontra mais.

3.2.5 Access

Access é mais um *Asset* definido no modelo, e representando o direito de acesso obtido. Um *Access* só é gerado se um *Requester* que realizar uma requisição de acesso (*RequestAccess*) satisfizer as condições definidas na política de acesso do recurso que queira acessar.

O *Access* criado é um registro referenciando o *Resource* requisitado e o *Requester* que satisfaz suas condições de acesso. Portanto, um *Access* vincula um *Requester* a um *Resource*. Assim, pode-se determinar que um *Requester* só tem permissão de acesso sobre um *Resource* se existir um *Access* vinculando os dois. Nesse sentido, a existência deste *Access* é comparável com a posse de um *token* lógico para acesso.

3.2.6 Transactions

As transações são as operações sobre a rede. No *Hyperledger*, toda transação tem uma entrada com dados novos e/ou dados já registrados, e retorna a criação, leitura, ou atualização de

informações para registrar na *blockchain*, ou ainda apagá-las. As transações são definidas como funções contidas no *chaincode* (*smart contract*). É importante ressaltar que a real aplicação das transações só ocorre caso haja consenso dos *peers* de endosso quanto ao resultado da execução.

Três transações primordiais foram definidas no modelo aqui proposto, sendo elas *AddResource*, *ComposeContext* e *RequestAccess*.

AddResource é a transação pela qual um Usuário (*ResourceOwner*) registra um recurso ao qual pretende conceder acesso perante determinadas condições. As entradas são as informações do recurso e as definições da política de acesso, gerando registro de um objeto *Resource* com uma política de acesso (*AcessPolicy*) vinculada. Um protocolo das operações realizadas na transação *AddResource* é mostrado na Figura 3.2.

Protocolo da transação AddResource

Recebe as entradas: *resourceId*, *address*, *accessPolicy*, *owner*;

Cria um objeto *Resource* com o identificador *resourceId*;

Adiciona o endereço *address* ao *Resource*;

Adiciona ao *Resource* as definições de política de acesso contidas em *accessPolicy*;

Adiciona o proprietário *owner* ao *Resource*;

Adiciona o *Resource* a lista de recursos do *owner*;

Emite o registro de *Resource*;

Emite a atualização do *ResourceOwner*;

Figura 3.2: Protocolo da transação *AddResource*, para registros de recursos.

No protocolo de *AddResource*, recebe-se como entrada as informações do recurso que será cadastrado. Assim, cria-se os objetos *Resource* e *AccessPolicy* com as informações recebidas e depois utiliza-se as operações de registro do *Hyperledger* para gravar no *blockchain* o novo recurso gerado com sua política de acesso vinculada, e atualizar as informações do proprietário adicionando o recurso à sua lista de recursos.

ComposeContext é a transação que um requerente deve fazer para registrar suas atuais variáveis de contexto. Esta transação tem as informações do requerente como entrada e gera um objeto do tipo *Context* que é tanto usado nas requisições de acesso, quanto permanece registrado para futuras verificações e audições. O protocolo das operações na transação *ComposeContext*

é descrito na Figura 3.3.

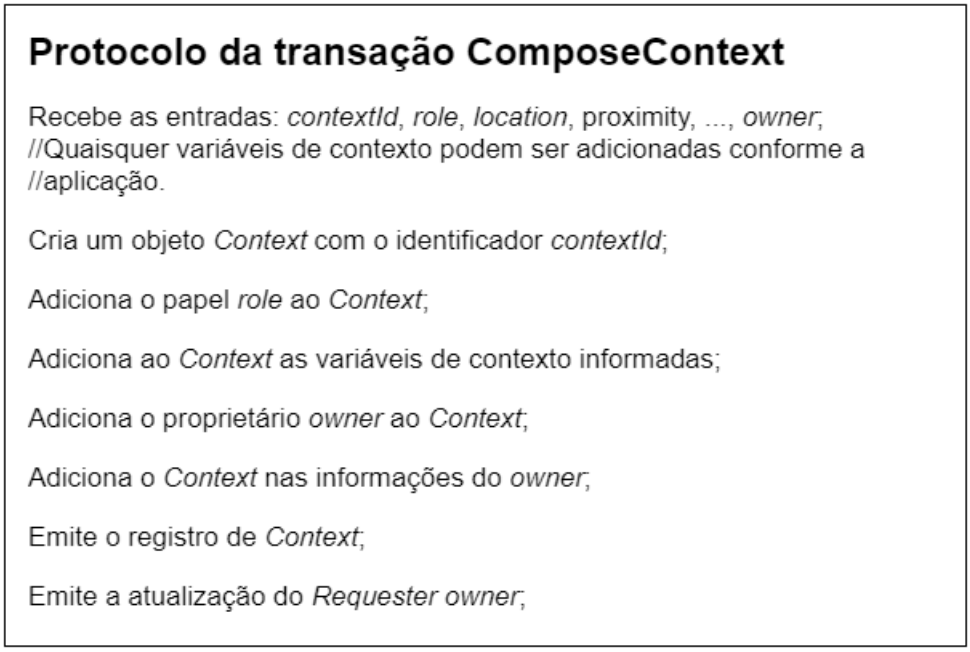

Figura 3.3: Protocolo da transação *ComposeContext*, para atualização de contexto.

No protocolo de *ComposeContext*, recebe-se como entrada o papel do *Requester*, suas informações de contexto e seu registro. Então, gera-se um objeto *Context* e adiciona-se nele as informações recebidas como entrada. Por fim, utiliza-se as operações do *Hyperledger* para registrar o objeto *Context* na *blockchain* e atualizar este contexto no registro do *Requester*.

RequestAccess é a transação em que um requerente solicita o acesso a um determinado recurso. O requerente aponta suas informações, como suas variáveis de contexto e o recurso que pretende acessar como entrada para a transação.

As verificações se o requerente satisfaz as regras de acesso são definidas nas operações dos *smart contracts* (*chaincodes*). O código da transação verifica se o contexto atual do requerente satisfaz as condições estabelecidas na política de acesso do recurso. Se as condições não são satisfeitas, nega o acesso. Caso as condições forem satisfeitas, a transação emite um acesso para o requerente e cria um objeto *Access* para manter o registro daquela ação. O protocolo da transação é descrito na Figura 3.4.

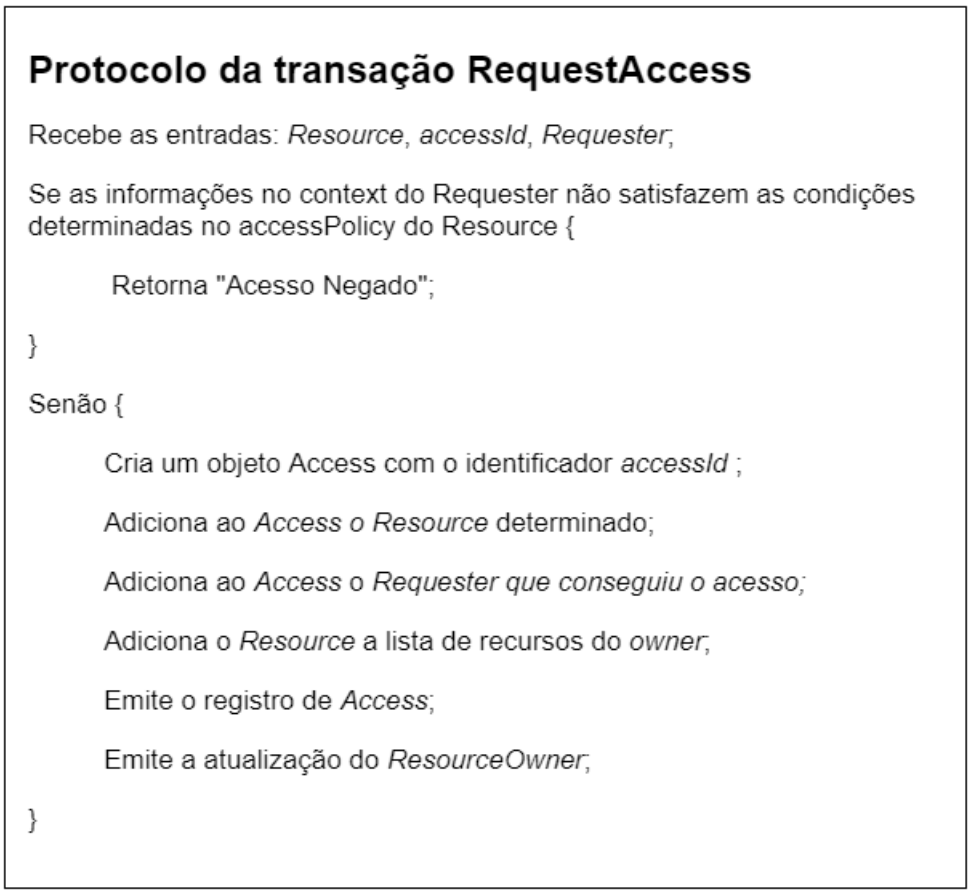

Figura 3.4: Protocolo da transação *RequestAccess*, para requisição de permissão de acesso.

Estas transações utilizam o mecanismo do *Hyperledger* disponibilizado para a definição de permissões sobre operações nos objetos configurados na rede. Estas permissões são descritas em um arquivo comumente nomeado como permissions.acl, onde define-se as operações que cada tipo de participante pode fazer sobre qual tipo de recurso ou transação. As permissões podem ser de criação, leitura, atualização e remoção (*CREATE*, *READ*, *UPDATE* e *DELETE*, ou ainda *ALL*, para denotar todas permissões).

Para exemplificar, a transação *RequestAccess* apresentada não gera diretamente o direito de acesso, mas gera um registro *Access* indicando que o *Requester* vinculado satisfaz as condições de acesso do *Resource*. O *Access* gerado é adicionado também ao conjunto de acessos obtidos (*Accesses*) do *Requester* que fez a solicitação. O acesso só ocorre de fato porque no arquivo *permissions.acl* define-se a permissão real, em forma de uma regra que expressa que um *Requester* só poderá ler e atualizar informações (permissões *READ* e *UPDATE*) de um *Resource* caso haja um registro na *blockchain* de um *Access* com o identificador deste *Requester* e o identificador deste *Resource*.

3.3 CAAC com smart contracts baseado em transferências de recursos

As transações propostas anteriormente são maneiras "estáticas" de definição de *smart contracts*, ou seja, foram inseridas na rede desde o momento de sua implementação e ficam disponíveis para serem utilizadas. Para usar essas transações as operações serão do tipo *invoke transactions* e serão invocadas durante as operações na rede.

No *Hyperledger Fabric*, existe um outro tipo de transação que são as *deploy transactions*. Estas transações permitem a criação dinâmica de novos *smart contracts* (*chaincodes*) para atuarem sobre a rede, o que gera um potencial novo de flexibilidade nas operações com alto nível de funcionalidades sobre a *blockchain*.

Para criar um novo *chaincode*, durante a execução da rede, é necessário utilizar o pacote *ChaincodeDeploymentSpec* da *API* do *Hyperledger*. Dessa forma, cria-se uma *deploy transaction* que, para ser aplicada, precisa incluir o código fonte do *chaincode*, as políticas de instanciação e execução das operações e uma lista de entidades que concordam com esse *chaincode*.

O código fonte do *chaincode* pode ser escrito em linguagem *Go*, *Java*, ou *node.js* e os dados são definidos a partir de um schema. Os *schemas* para o *chaincode* podem ainda ser associados a ontologias de controle de acesso, como as apresentadas nos trabalhos relacionados, aumentando ainda mais suas capacidades.

Um *schema* associado à linguagem de programação proporciona a um usuário criar um *chaincode* definindo como ele irá cadastrar seus recursos na rede *blockchain* e como oferecer acesso e operações sobre ele de forma diferente das transações apresentadas.

Supondo, por exemplo, que um usuário queira registrar um recurso e controlar o acesso sobre ele. Isso poderá ocorrer, com a definição do modelo de dados com auxílio de JSON, como na Figura 3.5.

Esquema de dados

```
//custom data models
type PersonalInfo struct {
    Firstname string `json:"firstname"`
    Lastname  string `json:"lastname"`
    DOB       string `json:"DOB"`
    Email     string `json:"email"`
    Mobile    string `json:"mobile"`
}
```

Figura 3.5: Uso de JSON para modelagem de dados.

Na Figura 3.5, utiliza-se um modelo JSON para auxiliar na definição de uma estrutura representando informações pessoais.

Juntamente às especificações dos dados, deverá ser provido um código fonte para o *chaincode*, definindo a lógica de operação deste.

```
concept GeoLocation {
        o String address optional
        o Double elevation optional
        o Double latitude optional
        o Double longitude optional
}
```

Figura 3.6: Definição de um *concept* representando localização.

Na Figura 3.6, mostra-se como pode ser definida uma estrutura para representar a localização de um usuário.

A possibilidade de trabalhar com *chaincodes* propostos durante a execução, possibilita a criação de um sistema de controle de acesso que possa variar com o tempo, já que novas lógicas de operação podem ser incluídas por esses *chaincodes*.

Um modelo de controle de acesso que varie com o tempo, traz a necessidade de adaptação no tratamento das informações de contexto. Novas informações de contexto podem ser

modeladas, e ter novas lógicas de validação.

Tendo essas necessidades em consideração, propõe-se um modelo alternativo, utilizando múltiplos *Assets* para representar um contexto, onde cada *Asset* se refere a prova de uma informação de contexto diferente.

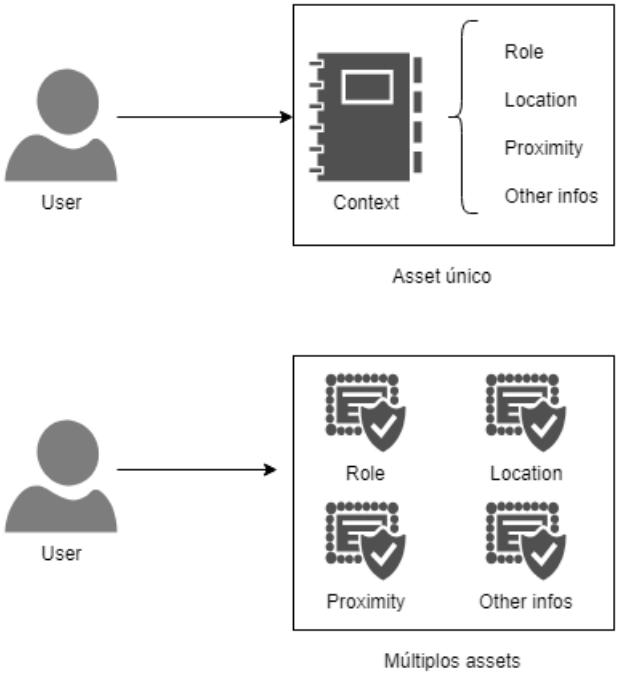

Figura 3.7: Diferença entre os modelos.

Na Figura 3.7, mostra-se a diferença entre o modelo anteriormente proposto, com apenas um *Asset* representando todo o contexto do usuário, e o modelo alternativo, com um *Asset* para cada informação de contexto.

A existência de um *Asset* único representando todas informações de contexto, limita o processo de controle de acesso às variáveis definidas no *Asset*. Ao representar diferentes informações de contexto com *Assets* independentes, torna-se possível a introdução de modelagens de novas variáveis de contexto, já durante a operação da rede, para serem utilizadas.

Neste modelo, ao invés de um *Requester* possuir um *Asset* do tipo *Context* obrigatório para provar seu contexto, o *Requester* passa a ser proprietário (*owner*) de vários *Assets*, cada um comprovando uma informação de contexto diferente. E para obter acesso a um serviço, o *Requester* deverá transferir os *Assets* requisitados para o gerente do serviço, comprovando satisfazer as condições de acesso.

A modelagem de novas informações de contexto e a lógica de validação e utilização delas,

pode ser feita ao propor novos *chaincodes* para a rede.

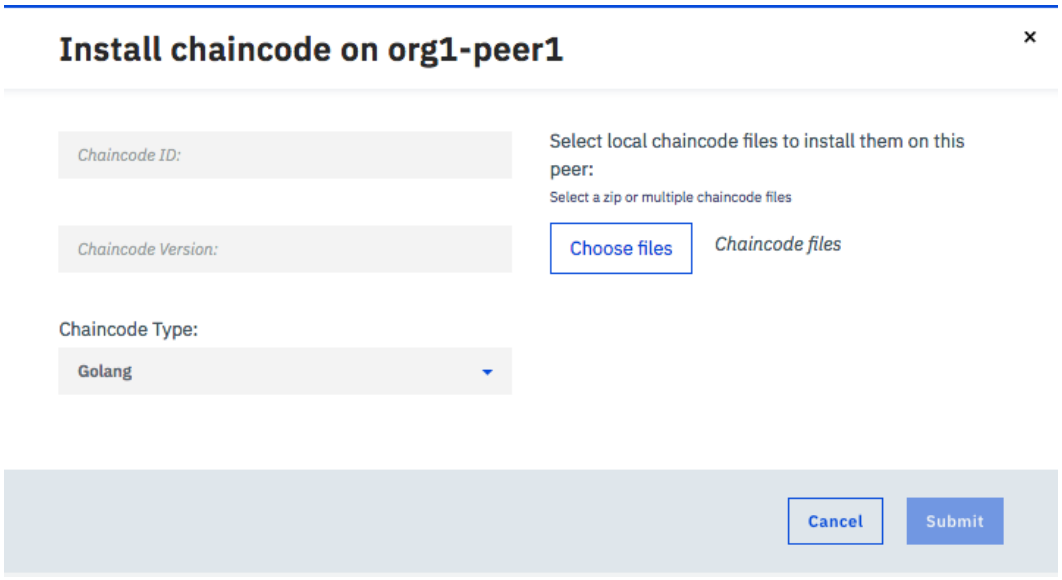

Figura 3.8: Instalação de *chaincode* através de ferramenta da IBM.

A Figura 3.8, mostra um exemplo de uso de uma ferramenta da IBM para auxiliar na instalação de novos *chaincodes* na rede. Com esta ferramenta, pode-se atribuir um identificador e uma versão para um *chaincode*, além de especificar a linguagem de programação utilizada e selecionar os arquivos com os códigos fonte para o *chaincode*.

3.4 Estruturas do sistema Hyperledger

As duas estratégias de controle de acesso propostas nesse trabalho, usando um ou vários Assets, consideram a organização de um sistema composto de algumas estruturas específicas: nós, estado, *ledger* e *chaincodes*.

3.4.1 Nós

Os nós são as entidades em comunicação na rede *blockchain*. São apenas unidades lógicas, já que vários nós de diferentes tipos podem estar alocados no mesmo servidor físico.

Há três tipos de nós: cliente (*client* ou *submitting-client*), *peer* e *orderer* (ou *ordering service node*).

Os nós clientes são aqueles que enviam propostas de transações aos endossadores e propagam propostas de transações endossadas ao *ordering service*. Um cliente é o nó que atua em função do usuário final que deseja interagir com a rede *blockchain*.

Um nó cliente é capaz de criar e invocar transações através de funções. Ele pode se comunicar tanto com os *peers*, quanto com o *ordering service*.

Os *peers* são os nós responsáveis pela aplicação das transações e por manter o estado e sua própria cópia do *ledger*. As atualizações de estado são recebidas de maneira ordenada na forma de blocos vindos do *ordering service*.

Peers também podem assumir o papel de endossar transações, podendo ser chamados de *endorsers* (ou *endorsing peer*). Este papel consiste em:

- uma escuta permanente por propostas de transações;
- endossar as propostas de transação em relação a um *chaincode* específico, utilizando as políticas de endosso contidas neste *chaincode*;
- aguardar comando de aplicação da transação (comando gerado pelo *ordering service*), caso ela tenha sido endossada pelo número suficiente de *endorsing peers*.

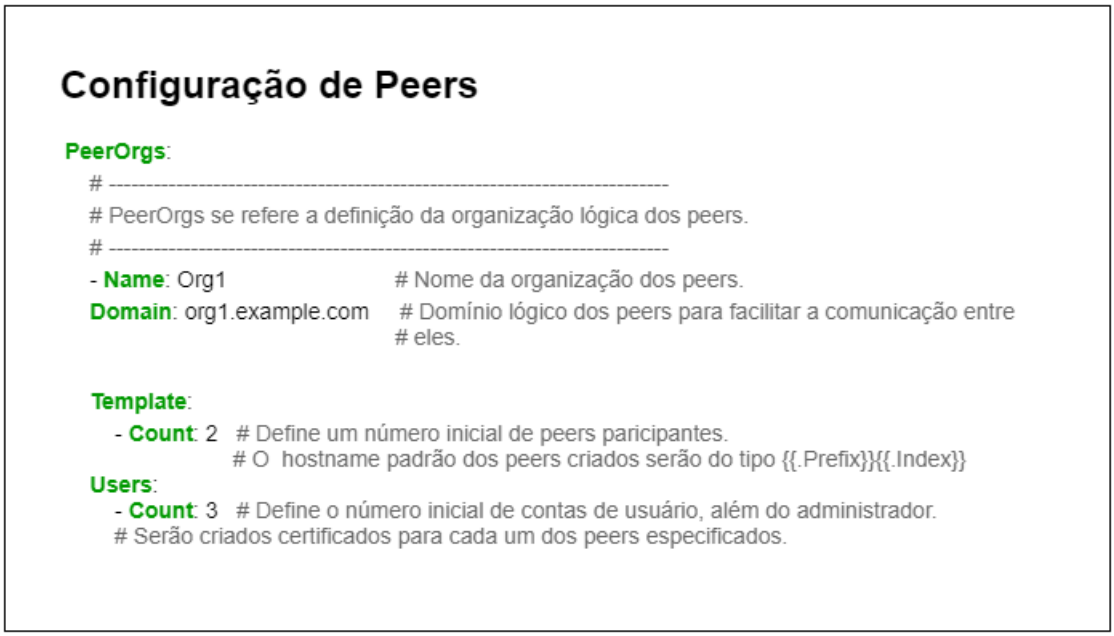

Figura 3.9: Documento de configuração de peers.

No sistema *Hyperledger Fabric*, os *peers* iniciais da rede são configurados a partir de um arquivo chamado *crypto-config.yaml*, como exemplificado na Figura 3.9, que determina a criação dos certificados e assinaturas dos *peers*. A configuração de *peers*, assim como a criação de novos, pode ser modificada dinamicamente. Tais *peers* são candidatos a atuarem como *endorsing peers* para os serviços na rede. A informação do *Domain*, aqui apresentada como *example.com*,

na verdade vai depender do domínio DNS em que a aplicação estiver sendo executada. Espera-se que o serviço DNS ajude na resolução de nomes e endereços.

Os *orderers* são os nós responsáveis por executar o *ordering service* (serviço de ordenação), um serviço de comunicação que implementa a garantia de entrega e a propagação (*atomic broadcast* ou *total order broadcast*).

O *ordering service* pode ser implementado como um serviço centralizado, ou com variados tipos de protocolos distribuídos. O *ordering service* fornece um canal de comunicação compartilhado para os clientes e *peers*.

Um cliente se conecta ao canal oferecido pelo *ordering service* e propaga neste canal suas mensagens, que serão entregues aos *peers*. O *ordering service* garante também que as mensagens serão entregues na mesma ordem lógica para todos os *peers*, a fim de buscar-se um consenso na comunicação.

Dessa forma, as funções dos *orderers* são:

- receber propostas de transações já endossadas;
- verificar se os *endorsers* realmente endossaram a transação;
- caso o endosso seja válido, emite aos *peers* o comando de aplicação da transação.

Configuração de Orderer Peers

```
OrdererOrgs:
  # ---------------------------------------------------------------------------
  # OrdererOrgs se refere a definição da organização lógica dos nós orderers.
  # ---------------------------------------------------------------------------
  - Name: Orderer          # Nome da organização dos orderers peers.
  Domain: example.com      # Domínio lógico dos peers para facilitar a comunicação entre eles.

  Specs:
    - Hostname: orderer1   # Define um orderer peer chamado orderer1.
    - Hostname: orderer2   # Define um orderer peer chamado orderer2.
    - Hostname: orderer3   # Define um orderer peer chamado orderer3.
    # Serão criados certificados para todos orderers especificados..
```

Figura 3.10: Documento de configuração de ordering peers.

Os *orderers* iniciais também são configurados pelo arquivo *crypto-config.yaml*, como exemplificado na Figura 3.10, gerando os certificados para estes. Novos *orderers* também podem ser

adicionados dinamicamente, assim como os outros *peers*.

A geração efetiva do certificados para estes nós iniciais ocorre através da invocação da função de geração pelo comando:

cyptogen generate --config=./crypto-config.yaml.

3.4.2 Estado

O estado do sistema no *blockchain* é definido pelo armazenamento de versões de pares chave-valor.

Formalmente, um estado s é um elemento do mapeamento $K->(VXN)$, onde K é um conjunto de chaves k, V é um conjunto de valores v, e N é um conjunto infinito ordenado de números de versão *ver*. Assim, um estado *s(k)* é definido pelo valor v e a versão *ver*, sendo:

$$s(k) = (v, ver)$$

Estas entradas podem ser manipuladas por *chaincodes* durante a execução, através de operações *put* e *get*.

3.4.3 Ledger

O *Ledger* é o livro de registros do sistema, que provê um histórico verificável de todas as mudanças de estado, efetuadas por transações válidas, e das tentativas não sucedidas de mudança de estado, pelas transações inválidas, durante o funcionamento do sistema.

O *Ledger* é uma *hashchain*, totalmente ordenada, de blocos com transações válidas ou inválidas. As transações também são totalmente ordenadas dentro de um *array* contido no bloco, o que impõe a ordenação total de todas as transações.

O primeiro bloco da cadeia é chamado de bloco gênesis (*genesis block*). No *Hyperledger*, cria-se o bloco gênesis a partir do arquivo *configtx.yaml*, onde podem ser definidos o intervalo de tempo em que cada lote de mensagens será gerado (*BatchTimeout*), o número máximo de mensagens em um lote (*MaxMessageCount*), o limite máximo de tamanho para cada mensagens (*AbsoluteMaxBytes*), o tamanho máximo preferível para cada mensagem (*PreferedMaxBytes*). Dessa forma, os blocos serão gerados com um lote de mensagens e registrados na *blockchain*.

As informações no *ledger* poderão ser acessadas pelas funções de transações nos *chaincodes*.

Configurando o genesis block

```
# Profile
Profiles:
    OneOrgOrdererGenesis:
        Orderer:
            <<: *OrdererDefaults
            Organizations:
                - *OrdererOrg
        Consortiums:
            SampleConsortium:
                Organizations:
                    - *Org1
    OneOrgChannel:
        Consortium: SampleConsortium
        Application:
            <<: *ApplicationDefaults
            Organizations:
                - *Org1
# Organizations - defines organization identities
Organizations:
    - &OrdererOrg
        Name: OrdererOrg
        ID: OrdererMSP
        MSPDir: crypto-config/ordererOrganizations/example.com/msp
    - &Org1
        Name: Org1MSP
        ID: Org1MSP
        MSPDir: crypto-config/peerOrganizations/org1.example.com/msp
        AnchorPeers:
            - Host: peer0.org1.example.com
              Port: 7051
# Orderer - defines the values to encode into a config transaction or genesis block
Orderer: &OrdererDefaults
    OrdererType: solo
    Addresses:
        - orderer.example.com:7050
    BatchTimeout: 2s              # The amount of time to wait before creating a batch
    BatchSize:                    # Controls the number of messages batched into a block
        MaxMessageCount: 10       # The maximum number of messages to permit in a batch
        AbsoluteMaxBytes: 99 MB   # The absolute maximum number of bytes allowed for
                                  # the serialized messages in a batch.
        PreferredMaxBytes: 512 KB # The preferred maximum number of bytes allowed for
                                  # the serialized messages in a batch. A message larger than the preferred
                                  # max bytes will result in a batch larger than preferred max bytes.
    Kafka:
        Brokers:
            - 127.0.0.1:9092
    Organizations:
# Application - define the values to encode into a config transaction or genesis block
Applications: &ApplicationDefaults
    Organizations:
```

Figura 3.11: Documento de configuração inicial da rede com o *genesis block*.

Na Figura 3.11 demonstra-se a definição de uma rede *blockchain* exemplo com apenas uma organização participante e um canal. Defini-se a única organização participante, chamada *Org1*, através de *OneOrgOrdererGenesis.* O canal único é definido através de *OneOrgChannel.* Os diretórios com as identificações dos *orderers* e dos *peers* previamente definidos no *crypto-config*

são passadas como referência nos campos *MSPDir*. Então define-se a configuração dos blocos para lotes criados a cada 2 segundos (*BatchTimeout*), com até 10 mensagens (*MaxMessageCount*), em que cada mensagem tenha preferencialmente 512 KB (*PreferredMaxBytes*), mas pode alcançar um tamanho máximo de 99 MB (*AbsoluteMaxBytes*). Tais configurações precisam ser adequadas a cada caso de aplicação, levando em conta a escalabilidade e desempenho da rede. Também deve ser definido um modo de atuação para o *ordering service*, como *solo* ou *kafka*, ou ainda uma implementação própria.

A partir da geração deste documento, pode se construir o *genesis block* através do comando:

./configtxgen -profile OneOrgOrdererGenesis -outputBlock ./channel-artifacts/genesis.block

Logo em seguida, deve-se gerar o canal que será utilizado. Isto é feito utilizando o comando:

./configtxgen -profile OneOrgChannel outputCreateChannelTx ./channel-artifacts/channel.tx -channelID testchannel.

Todos os *peers* e um subconjunto dos *orderers* mantêm em si o *Ledger*. O *Ledger* em um *peer* é chamado de *PeerLedger*, enquanto o *Ledger* em um *orderer* é chamado de *OrdererLedger*. A diferença entre esses *Ledgers* é que na sua versão local os *peers* mantêm uma *bitmask* para diferenciar transações válidas de inválidas.

Os *peers* podem aparar seu *PeerLedger*, normalmente se aproveitando da *bitmask* para poder remover as transações inválidas, já que estas transações não geram mudanças de estado.

Os *orderers* mantêm seu *OrdererLedger* para tolerância a falhas e disponibilidade, podendo apenas apará-lo caso haja garantia de que as propriedades do *ordering service* sejam mantidas.

3.4.4 Chaincodes

Chaincodes são a interpretação do *Hyperledger Fabric* para os *smart contracts*, visto que eles são responsáveis por conter uma lógica de execução de comum acordo entre membros da rede. Um *smart contract* pode definir termos, dados, regras, conceitos e processos, e define uma lógica executável para operações sobre o *ledger*.

Com um *smart contract* pode-se implementar um conjunto de regras, para qualquer tipo de objeto lógico, como programas executáveis e estas regras serão cumpridas durante a execução.

Um mesmo *chaincode* pode conter mais que um *smart contract*, visto que as operações

deles são descritas em forma de funções em linguagem de programação como *GoLang*, *Java* ou *JavaScript* e um *chaincode* pode ter várias funções definidas.

Os *chaincodes* podem ser definidos na implantação da rede, para serem invocados posteriormente, ou podem ser definidos já durante o funcionamento da rede, através de transações de implantação (*deploy transactions*). Para definir um *chaincode* por transação de implantação, o *peer* que a submeter, deverá passar o código fonte do *chaincode* e a modelagem dos dados que serão utilizados. A Figura 3.12 mostra as propriedades necessárias para a instalação de um *chaincode*, conforme a *SDK* do *Hyperledger Fabric*.

ChaincodeInstallRequest

Type:

- Object

Properties:

Name	Type	Description
targets	Array.<Peer> | Array.<string>	Optional. An array of Peer objects or peer names where the chaincode will be installed. When excluded, the peers assigned to this client's organization will be used as defined in the common connection profile. If the 'channelNames' property is included, the target peers will be based the peers defined in the channels.
chaincodePath	string	Required. The path to the location of the source code of the chaincode. If the chaincode type is golang, then this path is the fully qualified package name, such as 'mycompany.com/myproject/mypackage/mychaincode'
metadataPath	string	Optional. The path to the top-level directory containing metadata descriptors.
chaincodeId	string	Required. Name of the chaincode
chaincodeVersion	string	Required. Version string of the chaincode, such as 'v1'
chaincodePackage	Array.<byte>	Optional. Byte array of the archive content for the chaincode source. The archive must have a 'src' folder containing subfolders corresponding to the 'chaincodePath' field. For instance, if the chaincodePath is 'mycompany.com/myproject/mypackage/mychaincode', then the archive must contain a folder 'src/mycompany.com/myproject/mypackage/mychaincode', where the chaincode source code resides.
chaincodeType	string	Optional. Type of chaincode. One of 'golang', 'car', 'node' or 'java'. Default is 'golang'.
channelNames	Array.<string> | string	Optional. When no targets are provided. The loaded common connection profile will be searched for suitable target peers. Peers that are defined in the channels named by this property and in this client's organization and that are in the endorsing or chain code query role on the named channel will be selected.
txId	TransactionID	Optional. TransactionID object for this request.

Figura 3.12: Propriedades necessárias para instalar um *chaincode*.

Um *chaincode* é instalado por um peer e é instanciado em um canal, assim os *peers* associ-

ados naquele canal iniciarão contêineres para execução do *chaincode*.

Utilizando como exemplo uma rede *blockchain* para registros e transações com carros, mostrada na documentação do *Hyperledger Fabric*, um *chaincode* terá código e funções como na Figura 3.13.

```
async createCar(ctx, carNumber, make, model, color, owner) {

    const car = {
        color,
        docType: 'car',
        make,
        model,
        owner,
    };

    await ctx.stub.putState(carNumber, Buffer.from(JSON.stringify(car)));
}
```

Figura 3.13: Exemplo de função em um chaincode.

Na Figura 3.13, define-se uma função chamada *createCar*. A função foi definida em código *JavaScript* e registra um objeto '*car*' (carro) na *blockchain*, com as informações recebidas como entrada. A chamada a essa função equivale a uma transação na rede.

Ainda utilizando o mesmo exemplo, um *chaincode* pode representar toda a lógica de operações com carros nessa rede.

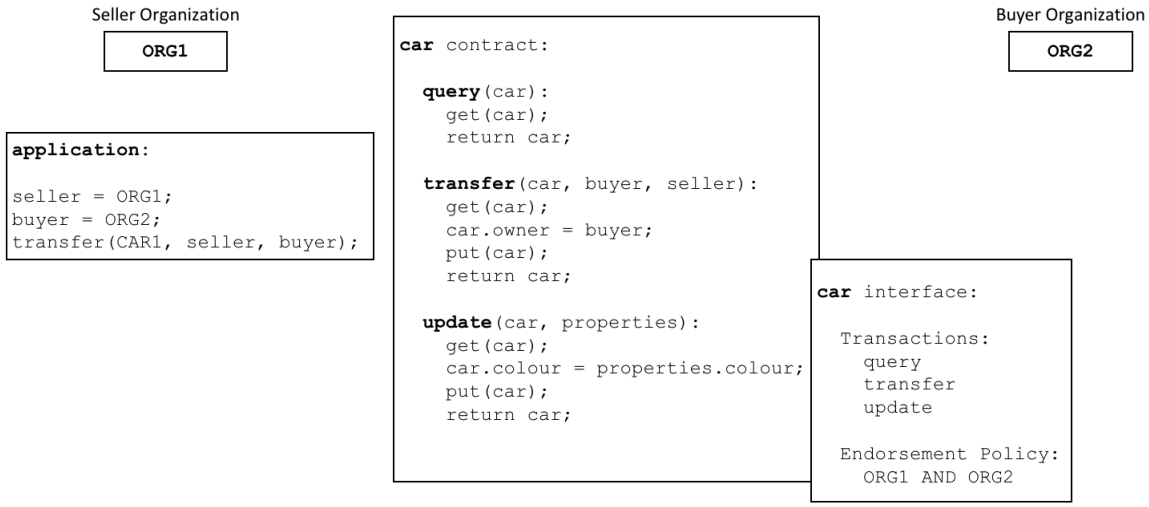

Figura 3.14: Exemplo de estrutura lógica de uma rede.

Na Figura 3.14 representa-se a estrutura lógica da rede de carros, com funções como busca (*query*), transferência (*transfer*) e atualização (*update*) de dados dos carros, entre os membros

chamados de *ORG1* e *ORG2*.

Também denota-se uma política de endosso *"ORG1 AND ORG2"*, que representa que os membros *ORG1* e *ORG2* devem endossar as transações realizadas. Ou seja, ambos os membros devem executar a transação, chegar ao mesmo resultado e assinar o endosso, confirmando um consenso entre eles.

As políticas de endosso de um *chaincode* definem quais *peers* precisam executar e assinar a transação. A transação só é considerada como endossada de fato se as regras da política de endosso forem cumpridas.

Uma política de endosso para um *chaincode* pode ser definida no momento da instanciação deste *chaincode*. Por exemplo, no momento da instanciação de um *chaincode* "*mycc*" atribui-se a política de endosso *"AND('Org1.peer', 'Org2.peer')"* através do seguinte comando:

peer chaincode instantiate -C <channelid> -n mycc -P "AND('Org1.peer', 'Org2.peer')"

Este comando torna obrigatório que as transações no chaincode sejam endossadas pelos dois peers, *Org1* e *Org2*.

Além da política com *AND*, as políticas de endosso podem ser do tipo *OR* e ainda ter combinações mais complexas, com variações como:

- *"OR('peer1', 'peer2')"* representando a necessidade de endosso pelo *peer1* ou *peer2*;
- *"OR('peer1', AND('peer2', 'peer3'))"* onde exige a assinatura de endosso do *peer1*, ou as assinaturas do *peer2* junto com a do *peer3*.

Essas políticas de endosso são ideais para se definir o tipo de consenso desejado na execução do *chaincode*, sendo possível implementar até mesmo um sistema de reputação para *peers*, atribuindo diferentes pesos para a assinatura de endosso deles.

A IBM que é membro participante no desenvolvimento do *Hyperledger* disponibiliza algumas ferramentas para auxiliar o uso desta plataforma *blockchain*. Uma dessas ferramentas oferece gerenciamento da instanciação de *chaincodes*, como na Figura 3.15.

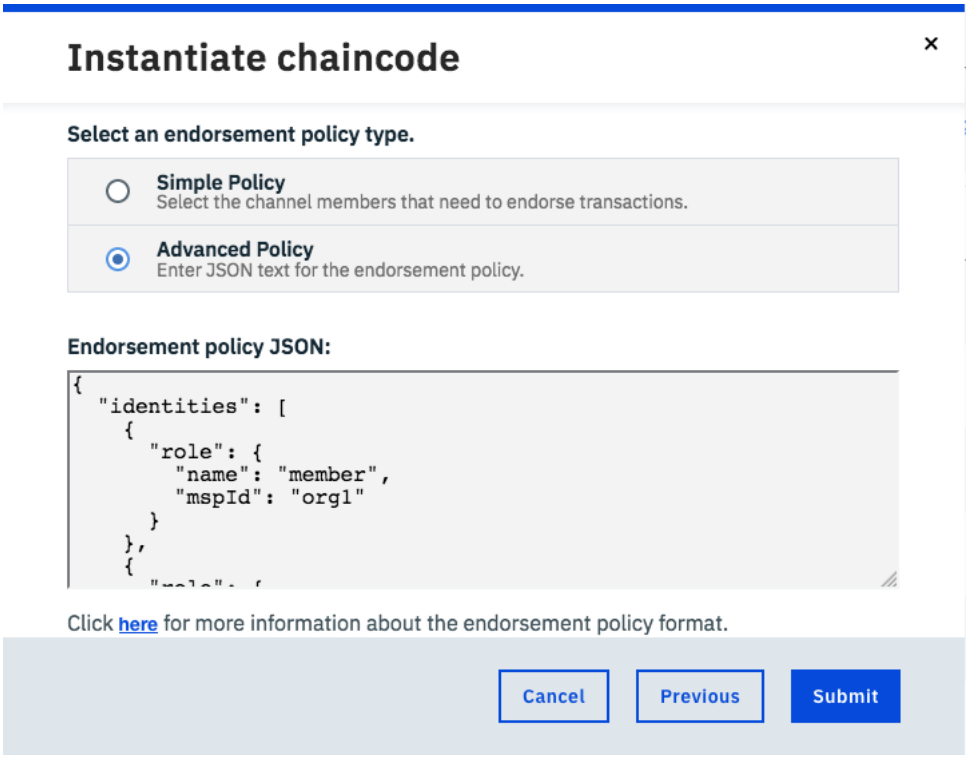

Figura 3.15: Definição de política de endosso na instanciação de chaincode, através de ferramenta da IBM.

Regras de endosso mais complexas de serem descritas podem ser modeladas em forma de JSON, como na Figura 3.16.

```
{
  identities: [
    { role: { name: "member", mspId: "org1" }},
    { role: { name: "member", mspId: "org2" }}
  ],
  policy: {
    "1-of": [{ "signed-by": 0 }, { "signed-by": 1 }]
  }
}
```

Figura 3.16: Política de endosso definida através de JSON.

A Figura 3.16 mostra a especificação de uma política de endosso *"1-of"* que determina que o endosso é válido quando assinado por qualquer membro de qualquer uma das organizações determinadas, no caso *org1* e *org2*.

As sentenças das políticas de endosso devem conter expressões lógicas que possam ser avaliadas como *TRUE* ou *FALSE*. A condição usará assinaturas digitais nas chamadas das transações emitidas pelos *peers* de endosso para o *chaincode*.

Supondo que em um *chaincode* seja especificado um conjunto de membros (*peers*) *E = {Alice, Bob, Charlie, Dave, Eve, Frank, George}*, pode-se produzir políticas como as seguintes:

- Requerer uma assinatura válida de todos os membros de *E* para a mesma proposta de transação.
- Requerer uma única assinatura válida de qualquer membro de *E*.
- Requerer assinaturas válidas dos *peers* para a mesma proposta de transação em condições como *(Alice OR Bob) AND (any two of: Charlie, Dave, Eve, Frank, George)*.
- Requerer assinaturas válidas para a mesma proposta de transação de qualquer combinação de cinco membros do total de sete do conjunto *E*. Pode ser usada para políticas de endosso por maioria dos membros.
- Atribuir pesos para os *peers* de endosso e exigir uma pontuação de peso que represente maioria em assinaturas válidas. Exemplo: Atribuir *{Alice=49, Bob=15, Charlie=15, Dave=10, Eve=7, Frank=3, George=1}* totalizando *100*, e exigir que as assinaturas de endosso correspondam à mais que *50*.
- As atribuições de pesos podem ser estáticas, fixadas nos metadados do *chaincode*, ou dinâmicas, sendo dependente do estado do *chaincode* e podendo ser modificada durante a execução.

A utilidade de tais tipos de políticas depende da forma de aplicação e das propriedades de segurança desejadas, como resiliência, solução de falhas e comportamento dos *peers*.

3.5 Transações e Registros usando Hyperledger

Nesta seção serão abordados aspectos técnicos sobre as transações e registros na plataforma *Hyperledger* utilizada na proposta apresentada, sendo que a maioria dessas operações é feita de forma transparente aos usuários.

3.5.1 Formato de proposta de transação

O fluxo básico de endosso de uma transação inicia-se com o Cliente criando uma proposta de transação *tx* . Tal proposta é composta pelo identificador do cliente que está submetendo a transação (*clientID*), o identificador do *chaincode* ao qual ela se refere (*chaincodeID*), um

conjunto de valores que depende do tipo de transação (*txPayload*), um marcador temporal (*timestamp*), e a assinatura do cliente que criou a proposta (*clientSig*).

$$tx =< clientID, chaincodeID, txPayload, timestamp, clientSig >$$

Os detalhes contidos no campo *txPayload* dependem do tipo de transação. No caso de transação de invocação (*invoke transaction*) há dois campos: o campo *operation*, especificando a operação / função e os argumentos do *chaincode* invocado; e outro campo chamado de *metadata*, contendo os atributos relacionados à invocação. No caso de transação de implantação (*deploy transaction*) há três campos: o campo *source*, com o código fonte do *chaincode*; o *metadata*, com os atributos relacionados ao *chaincode* e a aplicação; e o *policies*, contendo as políticas relacionadas ao chaincode que serão acessíveis a todos *peers*.

Após a composição da proposta de transação, o Cliente a envia para *peers* de endosso, conforme na Figura 3.17.

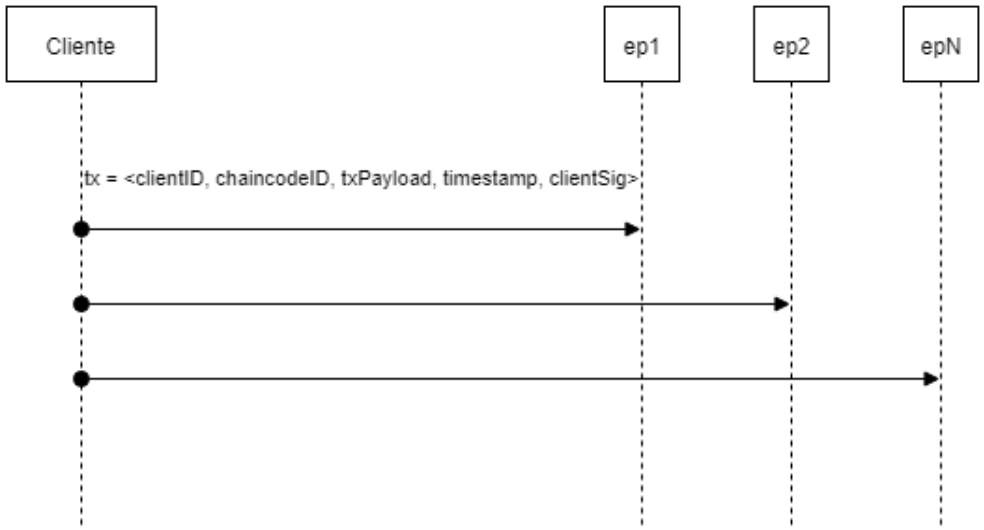

Figura 3.17: Proposta de transação encaminhada aos *peers* pelo cliente.

3.5.2 Simulação de transação e assinatura de endosso

Ao receber a mensagem de um cliente, o *peer* de endosso (*epID*) verifica a assinatura do cliente (*clientSig*) e simula a execução da transação. Essa simulação consiste em tentar executar a transação com o *txPayload* recebido, invocando o *chaincode* referenciado (*chaincodeID*) e a copia do *ledger* da cadeia que o peer mantém localmente (*PeerLedger*).

O *peer* irá utilizar um container para execução isolada do *chaincode*. Nesse ponto são

computadas as dependências (*readset*) e atualizações de estado (*writeset*) que seriam geradas.

Neste momento, a proposta será avaliada de acordo com a lógica de endosso que foi instanciada para o *chaincode*, mantida no *peer*.

Em caso de endosso, o *peer* envia ao cliente uma mensagem contendo uma *flag* que representa o endosso da transação (*TRANSACTION-ENDORSED*), o identificador da transação (*tid*), a proposta de transação (*tran-proposal*) e a assinatura do *peer* de endosso (*epSig*). A mensagem estará no formato seguinte:

$$< TRANSACTION - ENDORSED, tid, tran - proposal, epSig >$$

O envio da mensagem é demonstrado na Figura 3.18.

Figura 3.18: Resposta dos *peers* quando a transação é endossada.

A proposta de transação, por sua vez, é composta pelo identificador do *peer* de endosso (*epID*), o identificador da transação (*tid*), o identificador do *chaincode* (*chaincodeID*), informações específicas do *chaincode* (*txCotentBlob*), um conjunto de dependências da versão de leitura (*readset*) e o conjunto de atualizações de estados (*writeset*).

$$tran - proposal := (epID, tid, chaincodeID, txContentBlob, readset, writeset)$$

É importante ressaltar que, embora ocorra a geração do *writeset* durante a simulação da transação, nenhuma mudança de estado é aplicada nessa fase.

Em caso da lógica não endossar a transação, o *peer* enviará ao cliente uma mensagem

com uma *flag* indicando que a transação não pode ser validada (*TRANSACTION-INVALID*), o identificador da transação (*tid*), e uma *flag* indicando a rejeição da proposta de transação (*REJECTED*).

$$< TRANSACTION - INVALID, tid, REJECTED >$$

O envio da mensagem é demonstrado na Figura 3.19.

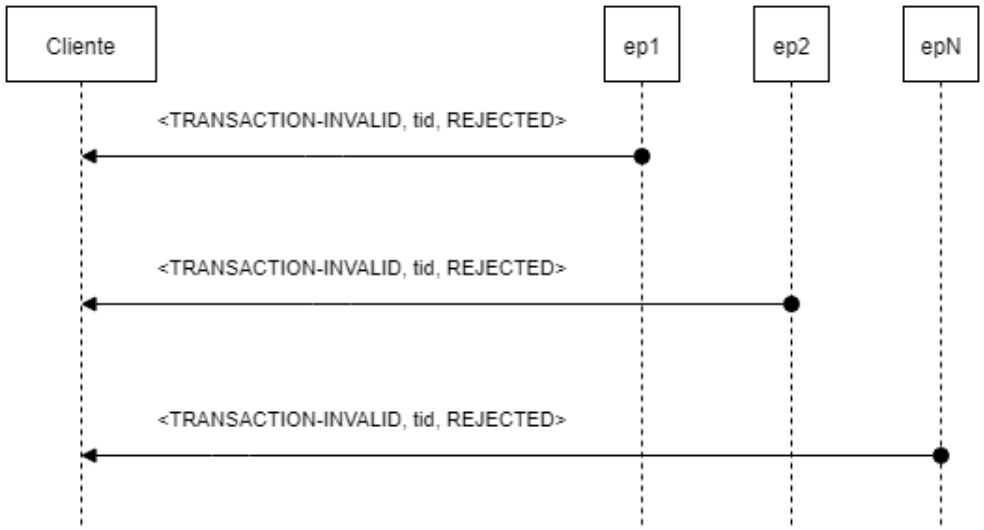

Figura 3.19: Resposta dos *peers* quando a transação é inválida.

3.5.3 Coleta de assinaturas de endosso e propagação pela rede

O cliente que propôs a transação irá coletar mensagens de endosso que tenham sido geradas nos passos anteriores. O cliente aguarda por um número de mensagens no formato $(TRANSACTION - ENDORSED, tid, *, *)$.

O número de mensagens necessárias para o endosso é especificado nas políticas de endosso instanciadas para o *chaincode*.

Quando o cliente coletar o número suficiente de mensagens dos *peers* de endosso, com a *flag TRANSACTION-ENDORSED*, o endosso da transação é estabelecida e denotada com *endorsement*.

Quando um cliente não consegue obter o número necessário de mensagens de endosso, ele pode abandonar a proposta de transação, podendo ainda tentar novamente em outro momento, ou criar outra proposta de transação.

Para as transações que receberam o *endorsement*, será invocado o *ordering service* através de *broadcast(blob)*, onde *blob* = *endorsement*. A invocação do *ordering service* pode ser feita automaticamente pelo cliente, ou por um *peer* escolhido por ele, no caso de não conseguir efetuar a invocação diretamente. A propagação do endosso ocorre como na Figura 3.20.

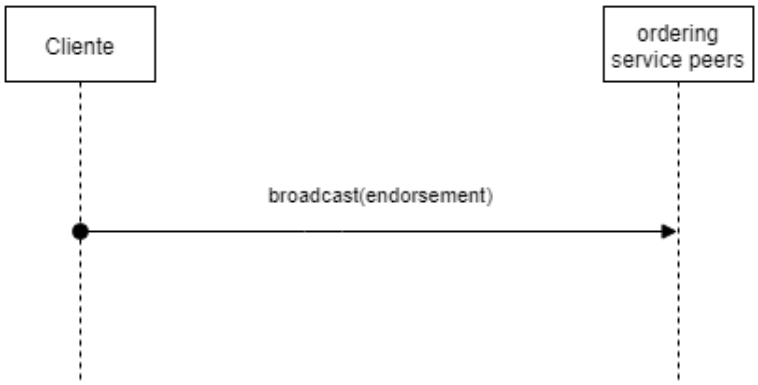

Figura 3.20: Propagação do endosso.

3.5.4 Entrega da transação aos peers

O *ordering service* entrega a transação aos *peers* através de um evento *deliver*, onde *seqno* é o número de sequência do *blob* na cadeia e *prevhash* é o *hash* do bloco anterior.

$$deliver(seqno, prevhash, blob)$$

A entrega da mensagem ocorre como ilustrado na Figura 3.21.

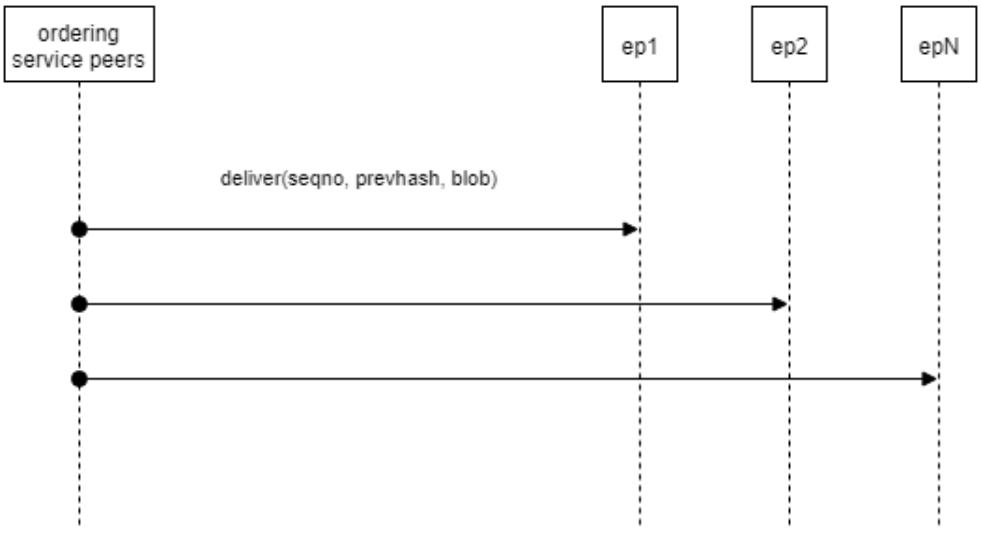

Figura 3.21: *Ordering service* reentregando as transações para os *peers*.

Cada *peer*, que já tenha realizado seus *updates* para os *seqno* anteriores, irá checar a validade do *blob.endorsement* atual em relação à política contida no *chaincode* (*blob.tran-proposal.chaincodeID*). O *peer* também verifica se as dependências da transação (*blob.endorsement.tran-proposal.readset*) não foram violadas.

A verificação de dependências pode ser implementada de diversas formas, visando a consistência ou garantia de isolamento escolhida para atualização de estados. Por padrão a garantia de isolamento é feita por serialização, mas pode ser escolhida outra forma especificando através das políticas no *chaincode*. A serialização pode ser obtida requerendo que a versão associada à cada *key* do *readset* seja igual a versão da chave no estado, e rejeitando transações que não satisfaçam esses requisitos.

Endorsing peers utilizam uma *bitmask* em sua cópia local do *ledger* para diferenciar transações válidas e inválidas.

Caso a transação tenha passado por todos esses passos, ela então é classificada como *valid* ou *commited*. O *peer* irá marcar a transação com *1* na *bitmask* do *PeerLedger*, e então aplicará o *writeset* contido em *blob.endorsement.tran-proposal.writeset* ao estado da *blockchain*.

Caso a transação falhe em passar sobre os passos de verificação do *blob.endorsement* citados anteriormente, a transação é classificada como *invalid* e o *peer* irá marcá-la com *0* na *bitmask* do *PeerLedger*, e nenhuma mudança no estado será efetuada.

A execução de todos os passos citados garante que todos os *peers* corretos terão o mesmo estado após a execução de um evento *deliver* com um determinado *seqno*. O *ordering service* garante que todos *peers* corretos irão receber exatamente a mesma sequência de eventos *deliver(seqno, prevhash, blob)*.

Como a avaliação das políticas de endosso e a avaliação da versão das dependências no *readset* são determinísticas, todos *peers* corretos irão chegar à mesma conclusão quanto a validade da transação. Então, todos *peers* aplicarão a mesma sequência de transações e atualizações de estado da mesma forma.

A Figura 3.22 ilustra o fluxo total de uma transação válida.

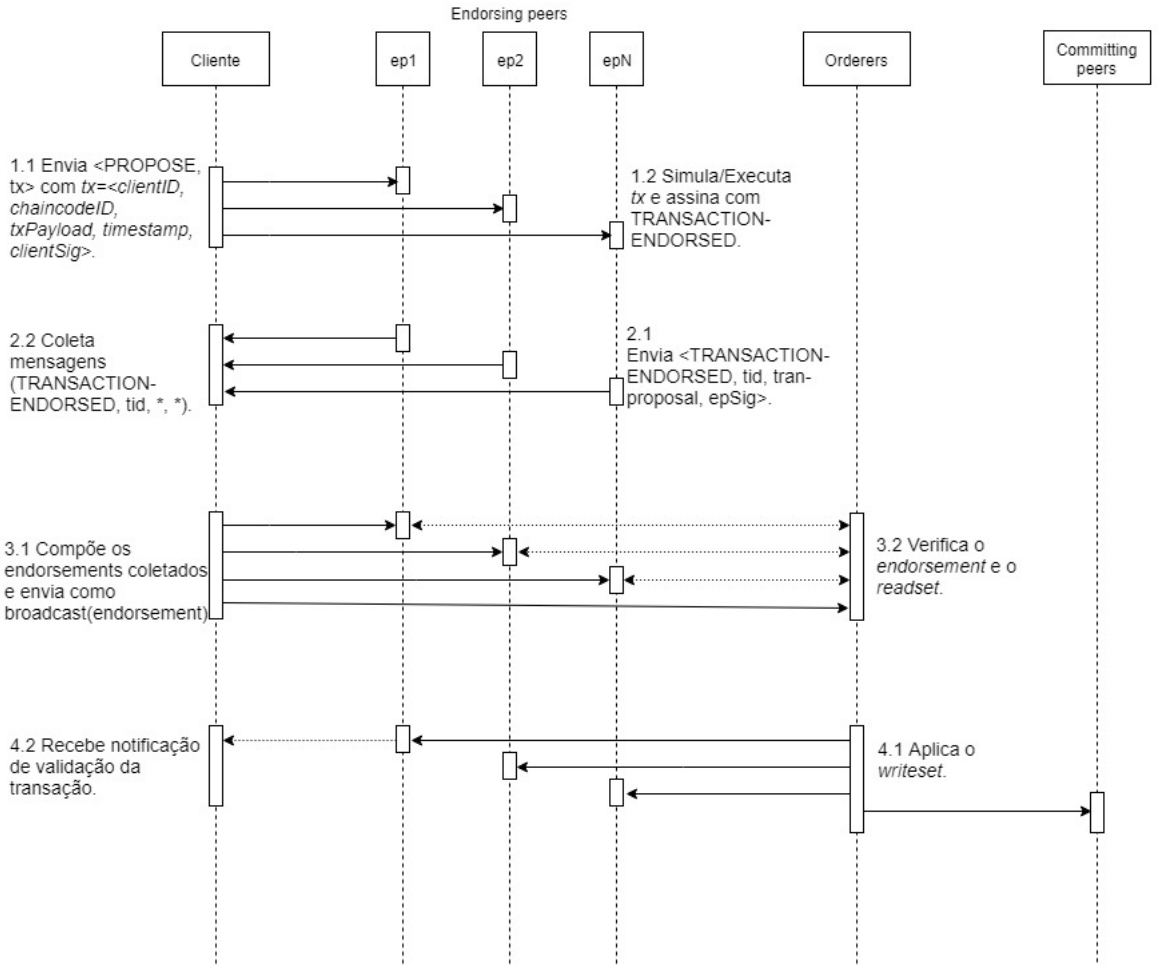

Figura 3.22: Fluxo comum de uma transação.

3.5.5 Especificação de políticas

Política de endosso é a condição sobre a qual a transação será endossada. Em uma rede *blockchain* os *peers* possuem um conjunto pré-especificado destas políticas, que foram instanciadas por uma transação *deploy* que instala um *chaincode* específico. A transação *deploy* também pode ser utilizada para especificar outros parâmetros para as políticas de endosso.

O conjunto de políticas de endosso deve ser um conjunto de políticas com um número limitado de funções, que possam garantir as propriedades de segurança como tempo de execução, determinismo, performance, entre outras garantias de segurança.

3.5.6 Avaliação das transações sob as políticas do endosso

Uma transação é considerada válida apenas se ela foi endossada de acordo com a política. Uma transação de invocação para um *chaincode* precisa primeiramente ser endossada, comprovando que satisfaça a política do *chaincode*. Caso contrário, a transação não passará ao estado de *committed*, ou seja, não será aplicada.

As políticas de endosso são predicados (como na lógica de predicados) sobre o endosso, e potenciais estados futuros, que são avaliados como verdadeiro (*TRUE*) ou falso (*FALSE*). O *endorsement* de transações *deploy* depende de políticas sobre o sistema todo, como as definidas no *chaincode* do sistema.

Os predicados em uma política de endosso podem estar relacionados à algumas variáveis, como chaves ou identidades relativas ao *chaincode*, conjuntos de *peers* de endosso, metadados adicionais do *chaincode*, elementos do *endorsement* e *endorsement.tran-proposal*, entre outros.

A avaliação de um predicado da política de endosso deve ser determinística, para que cada *peer* consiga avaliar o endosso da mesma forma conforme as políticas, sem a necessidade de interagir com outros *peers* e, ainda assim, todos *peers* corretos devem chegar ao mesmo resultado.

3.5.7 Livro de registros válidos

Diferente de outras plataformas *blockchain*, o mecanismo de registros no *Hyperledger* (chamado de *PeerLedger*) mantém registradas todas as transações, isto inclui também as transações classificadas como inválidas e que não foram aplicadas.

Para manter a abstração utilizada por outras plataformas *blockchain*, como no bitcoin, adicionou-se o *VLedger* (*Validated Ledger*) onde apenas as transações válidas e aplicadas são registradas.

O *VLedger* é uma cadeia com uso de *hash* criada a partir do *PeerLedger*, filtrando as transações inválidas de um bloco e colocando em outro, chamado de *vBlock*. Esta filtragem é realizada pelos *peers* usando a *bitmask* que foi associada no *PeerLedger*, durante a validação da transação. A filtragem é ilustrada na Figura 3.23.

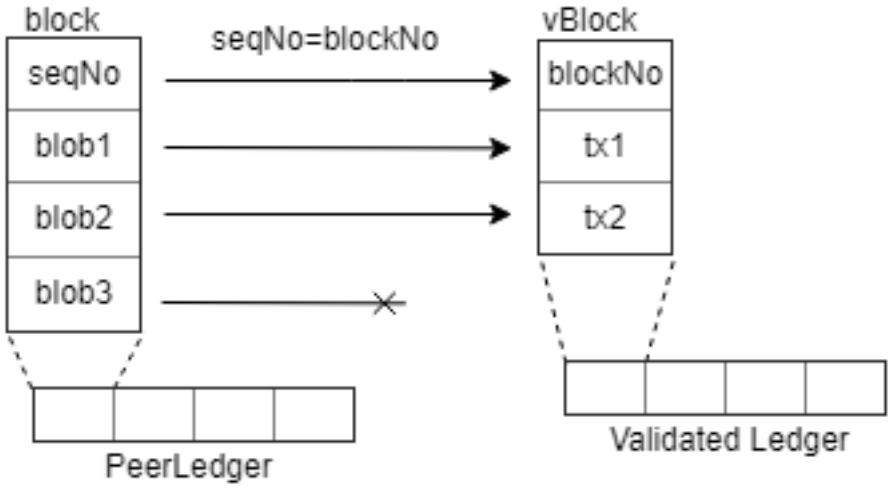

Figura 3.23: Diferença entre o *PeerLedger* e o *ValidatedLedger*.

Cada bloco do *VLedger* contém:

- O *hash* do *vBlock* anterior.
- O número do *vBlock*.
- Uma lista ordenada de todas transações válidas aplicadas pelos *peers* desde o último *vBlock* computado.
- O *hash* do bloco correspondente no *PeerLedger*, do qual o *vBlock* foi derivado.

Toda essa informação é concatenada e passa por uma função *hash* pelo *peer*, produzindo o *hash* do *vBlock* no *VLedger*.

3.5.8 Checagem do PeerLedger

O *PeerLedger* possui registros de transações inválidas que não precisam ser armazenadas para sempre, pois elas não podem efetuar mudanças de estado.

Sendo assim, é interessante para a rede que estas transações inválidas não fiquem salvas eternamente, para que se reduza o espaço de armazenamento gasto, além de facilitar o trabalho de reconstruir o estado quando novos *peers* entram na rede.

No entanto, tais transações não podem ser simplesmente descartadas, pois isso poderia gerar problemas para convencer novos *peers* sobre a validade dos blocos. Portanto, há um mecanismo

de checagem para ajudar os *peers* à aparar o *PeerLedger*, depois de estabelecer a validade dos *vBlocks* sobre a rede.

O mecanismo de checagem entra em ação à cada número determinado de blocos. Um *peer* pode iniciar uma checagem propagando uma mensagem para os outros *peers*, contendo o número do bloco atual (*blockno*) e seu *hash* (*blocknohash*), o *hash* do último bloco validado (*stateHash*) e a assinatura do *peer* (*peerSig*). A mensagem segue o formato seguinte.

$$< CHECKPOINT, blocknohash, blockno, stateHash, peerSig >$$

Os *peer* então coletam as mensagens de *CHECKPOINT*, sobre o mesmo bloco, dos outros *peers*, até ter um número suficiente de mensagens assinadas. Então, os *peers* concluem que a checagem foi válida.

Se a checagem foi válida, cada *peer* realiza os seguintes passos:

- SE $blockno > latestValidCheckpoit.blockno$ ENTÃO *latestValidCheckpoint = (blocknohash,bolckno).*
- Armazena as assinaturas dos *peers* que validaram o *CHECKPOIT* em *latestValidCheckpoitProof*.
- Armazena o *stateHash* do estado em *latestValidCheckpoitState*.
- Apara o seu *PeerLedger* até o bloco checado.

Capítulo 4

RESULTADOS E DISCUSSÕES

Para avaliar a funcionalidade dos modelos de controle de acesso desenvolvidos, um cenário alvo contendo uma implementação das funcionalidades abordadas foi criado e executado usando o sistema *Hyperledger*.

4.1 Metodologia

Os modelos de acesso desenvolvidos neste trabalho foram elaborados após a análise de trabalhos relacionados e das tecnologias apresentadas no Capítulo 2.

A implementação dos modelos foi realizada sobre a plataforma *blockchain Hyperledger Fabric v1.4* com o auxílio do conjunto de ferramentas de desenvolvimento *Hyperledger Composer v0.19*, onde os modelos foram codificados na linguagem *JavaScript.*

Através da interface gráfica do *Hyperledger Composer* também foram realizadas interações com os modelos implementados, efetuando os testes de prova de conceito com as transações previstas nos modelos.

4.2 Implementação do CAAC usando smart contracts

O diagrama de classes UML apresentado neste trabalho foi implementado conforme demonstrado na Figura 4.1, utilizando uma linguagem de modelagem chamada CTO (baseada no *framework* de modelagem *Concerto*). Para tanto, foi utilizado o *Hyperledger Compo*ser, um conjunto de ferramentas de desenvolvimento, onde o modelo foi criado em um arquivo chamado *model.cto*. Neste arquivo são definidos os *Assets*, os *Participants* e as *Transactions*.

```
asset Resource identified by resourceId {
        o String resourceId
        o String address
        o AccessPolicy accessPolicy
        --> ResourceOwner owner
}

asset Context identified by contextId {
        o String contextId
        o String role
        o String location optional
        o String proximity optional
        --> Requester owner
}

asset AccessPolicy identified by policyId {
        o String policyId
        o String role
        o String location optional
        o String proximity optional
        --> ResourceOwner owner
}

asset Access identified by accessId {
        o String accessId
        o Resource resource
        --> Requester permissionedRequester
}

abstract participant User identified by userId {
        o String userId
        o String name
}

participant ResourceOwner extends User {
        o Resource[] resources
}

participant Requester extends User {
        o Context context optional
        o Access[] accesses optional
}

transaction AddResource {
        o String resourceId
        o String address
        o AccessPolicy accessPolicy
        --> ResourceOwner owner
}

transaction ComposeContext {
        o String contextId
        o String role
        o String location optional
        o String proximity optional
        --> Requester contextOwner
}

transaction RequestAccess {
        o Resource resource
        o String accessId
        --> Requester requester
}
```

Figura 4.1: Implementação da modelagem proposta.

Os quatro *Assets* especificados na proposta deste trabalho são definidos, possuindo seus identificadores e suas variáveis relacionadas. *Resource* possui seu endereço (*address*), uma política de acesso (*AccessPolicy*) relacionada e o seu proprietário (*ResourceOwner*). *AccessPolicy*

possui variáveis para papel (*role*), localização (*location*) e proximidade (*proximity*) e a relação com o proprietário da política (*ResourceOwner*). *Context* foi definido contendo variáveis para papel (*role*), localização (*location*) e proximidade (*proximity*) de um *Requester* relacionado, proprietário do *Context*. *Access* foi definido referenciando um recurso (*Resource*) e um requerente (*Requester*) que obtém acesso ao recurso.

Quanto aos *Participants* apresentados, foram definidos um *abstract participant User* e os *participants* que o estendem, *ResourceOwner* e *Requester*. *User* define as variáveis comuns dos usuários como identificador (*userId*) e nome (*name*). Além das variáveis do *User*, um *ResouceOwner* possui um conjunto de recursos (*Resource[] resources*) para manter registro dos recursos que ele cadastrar. Já o *Requester* possui um contexto (*Context*) e um conjunto de acessos (*Access[] accesses*) para armazenar os acessos que ele obtém.

Em seguida, foram definidas as transações apresentadas no modelo. A transação *AddResource* recebe como parâmetros as variáveis do recurso a ser cadastrado, que são o identificador (*resourceId*), o endereço (*address*), a política de acesso (*AccessPolicy*) e a referência ao proprietário do recurso (*ResourceOwner*). A transação *ComposeContext* recebe como parâmetros as variáveis do contexto do *Requester* que são o identificador (*contextId*), papel (*role*), localização (*location*), proximidade (*proximity*) e a referência ao *Requester* proprietário do contexto. A transação *RequestAccess* recebe como parâmetros um identificador (*accessId*) para identificação do acesso caso ele seja permitido, a referência a um recurso e a referência ao *Requester* que deseja obter o acesso.

A lógica definida para cada transação definida nessa proposta foi especificada em um arquivo em *JavaScript* chamado *lib/logic.js* contendo a lógica de negócios (*smart contracts*), nas implementações das transações.

A Figura 4.2 mostra a implementação da transação *AddResource* como uma função *JavaScript* chamada *addResource*.

```
/**
* Add new Resource
* @param {org.caac.network.AddResource} addResource - adicionar novo recurso
* @transaction
*/
function addResource (newresource) {
        var resource = getFactory().newResource(NS, 'Resource', newresource.resourceId);
        resource.address = newresource.address;
        resource.accessPolicy = newresource.accessPolicy;
        resource.owner = newresource.owner;
        if (!resource.owner.resources) {
                resource.owner.resources = [];
        }
        resource.owner.resources.push(resource);
        return getAssetRegistry(NS + '.Resource'
                ).then( function(registry) {
                        return registry.add(resource);
                }
                ).then( function() {
                        return getParticipantRegistry(NS + '.ResourceOwner');
                }
                ).then( function(resourceOwnerRegistry) {
                        return resourceOwnerRegistry.update(newresource.owner);
                });
}
```

Figura 4.2: Código *JavaScript* para a função *AddResource*.

A função *addResource* cria um novo recurso baseado no *resourceId* com *newResource(NS, 'Resource', newresource.resourceId)*, depois adiciona as informações passadas como parâmetro (*address* e *AccessPolicy*) ao *Resource* criado, e o vincula ao *ResourceOwner* proprietário. Em seguida, utiliza um *if* para criar a lista de recursos do *ResourceOwner*, caso ela ainda não tenha sido criada. Posteriormente, faz um *push* do *resource* para a lista de recursos do proprietário. Depois disso, utiliza *promises* com o método *then()* para retornar o registro do *resource* com *registry.add(resource)* na *blockchain* e atualizar as informações do *ResourceOwner* com a chamada *resourceOwnerRegistry.update(newresource.owner)*, visto que o resource foi adicionado na sua lista.

A Figura 4.3 mostra a função *composeContext* que implementa a transação *ComposeContext*.

```
/**
* Compose context
* @param {org.caac.network.ComposeContext} composeContext - Compor o contexto
* @transaction
*/
function composeContext(newcontext){
        var context = getFactory().newResource(NS, 'Context', newcontext.contextId);
        context.role = newcontext.role;
        context.location = newcontext.location;
        context.proximity = newcontext.proximity;
        context.owner = newcontext.contextOwner;
        newcontext.contextOwner.context = context;
        return getAssetRegistry(NS + '.Context'
                ).then( function(registry) {
                        return registry.add(context);
                }
                ).then( function() {
                        return getParticipantRegistry(NS + '.Requester');
                }
                ).then( function(requesterRegistry) {
                        return requesterRegistry.update(newcontext.contextOwner);
                });
}
```

Figura 4.3: Código *JavaScript* para a função *ComposeContext*.

A função *composeContext* cria um novo *Context* baseado no *contextId* com *newResource(NS, 'Context', newcontext.contextId)* e adiciona nele as informações passadas como parâmetro, que são *role*, *location* e *proximity*, e vincula este *Context* ao *Requester* proprietário do contexto. Depois, a função atualiza a informação de contexto para o *Requester*. E então utiliza promises com o método *then()* para retornar o registro do *Context* na blockchain com *registry.add(context)* e a atualização das informações do *Requester* com *update(newcontext.contextOwner)*, visto que seu contexto foi modificado.

Na função *composeContext* podem ser adicionados mecanismos para a confirmação da autenticidade das informações de contexto fornecidas.

No *Hyperledger Fabric* há um componente chamado **MSP** (*Membership Service Provider*), que oferece a abstração dos mecanismos e protocolos de criptografia que são usados para **emissão** e **validação** de certificados e a **autenticação** do usuário. Tudo isto passa a ser abstraído em função da identidade do usuário. Dessa forma é possível trabalhar com certificados para as informações dos usuários.

No entanto, é possível trabalhar também com implementação própria de mecanismos de verificação de autenticidade das informações. Um mecanismo possível de se implementar, por exemplo, é o proposto em Choi et al. (2013) que utiliza *beacons bluetooth* para delimitar uma área (a sobreposição dos sinais dos *beacons* formam uma área de confiança), propagando suas próprias assinaturas e chaves parciais para os usuários no alcance. Dessa forma, um usuário que queira provar sua presença naquele determinado espaço, deverá compor uma assinatura agregando sua própria assinatura com as chaves e assinaturas recebidas dos *beacons*.

No *Hyperledger* é possível também trabalhar com eventos e atribuir funções para serem executadas no acontecimento do evento. Isto pode ser utilizado, por exemplo, para implementar eventos para forçar a atualização do contexto do usuário a cada intervalo de tempo. Também cabe a utilização de eventos para criar uma condição de expiração de uma permissão de acesso após um determinado tempo corrido.

```
/**
* Request access
* @param {org.caac.network.RequestAccess} requestAccess - requisição de acesso
* @transaction
*/
async function requestAccess(newrequest) {
        if (!((newrequest.requester.context.role == newrequest.resource.accessPolicy.role)
            && (newrequest.requester.context.location == newrequest.resource.accessPolicy.location)
            && (newrequest.requester.context.proximity == newrequest.resource.accessPolicy.proximity))) {
            throw new Error('Access Denied');
        }
        let accessId = newrequest.accessId;
        let resource = newrequest.resource;
        let requester = newrequest.requester;
        const access = getFactory().newResource(NS, 'Access', accessId);

        access.resource = resource;
        access.permissionedRequester = requester;
        if (!newrequest.requester.accesses) {
                newrequest.requester.accesses = [];
        }
        newrequest.requester.accesses.push(access);
        let assetRegistry = await getAssetRegistry(NS + '.Access');
        await assetRegistry.add(access);
        let participantRegistry = await getParticipantRegistry(NS + '.Requester');
        await participantRegistry.update(requester);
}
```

Figura 4.4: Código *JavaScript* para a função *RequestAccess*.

A Figura 4.4 mostra uma implementação da função *requestAccess* que representa a transação *RequestAccess* especificada no modelo apresentado neste trabalho. A função recebe como parâmetro um newrequest, que referencia as informações do requerente (*Requester*) e do recurso (*Resource*) que ele deseja acessar.

Neste caso, a função conta com um *if* para verificar se as variáveis de contexto do requerente (*role*, *location* e *proximity*) satisfazem aquelas condições definidas no *AccessPolicy* do recurso e, em caso negativo, retorna que o acesso foi negado. Qualquer lógica de controle de acesso, baseada na comprovação de requisitos, pode ser utilizada por uma aplicação, bastando

a implementação de funções como esta junto ao modelo.

Caso as condições sejam satisfeitas, a função continua sua execução criando um objeto *Access* com identificador *accessId*, com os comandos *newResource(NS, 'Access', accessId);*. Depois, insere no objeto as referências ao requerente que satisfaz a condição de acesso e ao recurso. O objeto então é adicionado à lista de acessos que o requerente possui por meio de *requester.accesses.push(access)*, e então gera o registro do *Access* para a *blockchain* com *assetRegistry.add(access)* e a atualização das informações do *Requester* com *participantRegistry.update(requester)*.

4.2.1 Execuções e resultados

A fim de testar o modelo apresentado e a sua implementação, foi criado um cenário contendo dez participantes, sendo cinco proprietários de recursos e cinco requerentes.

O intuito dos testes é a verificação das funcionalidades no modelo de controle de acesso a serviços proposto, realizando cadastros de recursos com suas respectivas políticas de acesso, atualizações de contexto e requisições de permissão de acesso.

Para cada proprietário de recurso (*ResourceOwner*) foram cadastrados cinco recursos com suas devidas informações e políticas de acesso, para avaliar o funcionamento da transação de registro de recurso (*AddResource*).

Para cada requerente (*Requester*), foi cadastrado um contexto, para avaliar o funcionamento da transação de registro de contexto (*ComposeContext*), que será posteriormente usado em requisição.

Por fim, foram feitas pelos requerentes requisições de permissão de acesso aos recursos cadastrados, para avaliar o funcionamento da transação de obtenção de permissão de acesso (*RequestAccess*).

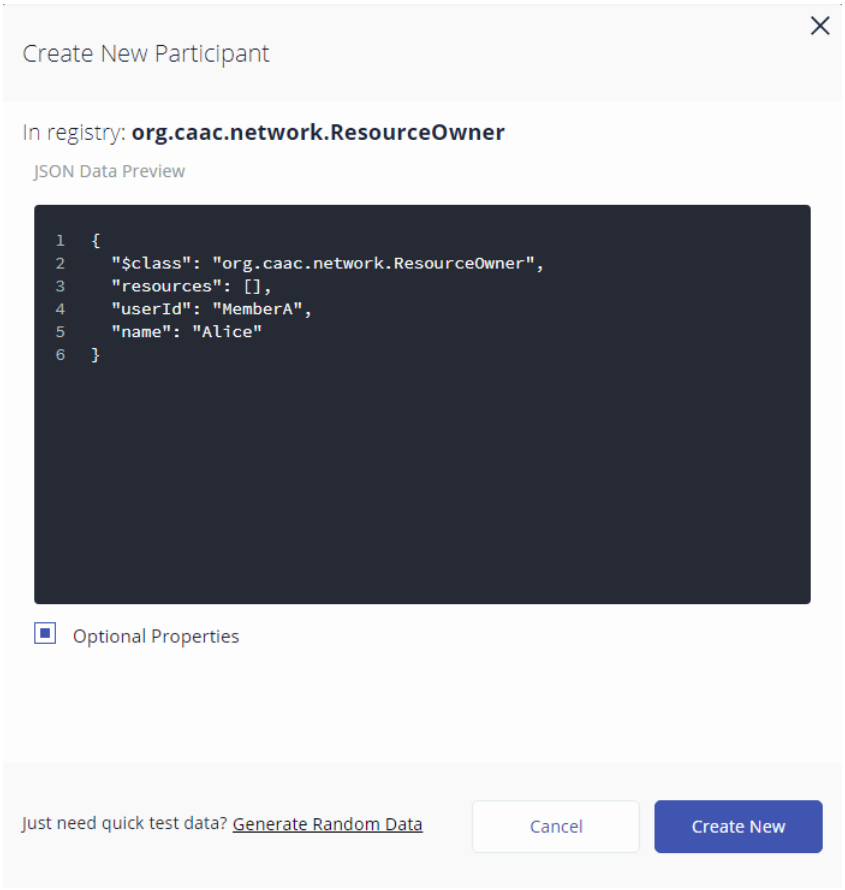

Figura 4.5: Criação de participante proprietário de recursos.

Na figura 4.5 observa-se a criação de um *ResourceOwner* para a rede. O *ResourceOwner* é criado passando um JSON com suas informações iniciais que são o nome e o identificador.

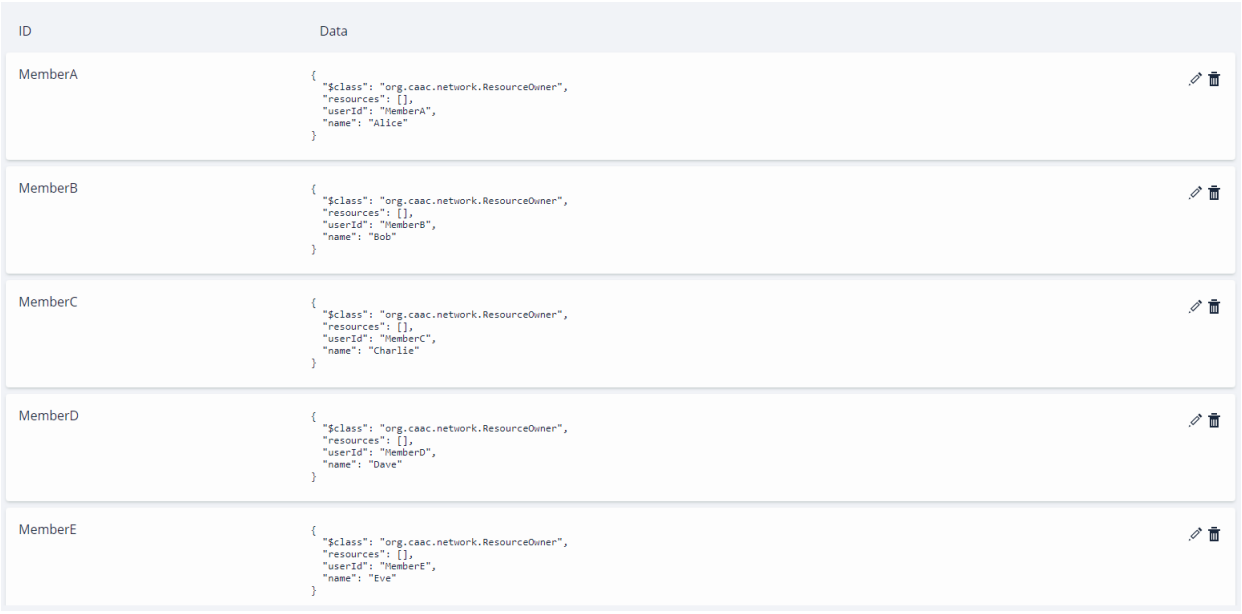

Figura 4.6: Registro dos participantes proprietários de recursos.

Foram criados cinco *ResoureOwners {Alice, Bob, Charlie, Dave, Eve}*, que receberam os identificadores de *MemberA*, *MemberB*, *MemberC*, *MemberD* e *MemberE*, respectivamente. A Figura 4.6 mostra a visualização do registro inicial destes, ainda sem recursos cadastrados.

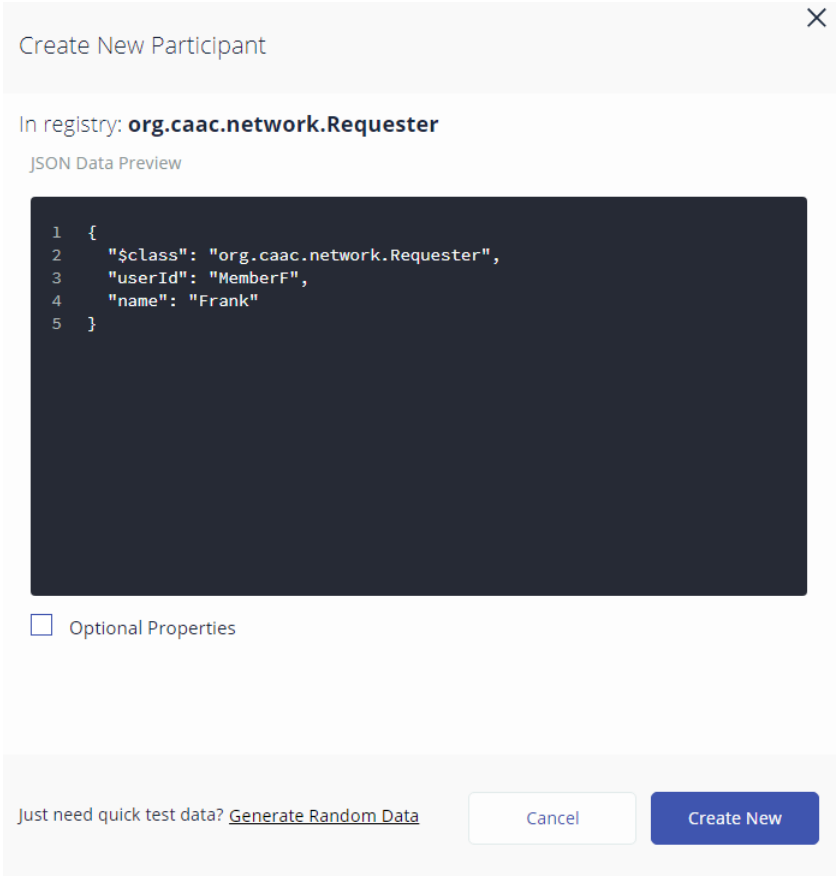

Figura 4.7: Criação de participante requerente.

A Figura 4.7 mostra o momento da criação de um *Requester*. O *Requester* é criado tendo como parâmetro a passagem de um JSON contendo o identificador e o nome.

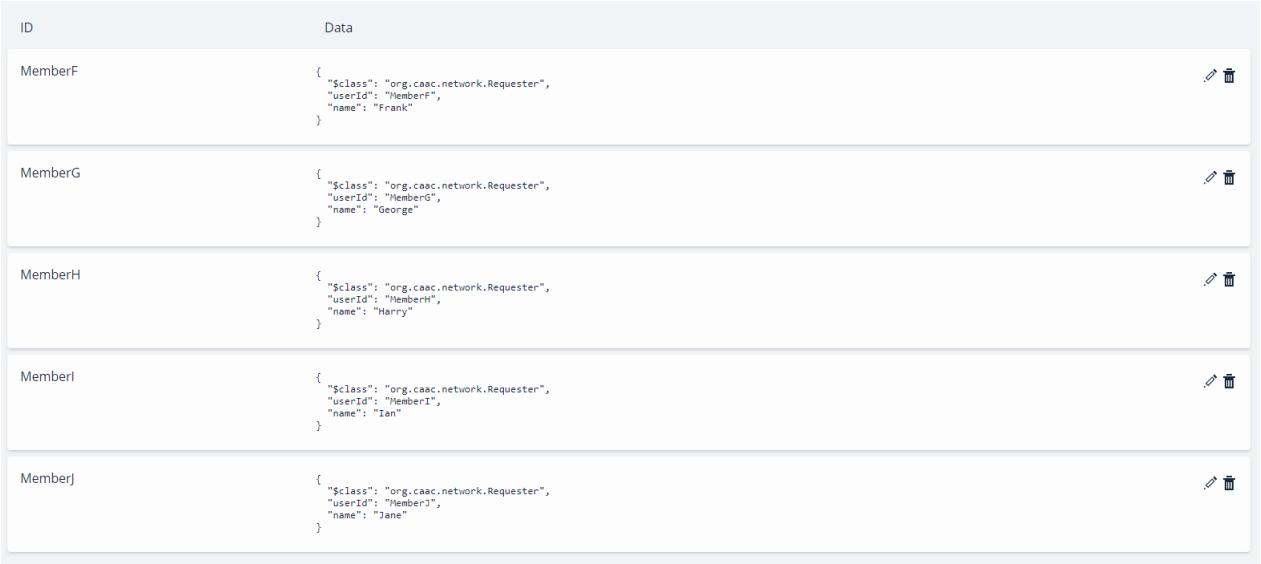

Figura 4.8: Registro de participantes requerentes.

Foram criados cinco *Requesters* para os testes, sendo eles *{Frank, George, Harry, Ian, Jane}* que receberam os identificadores *MemberF*, *MemberG*, *MemberH*, *MemberI* e *MemberJ*, respectivamente. A Figura 4.8 mostra o registro inicial dos *Requesters* criados.

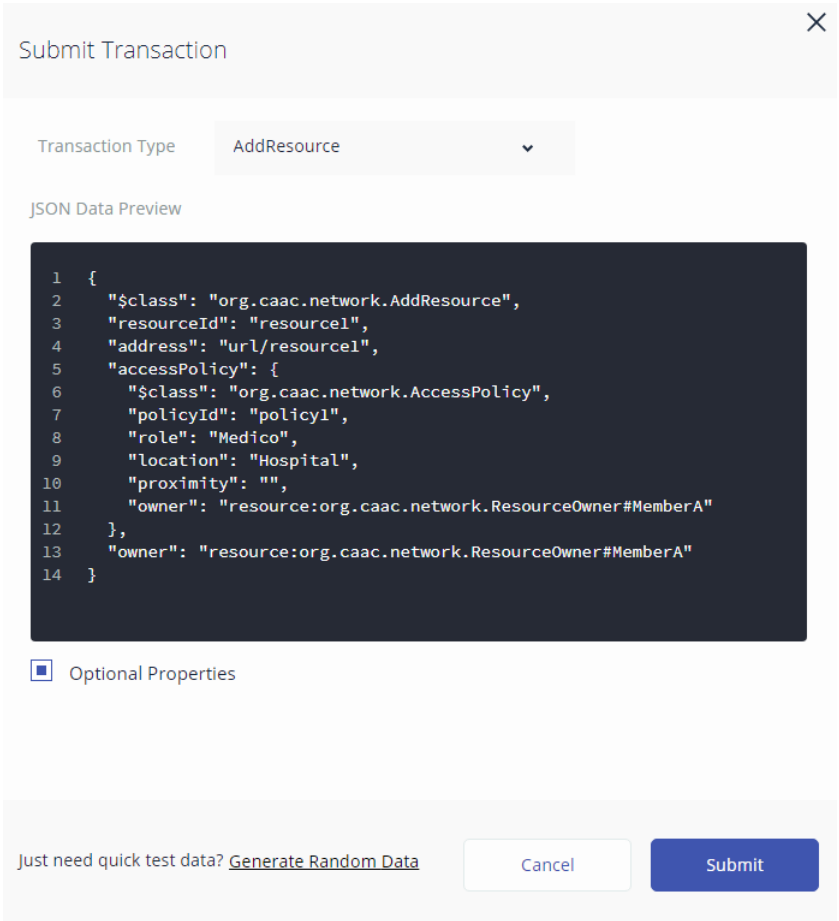

Figura 4.9: Cradastro de recurso.

A criação de um *Resource* se dá pela chamada à transação *AddResource*, passando como parâmetros os dados necessários, como o JSON representado na Figura 4.9. No JSON estão representadas as informações do recurso, como identificador e endereço, bem como as informações para a política de acesso, como papel, localização e proximidade.

Nesse JSON podemos ver também, nas linhas 11 e 13, referências ao identificador do *ResourceOwner* proprietário do recurso sendo cadastrado. Essa referência remete ao MSP, que trabalha com as abstrações dos mecanismos de criptografia. As assinaturas para uma transação são passadas pelo *Hyperledger* para a execução do chaincode, mas a abstração do MSP faz isto de forma transparente, não exigindo a codificação dessas passagens de assinaturas pelo desenvolvedor, ou usuário. Todas as informações passadas no JSON e operadas pela transação serão registradas na *blockchain*.

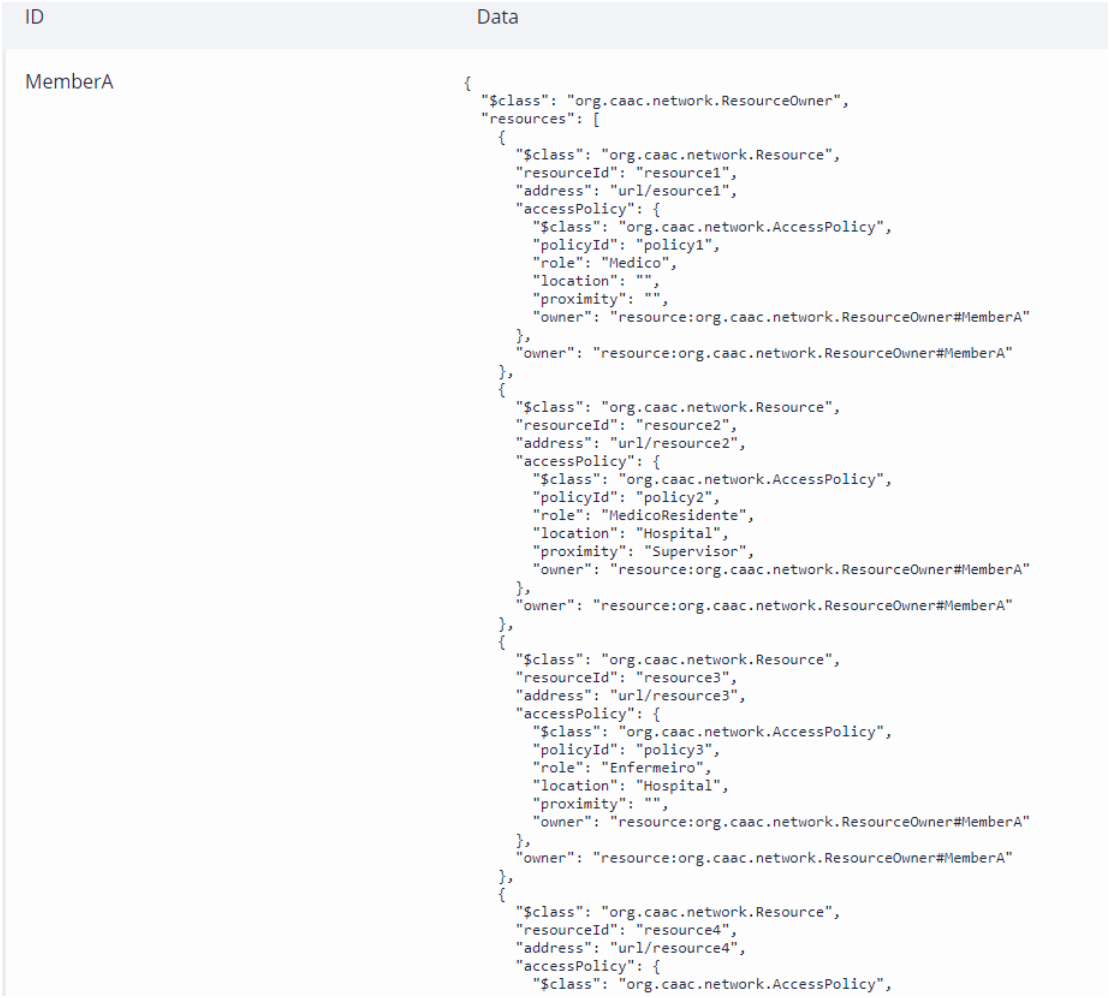

Figura 4.10: Registro de membro atualizado.

Na Figura 4.10 mostra-se um trecho das novas informações de *Alice* (*MemberA*), depois do cadastro de recursos. Nota-se que o campo resources foi atualizado, e passou a conter as

informações dos recursos adicionados com a transação *AddResource*.

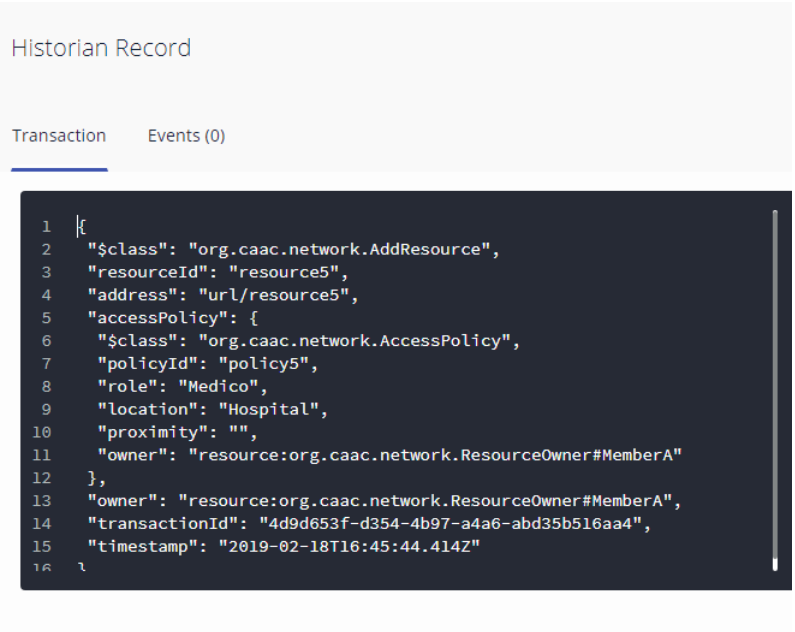

Figura 4.11: Registro da operação *AddResource* efetuada.

Na Figura 4.11 mostra-se o registro da transação que ocorreu na rede. Os registros das transações podem ser visualizados na ferramenta do *Hyperledger Composer*. No caso da Figura, nota-se que uma transação *AddResource* foi feita pelo *MemberA* para cadastrar um recurso chamado *resource5*.

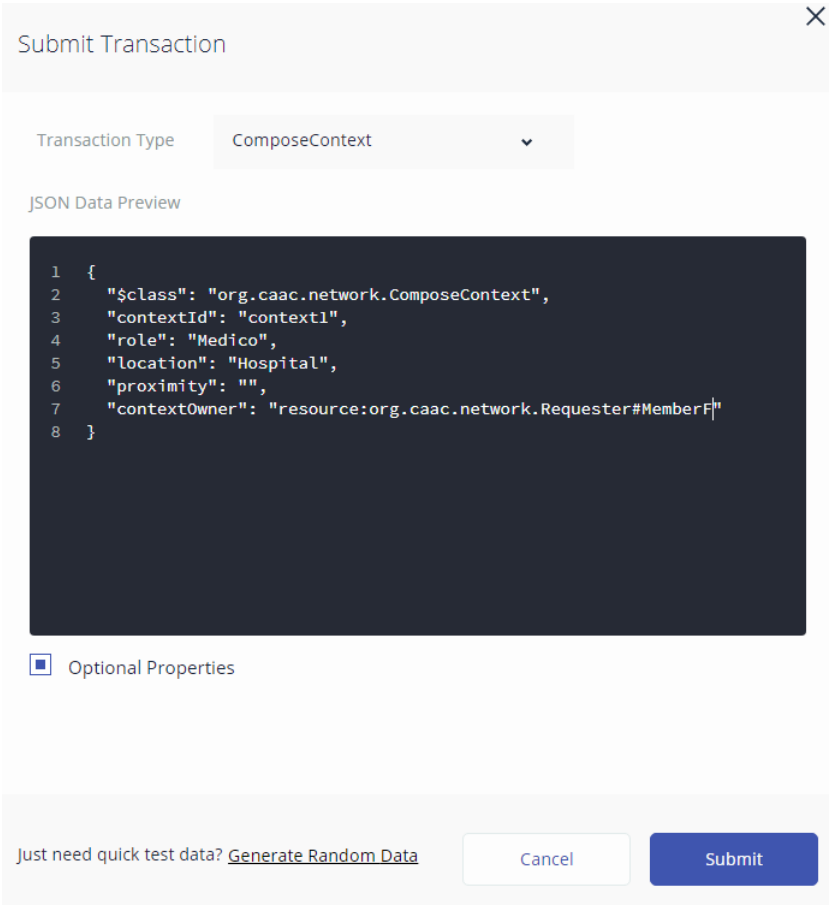

Figura 4.12: Atualização de contexto.

Na Figura 4.12, mostra-se a criação de um contexto para um *Requester*. O contexto é criado pela transação *ComposeContext* com a passagem de um JSON como na figura.

No caso da figura, foi criado para *Frank* (*MemberF*) um contexto denotando o papel de *Médico* e a localização no espaço *Hospital*. Num cenário específico, algum mecanismo de validação externo pode ser usado anteriormente à geração desse contexto.

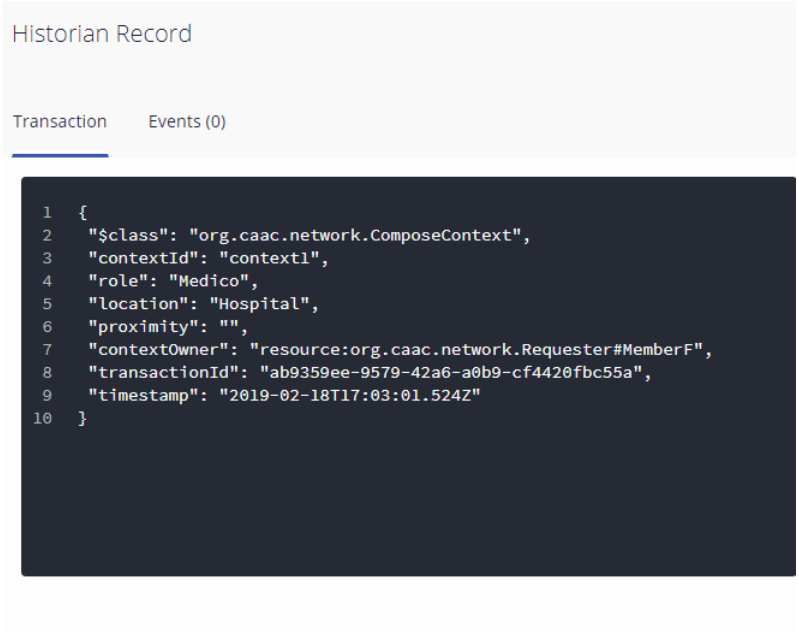

Figura 4.13: Registro da operação ComposeContext efetuada.

Na Figura 4.13 é possível observar o registro da transação realizada. O registro mostra que uma transação *ComposeContext* foi realizada, cadastrando um contexto identificado por *context1* pelo *MemberF*.

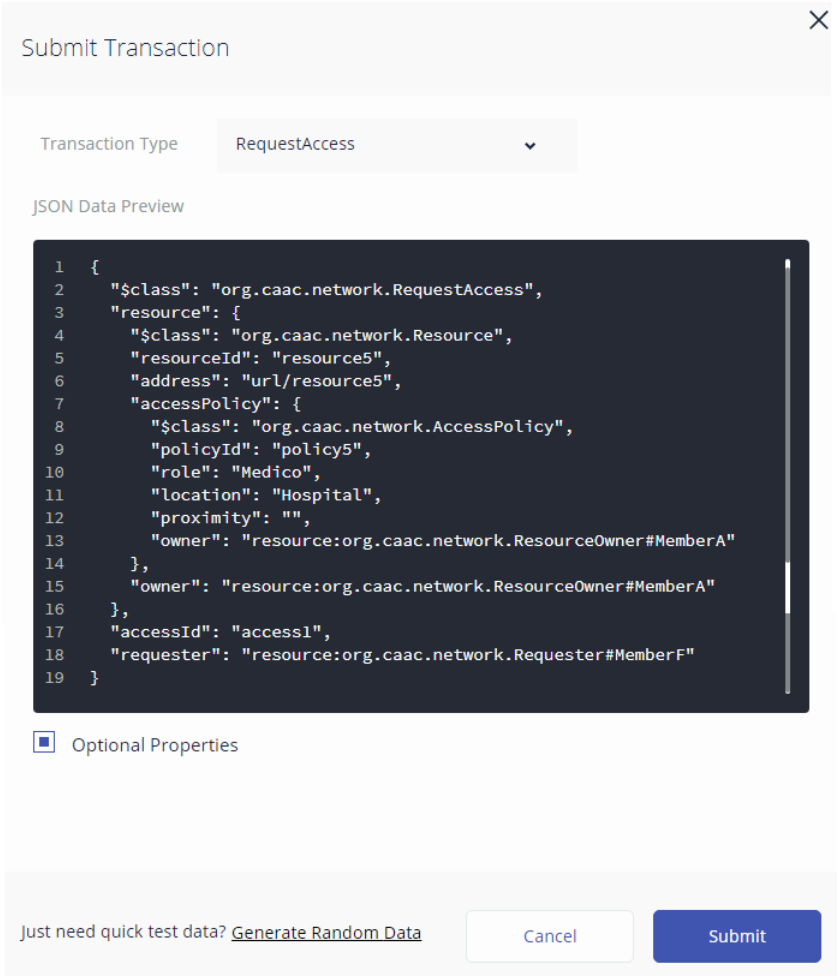

Figura 4.14: Requisição para acesso.

Na Figura 4.14, mostra-se a requisição de permissão de acesso com uma transação *RequestAccess*. A requisição é feita com a passagem de um JSON com as informações necessárias para a transação.

A imagem mostra *Frank* (*MemberF*) requisitando acesso ao *resource5* de *Alice* (*MemberA*). Como *Frank* possui um contexto que satisfaz a política de acesso do *resource5*, as informações exibidas na Figura 4.15 serão registradas.

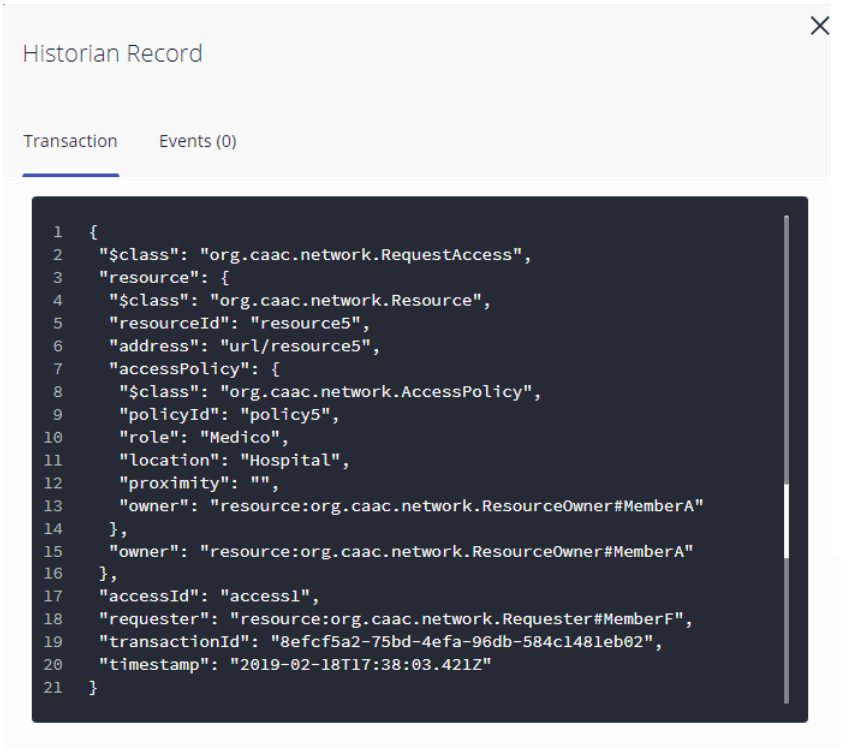

Figura 4.15: Registro da operação RequestAcess efetuada.

A Figurra 4.15 mostra o registro que ocorreu com a realização da transação *RequestAcces* pelo *MemberF* ao *resource5* do *MemberA*.

Tendo ocorido as etapas anteriores com a criação do *Access* vinculando requerente com recurso, basta a utilização de uma função para buscar a informação desejada do recurso.

4.3 Implementação do CAAC com smart contracts baseado em transferências de recursos

Considerando os mecanismos de verificação de assinaturas dos atores em transações no *Hyperledger*, e possivelmente em qualquer infraestrutura equivalente para uso das estratégias de *blockchain*, esse trabalho apresenta ainda um outro modelo de controle de acesso, baseado exclusivamente nas invocações de serviço.

Para tanto, propõe-se a associação de permissões com *Assets*, criados pelos proprietários de recursos compartilhados e requisitados nas invocações dos serviços desejados. Diferentes (infinitos) *assets* podem ser criados e gerenciados por uma infraestrutura de *blockchain*, assim como há mecanismos para transferência desses *assets* de forma validada e auditável entre os

participantes.

Nesse sentido, apresenta-se a ideia de um sistema com *chaincodes* adicionados em tempo de execução e *Assets* utilizado como "moeda de troca" a ser gasta diretamente na obtenção de uma informação (ou invocação de qualquer serviço / *chaincode*).

Neste cenário, um usuário recebe *Asset(s)* que prova(m) que ele está em um contexto, ou satisfaz requisito(s) de contexto e, portanto, tem as permissões necessárias para a execução do serviço. Diferentes mecanismos, inclusive externos, podem ser utilizados para a atribuição desses *Assets* de forma confiável. Para solicitar a execução de um serviço, um usuário gasta esse *Asset*, transferindo-o ao prestador do serviço desejado. Assim, se o *Asset* corresponde ao contexto necessário para o acesso, a transação de requisição requer que a posse desse *Asset* seja transferida para o provedor do serviço desejado e o registro dessa transferência na *blockchain* seja fornecido como parâmetro da chamada.

Esta alternativa visa evitar buscas constantes por permissões salvas e o reúso indevido destas permissões.

Nesta abordagem, o proprietário de um recurso implementa um *chaincode* especificando como as informações do seu recurso serão acessadas, como o contexto do requerente será avaliado para concessão do *Asset* necessário para o acesso e como esse *Asset* será gasto na requisição do acesso. Todos esses requisitos, validáveis pelos mecanismos de transferências de *Assets*, passam a ser embutidos numa avaliação lógica executada no preâmbulo de um *chaincode*.

Resumidamente, antes de executar a lógica associada a um serviço, é preciso que o código de um *chaincode* inclua a verificação da transferência dos *Assets* necessários, do solicitante para o dono do serviço.

Para solicitar um serviço, um cliente deve transferir os *Assets* necessários para o dono do serviço desejado. Para tanto, utiliza sua chave privada, em transação existente no *Hyperledger*, para transferir os *Assets* necessários. Essa transferência indica que o cliente do serviço satisfaz as condições necessárias, ou seja, tem as permissões requeridas.

Os identificadores das transferências realizadas, e devidamente salvas nas cadeias de informação do *Hyperledger*, são passados pelo cliente na solicitação de execução do serviço desejado. Retomando as operações do servidor, vê-se que ele será capaz de validar que os *Assets* foram transferidos e pode prestar o serviço.

As comunicações realizadas entre as partes neste modelo pode ser visualizada na Figura 4.16.

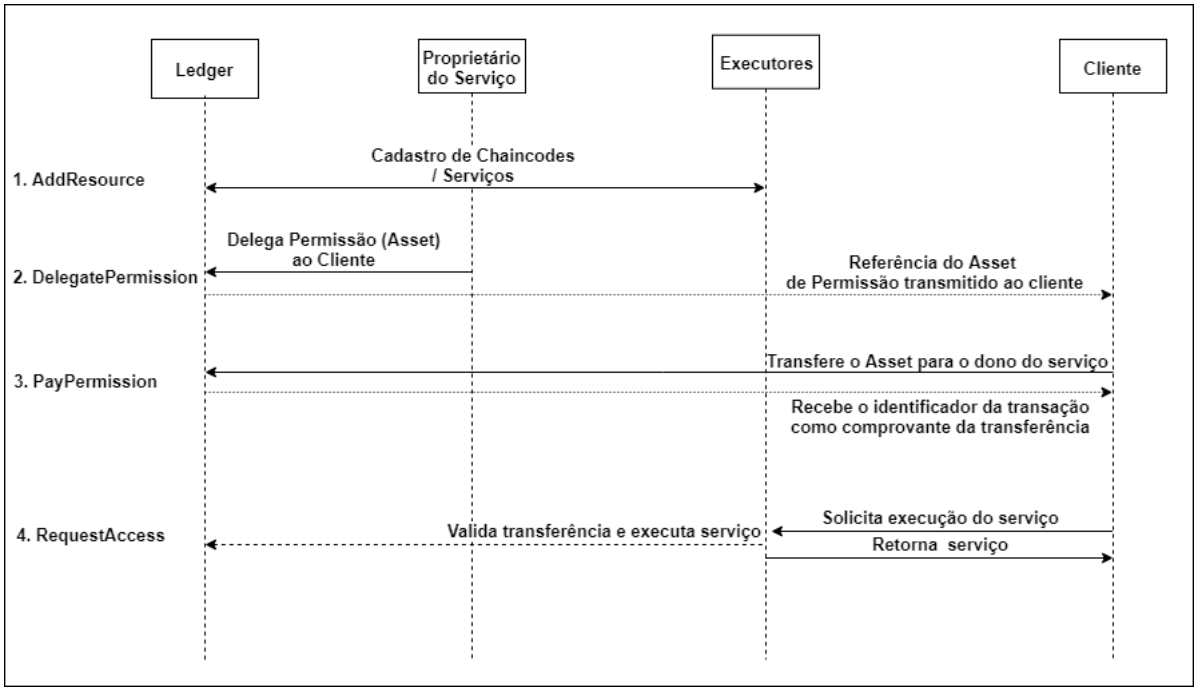

Figura 4.16: Fluxo de comunicações proposto.

Vale observar que os códigos executados pelo *Hyperledger* são todos implementados como *chaincodes*. Assim, há os chaincodes de serviço, associados às operações de transferências e criações de *Assets*, por exemplo, disponíveis na biblioteca / API do *Hyperledger*, ou criados pela aplicação que vai usar essa infraestrutura. Usando os serviços do *Hyperledger* também é possível criar dinamicamente novos *chaincodes*, que serão os serviços providos por nós participantes do sistema e cujas execuções serão solicitadas pelos clientes que satisfizerem os requisitos de permissão.

Para demonstrar a implementação dos *chaincodes* que viabilizam o modelo, foram criadas funções em *JavaScript* utilizando a ferramenta *Hyperledger Composer* que determinam o funcionamento das etapas de comunicação.

```
/**
* Add new Resource
* @param {org.caac.network.AddResource} newresource - adicionar novo recurso
* @transaction
*/
function addResource(newresource) {
        var resource = getFactory().newResource(NS, 'Resource', newresource.resourceId);
        resource.address = newresource.address;

        resource.owner = newresource.owner;
        if (!resource.owner.resources) {
        resource.owner.resources = [];
   }
        resource.owner.resources.push(resource);
  return getAssetRegistry(NS + '.Resource'
  ).then( function(registry) {
        return registry.add(resource);
  }
  ).then( function() {
                return getParticipantRegistry(NS + '.ResourceOwner');
  }
  ).then( function(resourceOwnerRegistry) {
        return resourceOwnerRegistry.update(newresource.owner);
  });
}
```

Figura 4.17: Função *JavaScript* para cadastro de recurso.

Vale ressaltar que um recurso ou serviço pode ser cadastrado diretamente por meio de um *chaincode* implementado pelo provedor, especificando seus requisitos e comportamento. No entanto, uma transação *AddResource* foi implementada para ilustrar um cadastro de serviço.

A Figura 4.17 mostra a implementação da função *addResource*. Nesta função realiza-se o registro de um recurso/serviço para ser acessado com os parâmetros passados, criando o registro a partir do identificador (*resourceId*) com o comando *newResource(NS, 'Resource', newresource.resourceId)*, e então adiciona as outras informações de endereço (*address*) e proprietário (*owner*). Posteriormente, a função ordena a gravação das informações do serviço/recurso e a atualização das informações do usuário proprietário na *blockchain*.

```
/**
 * Create a permission
 * @param {org.caac.network.DelegatePermission} newPermission - Permissão
 * @transaction
 */
async function delegatePermission(newPermission) {
        const permission = getFactory().newResource(NS, 'Permission', newPermission.permissionId);
        permission.resourceId = newPermission.resourceId;
        let participantRegistry = await getParticipantRegistry(NS + '.Requester');
        permission.permissionedRequester = await participantRegistry.get(newPermission.requesterId);
        permission.owner = await participantRegistry.get(newPermission.requesterId);
        let assetRegistry = await getAssetRegistry(NS + '.Permission');
        await assetRegistry.add(permission);
}
```

Figura 4.18: Função *JavaScript* para criação de *Assets* de permissão.

A criação de permissões pode ocorrer tanto pela delegação direta, quanto pela comprovação de condições de contexto junto à mecanismos validadores.

A Figura 4.18 mostra a implementação da função *delegatePermission*. Essa função realiza a criação de um *Asset* de permissão para um identificador passado como referência na chamada da função. Adiciona-se neste *Asset* o identificador do recurso/serviço para qual a permissão será válida, e o identificador do usuário que receberá a permissão. Posteriormente, ordena a atualização das informações do usuário que recebe a permissão, e o registro do *Asset*.

```
/**
* Pagar asset de permissão para o provedor
* @param {org.caac.network.PayPermission} pay - pagamento a ser processado
* @transaction
*/
async function payPermission(pay) {
        pay.permission.owner = pay.provider;
        let assetRegistry = await getAssetRegistry(NS + '.Permission');
        await assetRegistry.update(pay.permission);
        pay.permission.permissionedRequester.recibo = pay.transactionId;
        let participantRegistry = await getParticipantRegistry(NS + '.Requester');
        await participantRegistry.update(pay.permission.permissionedRequester);
}
```

Figura 4.19: Função *JavaScript* para transferência de *Assets* de permissão.

Na Figura 4.19 mostra-se a implementação da função *payPermission*, onde um usuário que quer realizar um acesso, deve transferir sua permissão para o dono/gerenciador do serviço. A função altera o proprietário do *Asset* de permissão e promove a atualização do registro do *Asset*. Depois registra o identificador da transação para o usuário que realizou a transferência como um comprovante (recibo) da operação.

```
/**
* Request access
* @param {org.caac.network.RequestAccess} newrequest - requisição de acesso
* @returns {org.caac.network.ServiceJson}
* @transaction
*/
async function requestAccess(newrequest) {

  let perRegistry = await getAssetRegistry(NS + '.Permission');
  let permission = await perRegistry.get(newrequest.permissionId);

  let resRegistry = await getAssetRegistry(NS + '.Resource');
  let resource = await resRegistry.get(newrequest.resourceId);

  let tranx = await query('selectPayment', {recibo : newrequest.requester.recibo,
                                  resourceId : newrequest.resourceId,
                                  userId : newrequest.requester.userId});

  if(!(tranx.length > -1)) {
    throw new Error('Falha na confirmação do pagamento');
  }

  const response = getFactory().newConcept(NS, 'ServiceJson');
  response.servicejson = 'Service';
  return response;
}
```

Figura 4.20: Função *JavaScript* para obtenção de acesso.

A Figura 4.20 mostra a função *requestAccess* pela qual um usuário, que já transferiu o *Asset* de permissão, realiza o pedido de acesso ao serviço. O usuário passa como referência para a função, o seu identificador e o identificador da transação de transferência previamente realizada, além do identificador do recurso/ serviço que quer acessar.

A função realiza a verificação se o *Asset* de permissão foi transferido corretamente, através de uma busca pela existência da transação com o identificador informado, e se a permissão transferida realmente era relacionada ao usuário requerente e ao serviço especificados.

Caso a verificação não encontre a transferência correta, com os dados especificados, a função retorna um aviso de falha na verificação. Caso a verificação seja bem sucedida, a função retorna um JSON com a resposta do serviço.

4.3.1 Execuções e resultados

Para testar o modelo apresentado e sua implementação, foi criado um cenário para teste. O cenário conta com vários nós clientes na rede e um provedor P de serviço s.

O provedor Ps registra *chaincode* na rede e é também o gerente de *Assets* de permissão As.

O provedor *Ps* cria Asset *As* e atribui aos clientes que têm permissão de acessar esse serviço. Também é possível que o *Asset* necessário para autorizar a execução de um serviço vá ser provido por um outro elemento. Por exemplo, por algum nó que garante que uma condição de contexto é satisfeita por um cliente.

A atribuição de *Asset* pode ser algo dinâmica, por exemplo, para comprovar um dado de contexto, como um *beacon* que espera requisição de 3 clientes diferentes dentro de um intervalo de tempo para prover *Asset* que indica que 3 pessoas necessárias estão num mesmo local num dado momento. Ou, então, um cliente precisa colher dados de 3 beacons e repassar a um serviço dentro de um intervalo de tempo para provar que ele está num local num dado momento.

Pode haver vários requisitos para a execução de um serviço; ou seja, mais de um *Assets* necessários para provar a permissão.

Um cliente gera transferência do(s) *Asset(s)* para o Provedor. Aqui, vale observar que a transferência do(s) *Asset(s)* deve(m) ser feita(s) para uma entidade específica, como se fosse o "caixa" de um serviço, não necessariamente o provedor. É o registro dessa transferência no *ledger* que será analisado por cada um dos *endorsing peers* para verificar que o cliente tem permissão para que o serviço seja executado. Para evitar reúso das autorizações, a verificação feita no preâmbulo de controle de acesso do *chaincode* executado pelos *endorsing peers* leva em consideração a recência da transferência. Ou então, pode-se salvar uma referência de transferências já usadas no próprio serviço.

Na invocação de um serviço desejado, um cliente deve prover referências dos registros no *ledger* que indicam que ele fez as transferências necessárias ao provedor do serviço. Isso pode ser passado dentro de um JSON usado na invocação do serviço. A existência dessas referências vai facilitar a comprovação da permissão pelo provedor do serviço, que ele não precisa vasculhar o *ledger* em busca dessa informação.

No retorno de um serviço provido, um JSON pode também ser retornado ao cliente, contendo quaisquer dados relevantes.

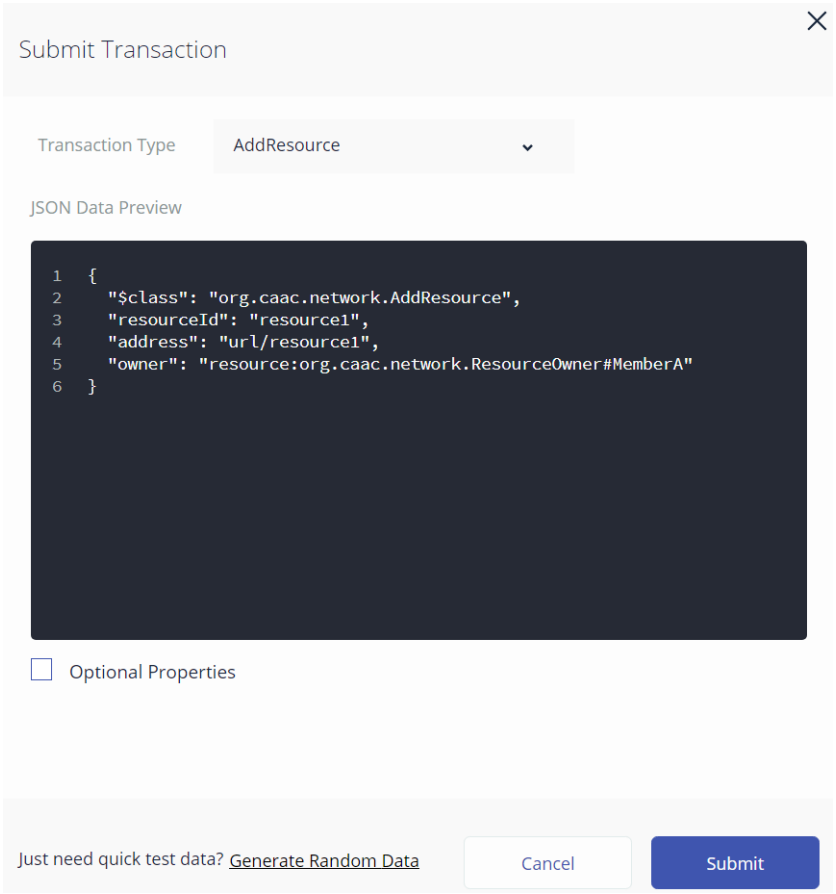

Figura 4.21: Cadastro de recurso/serviço.

Na Figura 4.21, utiliza-se a operação de cadastrar um novo recurso/serviço. A operação é realizada por um usuário que vá cadastrar o serviço e utiliza a transação *AddResource*, passando um JSON com um identificador, um endereçamento de referência do serviço e o proprietário do serviço.

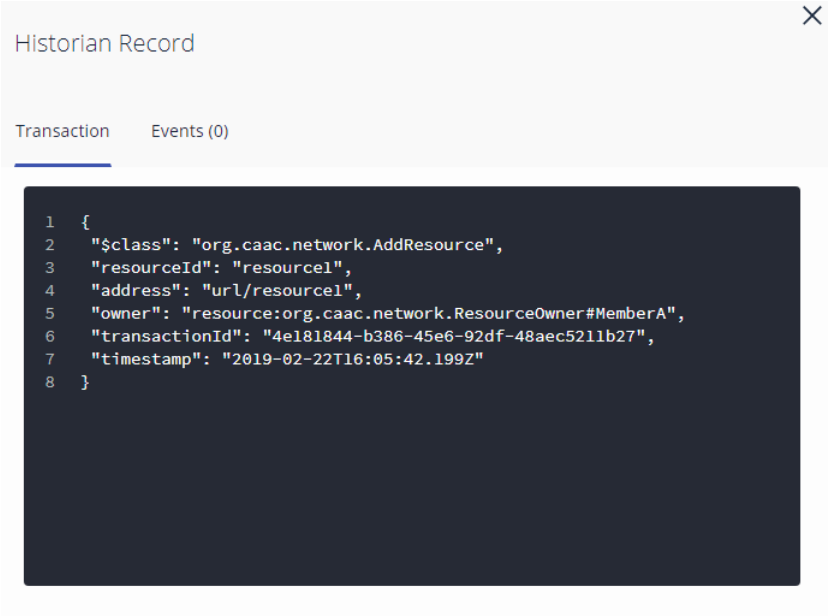

Figura 4.22: Registro do cadastro de recurso/serviço efetuado.

Na Figura 4.22, mostra-se o registro da transação que foi feita para cadastrar o recurso/serviço. Observa-se que foi cadastrado um recurso com identificador *resource1*, com o referenciamento de um endereço e a referência ao proprietário do recurso *MemberA*.

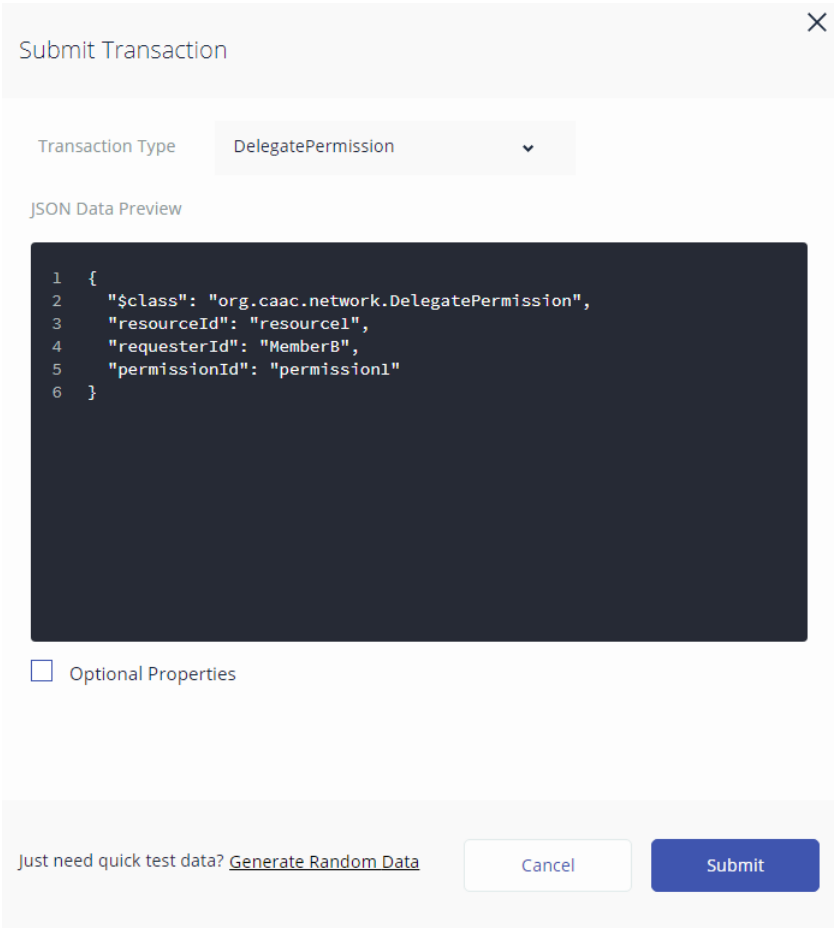

Figura 4.23: Criação de *Asset* para permissão.

Na figura 4.23, realiza-se a operação de emissão de permissão. A operação ocorre pela chamada da transação *DelegatePermission*, pelo proprietário do serviço, ou um validador, passando um JSON com o identificador do recurso para o qual a permissão será valida, o identificador do usuário para o qual a permissão será concedida e um identificador para a permissão.

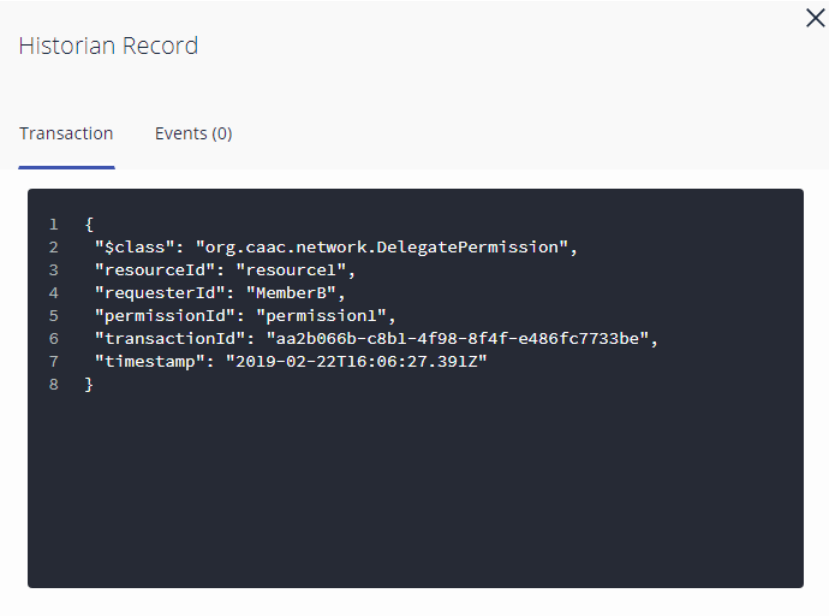

Figura 4.24: Registro da criação de Asset efetuada.

A Figura 4.24 mostra o registro gerado após a transação *DelegatePermission*. Observa-se que foi criado um *Asset* de pemissão *permission1*, para acesso ao serviço *resource1*, para o usuário *MemberB*.

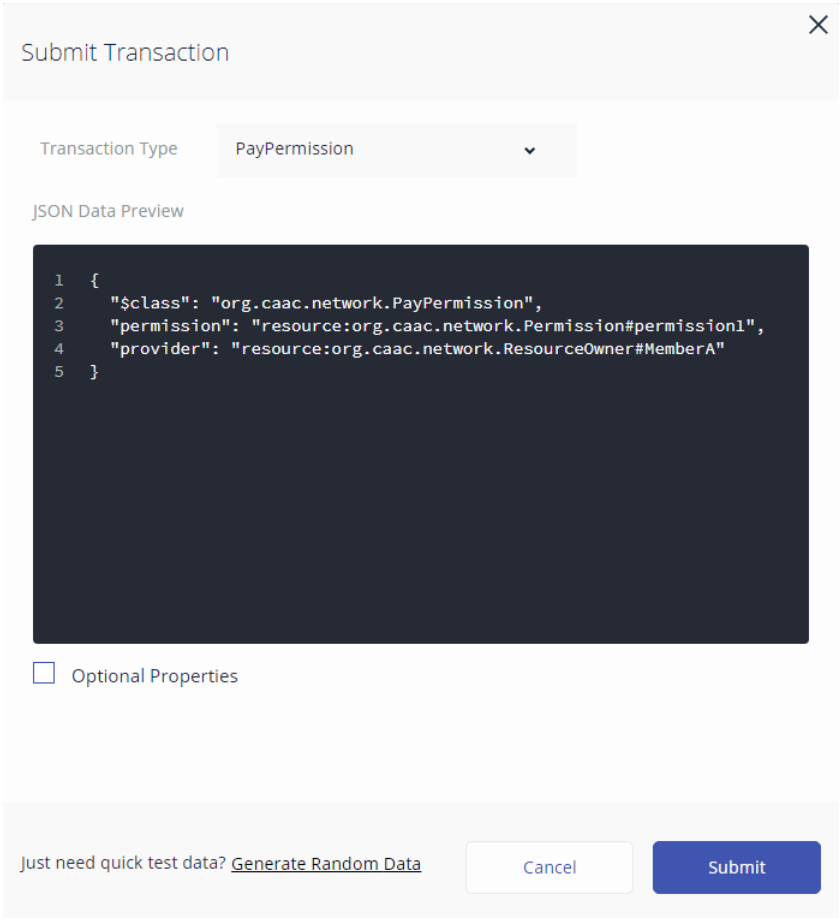

Figura 4.25: Transferência de *Asset* de permissão.

Na figura 4.25, mostra-se a operação para o pagamento do *Asset* de permissão, para obter o acesso. A operação ocorre quando um usuário que quer obter o acesso utiliza a transação *PayPermission* para pagar pelo acesso, passando um JSON com identificador da permissão e o identificador do proprietário/provedor do serviço.

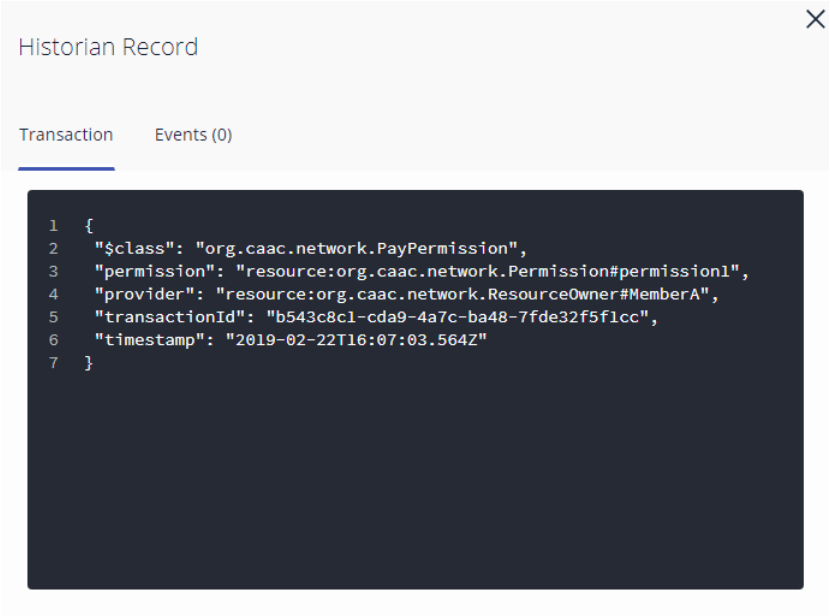

Figura 4.26: Registro da transferência efetuada.

A Figura 4.26 mostra o registro do pagamento efetuado na transação *PayPermission*. O registro mostra que a permissão permission1 foi paga ao provedor *MemberA*. O campo *transactionId* (linha 5) se refere ao identificador da transação, que é utilizado para comprovar o pagamento do *Asset*.

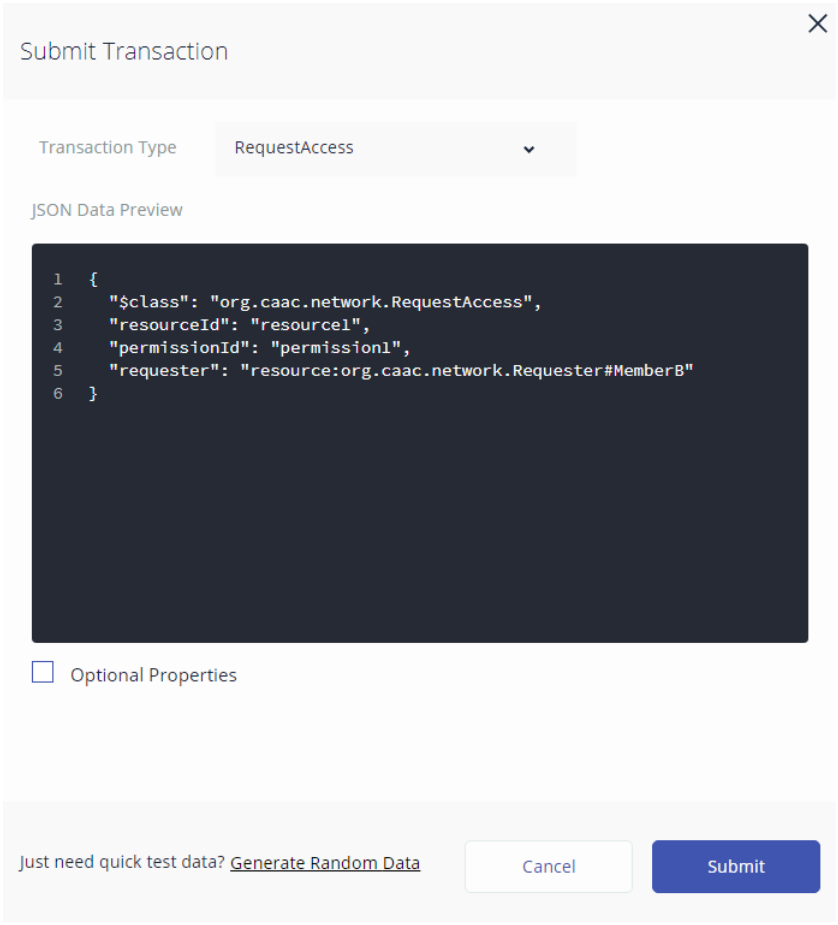

Figura 4.27: Requisição de acesso.

Na Figura 4.27, ocorre a operação de requisição de acesso. A operação ocorre com o usuário requerente de acesso fazendo a chamada da transação *RequestAccess* passando um JSON com o identificador do recurso a ser acessado, o identificador do *Asset* de permissão que ele pagou pelo acesso, e o seu próprio identificador, pelo qual será referenciada a transação anterior de transferência do *Asset* de permissão, verificando se foi pago corretamente.

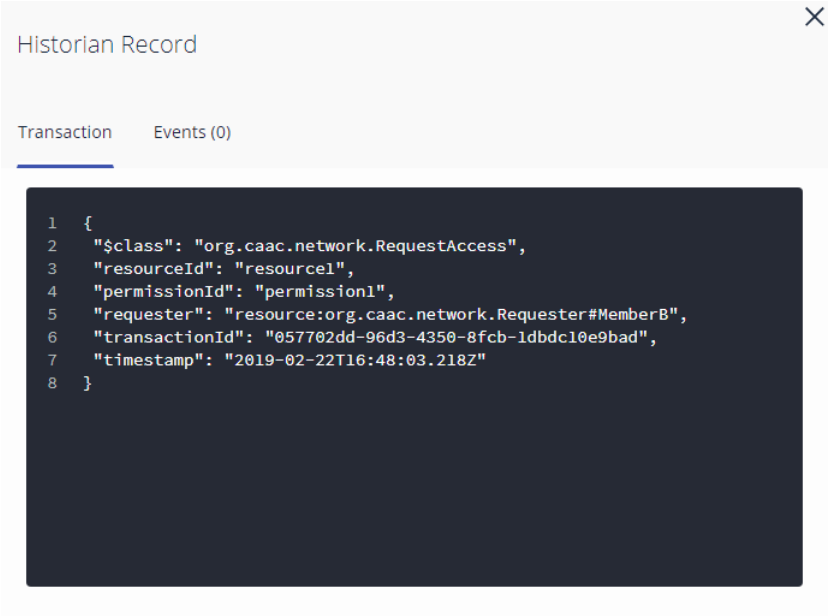

Figura 4.28: Registro da requisição efetuada.

A Figura 4.28, mostra o registro ocorrido após a transação *RequestAccess*. Verifica-se que o acesso ao serviço *resource1* foi efetuado para o *MemberB*, após ter pago a permisssão *permission1*.

4.4 Aspectos operacionais das propostas usando CAAC

Na operação dos dois modelos de controle de acesso propostos neste trabalho, o seguinte cenário é previsto:

- Há um conjunto de nós interligados em rede, disponíveis para execução das atividades de manutenção da *blockchain* (neste caso, do *Hyperledger*): dois ou mais *endorsing peers* para execução do serviço e ao menos um *ordering server*.
- Um nó que deseja participar da rede provendo serviço deve registrar-se na rede, criando uma identidade e registrando seus serviços em algum *channel*.
- Um cliente que deseja usar qualquer serviço provido na rede também deve criar uma identidade e conectar-se a um *channel* do *Hyperledger*.

Para o modelo de *CAAC usando smart contracts*, as operações de clientes e servidores são as seguintes:

- Ao cadastrar um recurso, um servidor deve cadastrar uma política de acesso vinculada, que deverá ser cumprida para geração da permissão.
- Um cliente que deseja executar um serviço deve cadastrar seu contexto e solicitar a permissão. Isso é feito através do serviço *RequestAccess*, que verifica se o contexto do cliente satisfaz a política de acesso do serviço, e gera um registro de permissão. Esse registro é verificado para a realização do acesso.

Para o modelo *CAAC com smart contracts baseado em transferências de recursos*, as operações de clientes e servidores são as seguintes:

- Na fase inicial de qualquer serviço provido deve existir uma verificação de requisitos que correspondem a transferência(s) de *Asset(s)*.
- Um cliente que deseja o serviço deve satisfazer os critérios que levam ao recebimentos dos *Assets* necessários. Assim, solicitar permissão significa obter os *Assets* apropriados.
- Para solicitar um serviço, um cliente transfere os *Assets* associados aos privilégios necessários para o dono do serviço, ou entidade responsável.

Para viabilizar esses modos de operação previstos, serviços específicos (*chaincodes*) foram desenvolvidos na plataforma *Hyperledger*.

No modelo CAAC com smart contracts:

- *addResource*: cadastro um novo recurso e uma política de acesso associada a ele.
- *composeContext*: cadastra as informações de contexto do usuário.
- *requestAccess*: verifica se o contexto do usuário satisfaz a política de acesso do recurso e gera o registro da permissão.

No modelo *CAAC com smart contracts* baseado em transferências de recursos:

- *addResource*: cadastra um novo serviço.
- *delegatePermission*: gera *Asset* de permissão para um cliente.
- *payPermission*: cliente transfere *Asset* de permissão para o proprietário ou gestor do serviço.
- *requestAccess*: verifica se as permissões foram corretamente transferidas e invoca serviço.

Na operação desses serviços, diferentes serviços do *Hyperledger* foram utilizados:

- *getFactory().newResource(parâmetro)*: cria um *Asset*.
- *getFactory().newConcept(parâmetro)*: cria *Concept* (Estrutura conceitual que pode ser definida no modelo. Por exemplo, uma estrutura para JSON de retorno de serviço).
- *getAssetRegistry(parâmetro)*: busca registro de *Assets* no *ledger*.
- *getParticipantRegistry(parâmetro)*: busca registro de *Participants* no *ledger*.
- *add(parâmetro)*: registra informação no *ledger*.
- *update(parâmetro)*: atualiza valor de registro no *ledger*.
- *query(parâmetro)*: usado para implementação de buscas específicas no *ledger*. Esse tipo de mecanismo foi particularmente utilizado na verificação da transferência de Asset, através do identificador da transação de transferência.

Implementação

```
query selectPayment {
   description: "Select the payment"
   statement:
      SELECT org.caac.network.PayPermission
         WHERE (transactionId == _$recibo AND
               permission.resourceId == _$resourceId AND
               permission.permissionedRequester.userId == _$userId)
}
```

Chamada

```
query('selectPayment', {recibo : newrequest.requester.recibo,
                        resourceId : newrequest.resourceId,
                        userId : newrequest.requester.userId});
```

Figura 4.29: *Query* de verificação de tranferência.

A Figura 4.29mostra como foi implementado o mecanismo de busca para verificação de uma transferência de *Asset*. A query da figura verifica se o "recibo" do usuário requerente se refere de fato a transferência (*transactionId*) do *asset* necessário e se o *asset* era de uma permissão para esse mesmo requerente (*userId*) e ao serviço requerido (*resourceId*).

Capítulo 5

CONCLUSÕES

Neste trabalho foram apresentados dois modelos de controle de acesso sensíveis ao contexto com uso de *smart contracts*. Os dois modelos foram desenvolvidos utilizando a plataforma *blockchain Hyperledger Fabric*.

No primeiro modelo apresentado, buscou-se a criação de um modelo CAAC seguindo a mesma linha de desenvolvimento apresentada na bibliografia com os trabalhos relacionados. Portanto, utiliza-se uma estrutura baseada no modelo RBAC e adiciona-se novas restrições sobre as políticas e permissões de acesso de acordo com variáveis de contexto.

Este modelo foi apresentado utilizando-se um *Asset* que representa o recurso a ser acessado, com uma política de acesso que dita as variáveis de contexto para o acesso. Outro *Asset* foi utilizado para representar o papel (*role*) do usuário e suas informações de contexto. Assim, utilizou-se um *smart contract* (*chaincode*) para verificar se o papel e as informações de contexto do usuário satisfazem as condições da política de acesso do recurso.

No segundo modelo apresentado idealizou-se um mecanismo baseado em interação por serviços com controle de permissões de forma distribuída e segura. Tipicamente, num sistema distribuído utiliza-se:

- *broker* para fazer os registros e resolução de endereços;
- provedor para registrar e oferecer os serviços;
- cliente que realiza a descoberta dos serviços e provedores, via *broker*, e invoca os serviços diretamente aos provedores.

Levando isto em consideração, no modelo apresentado o *brokering* é feito a partir do *ledger* com os registros distribuídos; o provedor é representado pelos peers que cadastram *chaincodes*

e serviços e que gerenciam as permissões, além dos *endorsing peers*; e o cliente são os *peers* que interagem com o *ledger* para validações e consultas, e interagem com os mecanismos de execução (*endorsing* e *ordering*).

A infraestrutura utilizada conta com o uso do *Hyperledger* e do conjunto de serviços que viabilizam o modelo de permissões elaborado. Assim, utiliza *Assets* como moeda de troca para obtenção de acesso. Um usuário recebe um *Asset* de permissão caso satisfaça as políticas de acesso e, posteriormente o utiliza, transferindo-o para o provedor, em troca do acesso.

Dessa forma, o modelo possui os seguintes benefícios:

- *brokering* distribuído no *ledger*;
- mecanismo de consenso (*endorsement*);
- mecanismo de permissões flexível, com quantos requisitos forem desejáveis por serviço, através dos *smart contracts* (*chaincodes*);
- permite a transferência de permissões como moeda de troca;
- evita reúso de permissões;
- pode usar validações externas e circunstanciais para dados de contexto;
- e garantia de autenticidade e consistência através das assinaturas e chaves cripto-assimétricas.

A avaliação das propostas de controle de acesso com informação de contexto (CAAC) usando a plataforma *Hyperledger* mostrou a flexibilidade e simplicidade dos modelos. Utilizando o *CAAC com smart contracts*, a permissão de acesso é gerada a partir da verificação de um *Asset* de contexto em relação a uma política de acesso vinculada ao recurso. A permissão é gerada na forma de um registro na rede *blockchain*, que precisa ser verificado para obter o acesso.

Já no *CAAC com smart contracts baseado em transferências de recursos*, cada requisito para permissão é associado a um *Asset* a ser transferido ao dono do serviço como comprovação dos requisitos necessários. Informações de contexto, que podem ser validadas por elementos externos ao sistema, podem estar associadas à atribuição dos *Assets* necessários, inclusive de maneira dinâmica, como em situações que requerem presença e proximidade.

As execuções com o *Hyperledger* mostraram que a verificação dos requisitos de acesso pode estar embutida na própria lógica dos serviços disponibilizados na rede.

Considerando que os mecanismos de verificação de autenticidade e garantia de confidencialidade do *Hyperledger* estão de acordo com a lógica definida nos sistemas de blockchain,

as transações previstas nessa proposta atendem os requisitos de segurança envolvidos com as comunicações. Providas as autenticidades das informações transmitidas, as propostas de controle de acesso desenvolvidas podem ser facilmente implementadas e embutidas nas lógicas de quaisquer serviços que queiram utilizá-las. Para isso basta especificar requisitos como *Assets* e suas transferências como validações de seus cumprimentos.

Os mecanismos de consenso e tolerância a falhas providos pelo *Hyperledger* na execução dos serviços e atualização do *ledger* provêm uma segurança adicional ao modelo vislumbrado.

A proposta do modelo de segurança apresentado atende também aos princípios de segurança apresentados no Capítulo 2 pois:

- há uma clara separação de deveres (SoD), provendo-se o gerenciamento das permissões aos serviços diretamente aos seus provedores;
- ao definir os *assets* necessários como permissões, usadas exclusivamente para o acesso a um serviço, atende-se o requisito de mínimo privilégio;
- o uso dos registros no livro razão da *blockchain* permite a responsabilização dos atores de forma praticamente inviolável;
- a possibilidade de gerar novos *assets* para concessões de privilégios de acesso, ou mesmo mudar os requisitos na prestação do serviço, permite o manutenção e revogação de direitos de acesso;
- ao basear suas operações em requisições existentes em uma infraestrutura de *blockchain*, a proposta atende também a requisitos de usabilidade.

Quanto às camadas OM-AM, também apresentadas no Capítulo 2, as propostas apresentadas podem ser representadas de forma semelhante às de Ouaddah, Elkalam e Ouahman (2016) mostrada na Figura 2.5. No entanto, substituindo o gerenciador de autorizações (AMP - *authorization manager point*), o Bitcoin e os *tokens*, por *chaincodes*, o *Hyperledger* e o uso de *assets*.

As contribuições deste trabalho foram publicadas em formato de artigo científico, intitulado "*Controle de acesso a serviços usando Smart Contracts*", no II Workshop Blockchain: Teoria, Tecnologias e Aplicações (WBlockchain 2019), do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2019).

5.1 Trabalhos futuros

Um refinamento dos serviços implementados para viabilizar as duas estratégias CAAC no *Hyperledger* ainda precisa ser realizado.

Também é preciso investigar mais profundamente e criar mecanismos de verificação de contextos, associando-os a recursos (*Assets*).

Um estudo mais efetivo sobre a formalização da proposta precisa ser realizado, inclusive para possíveis transferências do modelo para outras plataformas *blockchain*.

Aspectos de escalabilidade e operação das transações em tempo real, também precisam ser melhor investigados. Principalmente em plataformas de *blockchain* nas quais o *proof-of-work* demanda processamento significativo, assim como as operações de verificação completa das operações em relação aos blocos de registro.

REFERÊNCIAS

AL-BASSAM, M. et al. Chainspace: A sharded smart contracts platform. *ArXiv e-prints*, aug 2017.

AMATO, F. et al. An integrated framework for securing semi-structured health records. *Know.-Based Syst.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 79, n. C, p. 99–117, maio 2015. ISSN 0950-7051. Disponível em: <http://dx.doi.org/10.1016/j.knosys.2015.02.004>.

AZARIA, A. et al. Medrec: Using blockchain for medical data access and permission management. In: *2016 2nd International Conference on Open and Big Data (OBD)*. [S.l.: s.n.], 2016. p. 25–30.

BACK, A. Hashcash - a denial of service counter-measure. 09 2002.

BELLAVISTA, P. et al. Context-aware middleware for resource management in the wireless internet. *IEEE Transactions on Software Engineering*, IEEE, v. 29, n. 12, p. 1086–1099, 2003.

BERTINO, E.; KIRKPATRICK, M. S. Location-based access control systems for mobile users: Concepts and research directions. In: *Proceedings of the 4th ACM SIGSPATIAL International Workshop on Security and Privacy in GIS and LBS*. New York, NY, USA: ACM, 2011. (SPRINGL '11), p. 49–52. ISBN 978-1-4503-1032-1. Disponível em: <http://doi.acm.org/10.1145/2071880.2071890>.

BETTINI, C. et al. A survey of context modelling and reasoning techniques. *Pervasive Mob. Comput.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 6, n. 2, p. 161–180, abr. 2010. ISSN 1574-1192. Disponível em: <http://dx.doi.org/10.1016/j.pmcj.2009.06.002>.

BOONYARATTAPHAN, A. et al. Spatial-temporal access control for e-health services. In: *2010 IEEE Fifth International Conference on Networking, Architecture, and Storage*. [S.l.: s.n.], 2010. p. 269–276.

CACHIN, C. et al. Architecture of the hyperledger blockchain fabric*. In: . [S.l.: s.n.], 2016.

CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 20, n. 4, p. 398–461, nov. 2002. ISSN 0734-2071. Disponível em: <http://doi.acm.org/10.1145/571637.571640>.

CHOI, S. et al. Secure and resilient proximity-based access control. In: *Proceedings of the 2013 International Workshop on Data Management & Analytics for Healthcare*. New

York, NY, USA: ACM, 2013. (DARE '13), p. 15–20. ISBN 978-1-4503-2425-0. Disponível em: <http://doi.acm.org/10.1145/2512410.2512425>.

CLEEFF, A. v.; PIETERS, W.; WIERINGA, R. Benefits of location-based access control: A literature study. In: *Proceedings of the 2010 IEEE/ACM Int'L Conference on Green Computing and Communications & Int'L Conference on Cyber, Physical and Social Computing*. Washington, DC, USA: IEEE Computer Society, 2010. (GREENCOM-CPSCOM '10), p. 739–746. ISBN 978-0-7695-4331-4. Disponível em: <http://dx.doi.org/10.1109/GreenCom-CPSCom.2010.148>.

CORRADI, A.; MONTANARI, R.; TIBALDI, D. Context-based access control management in ubiquitous environments. In: IEEE. *Network Computing and Applications, IEEE International Symposium on*. [S.l.], 2004. p. 253–260.

DEY, A. K. Understanding and using context. *Personal Ubiquitous Comput.*, Springer-Verlag, London, UK, UK, v. 5, n. 1, p. 4–7, jan. 2001. ISSN 1617-4909. Disponível em: <http://dx.doi.org/10.1007/s007790170019>.

ELSHAFEE, A. A survey on ubiquitous computing: Towards empowering wireless network technologies. *International Journal of Computer Applications*, Foundation of Computer Science (FCS), NY, USA, New York, USA, v. 156, n. 2, p. 30–36, Dec 2016. ISSN 0975-8887. Disponível em: <http://www.ijcaonline.org/archives/volume156/number2/26683-2016912373>.

ENGLISH, M.; AUER, S.; DOMINGUE, J. Block chain technologies & the semantic web: A framework for symbiotic development. In: *Computer Science Conference for University of Bonn Students, J. Lehmann, H. Thakkar, L. Halilaj, and R. Asmat, Eds*. [S.l.: s.n.], 2016. p. 47–61.

FERRAIOLO, D.; KUHN, R. Role-based access controls. In: BALTIMORE, MARYLAND: NIST-NCSC. *Proceedings of 15th NIST-NCSC National Computer Security Conference*. [S.l.], 1992. v. 563.

FININ, T. et al. Rowlbac: Representing role based access control in owl. In: *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*. New York, NY, USA: ACM, 2008. (SACMAT '08), p. 73–82. ISBN 978-1-60558-129-3. Disponível em: <http://doi.acm.org/10.1145/1377836.1377849>.

GUPTA, A.; KIRKPATRICK, M.; BERTINO, E. A formal proximity model for rbac systems. In: *8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*. [S.l.: s.n.], 2012. p. 1–10.

HASHEMI, S. H. et al. World of empowered iot users. In: *2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI)*. [S.l.: s.n.], 2016. p. 13–24.

HU, V. C. et al. Guide to attribute based access control (abac) definition and considerations (draft). *NIST special publication*, v. 800, n. 162, 2014.

JAYABALAN, M.; O'DANIEL, T. Access control and privilege management in electronic health record: A systematic literature review. *J. Med. Syst.*, Plenum Press, New York, NY, USA,

v. 40, n. 12, p. 1–9, 2016. ISSN 0148-5598. Disponível em: <https://doi.org/10.1007/s10916-016-0589-z>.

KAYES, A. S. M.; HAN, J.; COLMAN, A. Po-saac: A purpose-oriented situation-aware access control framework for software services. In: ____. *Advanced Information Systems Engineering: 26th International Conference, CAiSE 2014, Thessaloniki, Greece, June 16-20, 2014. Proceedings*. Cham: Springer International Publishing, 2014. p. 58–74. ISBN 978-3-319-07881-6.

KHAN, A.; MCKILLOP, I. Privacy-centric access control for distributed heterogeneous medical information systems. In: *2013 IEEE International Conference on Healthcare Informatics*. [S.l.: s.n.], 2013. p. 297–306.

KHAN, M. F. F.; SAKAMURA, K. Context-awareness: Exploring the imperative shared context of security and ubiquitous computing. In: *Proceedings of the 14th International Conference on Information Integration and Web-based Applications and Services*. New York, NY, USA: ACM, 2012. (IIWAS '12), p. 101–110. ISBN 978-1-4503-1306-3. Disponível em: <http://doi.acm.org/10.1145/2428736.2428755>.

KHAN, M. F. F.; SAKAMURA, K. Fine-grained access control to medical records in digital healthcare enterprises. In: *2015 International Symposium on Networks, Computers and Communications (ISNCC)*. [S.l.: s.n.], 2015. p. 1–6.

KIRKPATRICK, M. S.; DAMIANI, M. L.; BERTINO, E. Prox-rbac: A proximity-based spatially aware rbac. In: *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. New York, NY, USA: ACM, 2011. (GIS '11), p. 339–348. ISBN 978-1-4503-1031-4. Disponível em: <http://doi.acm.org/10.1145/2093973.2094018>.

KUHN, D. R.; COYNE, E. J.; WEIL, T. R. Adding attributes to role-based access control. *Computer*, v. 43, n. 6, p. 79–81, June 2010. ISSN 0018-9162.

LI, Z.; CHU, C.-H.; YAO, W. A semantic authorization model for pervasive healthcare. *Journal of Network and Computer Applications*, Academic Press, v. 38, p. 76–87, 2014.

NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. November 2008.

OUADDAH, A.; ELKALAM, A. A.; OUAHMAN, A. A. Fairaccess: a new blockchain-based access control framework for the internet of things. *Security and Communication Networks*, v. 9, n. 18, p. 5943–5964, 2016. ISSN 1939-0122. SCN-16-0184. Disponível em: <http://dx.doi.org/10.1002/sec.1748>.

PELEG, M. et al. Situation-based access control: Privacy management via modeling of patient data access scenarios. *Journal of Biomedical Informatics*, v. 41, n. 6, p. 1028 – 1040, 2008. ISSN 1532-0464. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1532046408000506>.

RIBONI, D.; BETTINI, C. Owl 2 modeling and reasoning with complex human activities. *Pervasive Mob. Comput.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 7, n. 3, p. 379–395, jun. 2011. ISSN 1574-1192. Disponível em: <http://dx.doi.org/10.1016/j.pmcj.2011.02.001>.

SANDHU, R. Engineering authority and trust in cyberspace: The om-am and rbac way. In: *Proceedings of the Fifth ACM Workshop on Role-based Access Control*. New York, NY, USA: ACM, 2000. (RBAC '00), p. 111–119. ISBN 1-58113-259-X. Disponível em: <http://doi.acm.org/10.1145/344287.344309>.

SOUSA, J.; BESSANI, A. From byzantine consensus to bft state machine replication: A latency-optimal transformation. In: *2012 Ninth European Dependable Computing Conference*. [S.l.: s.n.], 2012. p. 37–48.

TONINELLI, A. et al. A semantic context-aware access control framework for secure collaborations in pervasive computing environments. In: ____. *The Semantic Web - ISWC 2006: 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 473–486. ISBN 978-3-540-49055-5.

UGARTE, H. *A more pragmatic Web 3.0: Linked Blockchain Data*. 03 2017.

WOOD, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, v. 151, 2014.

YARMAND, M. H.; SARTIPI, K.; DOWN, D. G. Behavior-based access control for distributed healthcare systems. *J. Comput. Secur.*, IOS Press, Amsterdam, The Netherlands, The Netherlands, v. 21, n. 1, p. 1–39, jan. 2013. ISSN 0926-227X. Disponível em: <http://dl.acm.org/citation.cfm?id=2595846.2595847>.

ZYSKIND, G.; NATHAN, O.; PENTLAND, A. S. Decentralizing privacy: Using blockchain to protect personal data. In: *Proceedings of the 2015 IEEE Security and Privacy Workshops*. Washington, DC, USA: IEEE Computer Society, 2015. (SPW '15), p. 180–184. ISBN 978-1-4799-9933-0. Disponível em: <http://dx.doi.org/10.1109/SPW.2015.27>.