4
Programação de Protocolos

Um protocolo no framework é modelado pela classe abstrata `Protocol`, onde a interface do protocolo é formada pelos parâmetros e pelos resultados dos métodos da classe, e os serviços que o protocolo oferece são definidos pela semântica dos seus métodos.

Para programar protocolos no MobiCS2 deve-se estender as classes abstratas `Protocol`, `ServiceDataUnit`, `ProtocolDataUnit` e `Address`.

Um protocolo quando é instanciado tem que passar alguns parâmetros no seu construtor, como por exemplo, o nome (`String name`) e a versão (`String version`) do protocolo, e o nó da rede onde este protocolo vai estar sendo executado (`Node owner`). Um protocolo é incluído em um nó da rede através do método `addProtocol()` da classe `Node`. Este método é chamado automaticamente pelo construtor da classe `Protocol`.

As classes `ServiceDataUnit` e `ProtocolDataUnit` representam os dados trocados entre os protocolos. Quando a classe `ProtocolDataUnit` é instanciada tem que passar como parâmetro no seu construtor uma instância da classe `ServiceDataUnit`.

Nem todo protocolo precisa ter endereço, como por exemplo os protocolos da camada física. Se um protocolo precisar da abstração de endereço, deve-se estender a classe `Address`.

4.1.
A Classe `Protocol`

Para o MobiCS2 um protocolo é um centro de serviço, com apenas um servidor e com três filas de entrada, todas com disciplina FCFS (*First Come First Serve*). O servidor não é preemptivo, isto é, o servidor trata um elemento da fila integralmente de cada vez. O servidor atende uma fila de cada vez, pegando

PUC-Rio - Certificação Digital Nº 0016060/CA

sempre o primeiro elemento de cada fila, e seguindo circularmente para a próxima fila.

Para o framework, um protocolo presta serviços de comunicação para outros protocolos, numa relação de cliente/fornecedor. Esta relação entre os protocolos pode ser representada como uma estrutura em grafo dentro do nó da rede. O método `addClient()` da classe `Protocol` informa ao protocolo quem é o seu protocolo cliente. Já o método `addProvider()` informa ao protocolo quem é o seu protocolo fornecedor. Estas informações são armazenadas em dois vetores: `clients` e `providers`. Esta relação tem que ser estabelecida nos dois protocolos. Por exemplo, imagine que temos dois protocolos, um da camada de transporte e um da camada de rede, representados respectivamente pelas instâncias `transp` e `net`. Para estabelecer a relação entre estes dois protocolos é necessário escrever estes dois comandos: `transp.addProvider(net)` e `net.addClient(transp)`.

A fila de entrada do protocolo que recebe unidades de dados que são enviadas pelos protocolos clientes, se chama `clientQueue`. E a fila de entrada do protocolo que recebe unidades de dados provenientes dos protocolos fornecedores se chama `providerQueue`.

A princípio o tamanho das filas de entrada é ilimitado. Porém existe uma forma de limitar o tamanho da fila, ou de filtrar quais unidades de dados podem entrar na fila. Isto é feito pelos métodos (*Template Method*) `acceptClientDataUnit()` e `acceptProviderDataUnit()` que decidem se uma nova unidade de dados deve entrar ou não na sua fila respectiva. Estes métodos retornam um booleano, e são implementados para sempre retornarem `true`. Estes métodos devem ser sobrescritos caso se deseje limitar ou filtrar as filas, ou até para simular a ausência de fila.

O método `send()` serve para enviar uma unidade de dados para o protocolo fornecedor. Existem duas assinaturas para este método. Uma opção é o método `send()` receber só um parâmetro, a unidade de dados a ser enviada. Neste caso o protocolo agenda o recebimento da unidade de dados para o seu protocolo fornecedor default, que é o primeiro protocolo do seu vetor `providers`. Na segunda opção do método `send()`, o protocolo fornecedor tem que ser indicado explicitamente através de um número inteiro, que é a sua

PUC-Rio - Certificação Digital Nº 0016060/CA

posição no vetor `providers`. Analogamente, existe o método `receive()` que serve para entregar uma unidade de dados para o protocolo cliente.

A terceira fila de entrada do protocolo é a fila para eventos de tempo, e se chama `timerQueue`. Da mesma forma das outras filas, esta fila tem o tamanho ilimitado, mas pode ser limitado pelo método (*Template Method*) `acceptTimer()`.

Os eventos de tempo em um protocolo servem para controlar *timeouts*. Por exemplo, um protocolo pode usar um evento de tempo para controlar o recebimento de um *acknowledgement*. Para se agendar um evento de tempo usa-se o método `setTimer()`. Este método espera apenas dois parâmetros: o tempo relativo em que o evento deve ser tratado, e um argumento (`Object`) que vai ser passado para o método que trata o evento. Como os eventos de tempo são tratados será discutido na próxima seção.

O tempo que um protocolo leva para processar uma unidade de dados ou um evento de tempo se chama "tempo de serviço", e é determinado pelo método (*Template Method*) `calculateServiceTime()`. Inicialmente este método retorna o valor do atributo `defaultServiceTime`. Este atributo inicialmente possui o valor zero, e pode ser modificado pelo método `setDefaultServiceTime()`, e consultado pelo método `getDefaultServiceTime()`. O método `calculateServiceTime()` pode ser sobrescrito para retornar outro valor que o usuário do framework deseje. Por exemplo, este método pode retornar um valor aleatório com distribuição exponencial, utilizando o atributo `defaultServiceTime` como valor médio.

No protocolo existem dois eventos síncronos que podem ser tratados: `onBeforeService` e `onAfterService`. Estes métodos são chamados imediatamente antes e depois de um serviço do protocolo.

Um protocolo, sendo um centro de serviço, pode ter o seu processamento suspenso por um tempo e depois reiniciado. Isto pode ser feito com os métodos `start()` e `stop()`. Pode-se conseguir um controle mais fino, podendo suspender o tratamento de apenas uma de suas filas. Isto pode ser feito com os métodos: `startClientQueue()`, `stopClientQueue()`, `startProviderQueue()`, `stopProviderQueue()`, `startTimerQueue()` e `stopTimerQueue()`. Esses métodos podem ser

PUC-Rio - Certificação Digital Nº 0016060/CA

usados em teste e depuração de protocolos em simulações de ordem determinística.

4.2. O Tratamento de Eventos nos Protocolos

Como vimos na seção anterior, existem três filas em um protocolo, uma para cada tipo de evento: eventos de tempo, eventos de chegada de unidade de dados do cliente e do provedor. Nesta seção veremos como estes eventos são tratados no protocolo.

As unidades de dados enviadas por protocolos clientes são processadas pelo método abstrato `whenSDU()`. A unidade de dados a ser tratada é passada por parâmetro para este método. Este método tem que ser implementado pela classe derivada.

Já para processar as unidades de dados provenientes dos protocolos fornecedores, podem existir mais de um método. Estes métodos têm a assinatura `when<DataUnit>`, onde `<DataUnit>` é o nome da classe da unidade de dados que deve ser tratada, e que é passada como parâmetro para o método. Por exemplo, um protocolo MAC pode possuir um quadro chamado ACK para confirmação de recebimento. Este quadro ACK é representado por uma classe chamada convenientemente de `Ack`. Então o método para tratar este tipo de quadro deve ter a seguinte assinatura: `whenAck(Ack frame)`. Este tipo de solução é a mesma utilizada pelo framework MobiCS (rede sem fio estruturada).

E os métodos que tratam eventos de tempo possuem a assinatura `when<Timer>`, onde `<Timer>` é o nome da classe do objeto que é passado como parâmetro para o método `setTimer()`. Por exemplo, se o protocolo MAC do exemplo anterior quiser tratar o evento de *timeout* do recebimento do quadro ACK, pode criar uma classe chamada convenientemente de `AckTimeout`. Na instância desta classe pode-se colocar dados que referenciem a que quadro ACK este *timeout* trata. Então a assinatura do método seria: `whenAckTimeout()`.

PUC-Rio - Certificação Digital Nº 0016060/CA

4.3. A Camada Física

A camada física é representada pelas classes abstratas `Physical` e `PhyPDU`. A classe `Physical` é uma extensão da classe `Protocol`. A classe `PhyPDU` estende a classe `ProtocolDataUnit`, e representa a unidade de dados que é transmitida pelo meio físico.

A classe `Physical` possui um atributo chamado `id`, que é um número inteiro que serve como identificador único de uma instância desta classe na simulação. Este atributo é utilizado pelo modelo `Topology` (conectividade do canal), mas pode também servir de base para ser utilizado na geração do endereço MAC.

A classe `PhyPDU` possui alguns atributos referentes à transmissão no meio físico. O atributo `dataRate` representa o valor da taxa de transmissão em bits por segundo. O atributo `signature` é uma *string* que representa uma espécie de "assinatura" da transmissão, como por exemplo, a codificação e a modulação utilizadas na transmissão do sinal no meio físico. Esta assinatura é utilizada pelo protocolo `Physical` do nó de destino para identificar se a `PhyPDU` recebida é originada de um mesmo protocolo ou de um protocolo diferente. Se a assinatura da `PhyPDU` for diferente da assinatura do protocolo, então a `PhyPDU` deve ser considerada como um ruído ou uma interferência. A classe `Physical` também possui esses dois atributos, que são passados para a `PhyPDU` no momento da transmissão no meio físico.

Outros atributos da `PhyPDU` são atributos relacionados com o tempo: `beginTransmission`, `beginReception` e `endReception`. O `beginTransmission` é o tempo simulado em que o `Physical` começa a transmissão no meio físico. O `beginReception` e `endReception` são, respectivamente, os tempos de início e fim de recepção da `PhyPDU` pelo `Physical` do nó de destino.

Como um protocolo `Physical` não possui nenhum protocolo fornecedor, a fila `providerQueue` não é utilizada. Os métodos `addProvider()`, `acceptProviderDataUnit()`, `send()` e `startProviderQueue()` são sobrescritos para serem métodos sem efeito.

PUC-Rio - Certificação Digital Nº 0016060/CA

O protocolo Physical possui uma nova fila de entrada chamada receptionQueue. Esta fila recebe a PhyPDU do meio físico e a ordena pela ordem cronológica do atributo endReception. No mesmo instante simulado, quando o Physical de origem transmite a PhyPDU, este é encaminhado pelo modelo de conectividade do canal diretamente para a fila receptionQueue do Physical de destino, e já é agendado o seu tratamento para o tempo endReception, independentemente dos elementos que já estejam na fila receptionQueue. Este comportamento é completamente diferente de como é tratada uma unidade de dados quando é enviada de um protocolo para outro dentro de um nó, onde inicialmente é agendada a sua entrada na fila, e não o seu tratamento, e é respeitada a disciplina FCFS. Já na recepção da PhyPDU é diferente, pode ocorrer o tratamento de mais de uma unidade de dados no mesmo instante simulado (p.ex., uma colisão).

PUC-Rio - Certificação Digital Nº 0016060/CA

4.3.1. A Transmissão no Meio Físico

Esta seção explica como o MobiCS2 trata a transmissão da PhyPDU do nó de origem até o nó de destino, através do meio físico.

Para que um protocolo Physical possa transmitir no meio físico é preciso que este esteja ligado diretamente neste meio (Medium). Isto é feito através do método attach(). Uma vez ligado na mídia, é necessário alocar um canal (Channel), o que é feito através do método allocateChannel().

Através do método transmit(), o protocolo Physical entrega a PhyPDU para o canal. Este método altera alguns dos atributos da PhyPDU (beginTransmission, dataRate, e signature).

O canal executa um laço, chamando o método transmit() da classe ChannelConnectivityModel para todos os protocolos Physical que estão alocados neste canal.

A classe ChannelConnectivityModel recebe os protocolos Physical do nó de origem e do possível nó de destino, junto com a PhyPDU original a ser transmitida. Esta classe então verifica se o nó de destino é alcançável, através do método isReachable(). Se não for alcançável, então

este método termina, passando para o próximo nó. Se for alcançável, então é feita uma cópia da PhyPDU, para poder alterar os atributos da cópia e preservar a PhyPDU original. Através do método getPropagationTime() descobre-se o tempo de propagação da transmissão. Este tempo é somado ao beginTransmission e atribuído ao beginReception. O atributo endReception é calculado da seguinte forma: beginReception + (sizeInBits/dataRate). Então o modelo de conectividade do canal chama o método processData() passando a cópia da PhyPDU. Com a PhyPDU pronta para ser entregue ao destino, esta é colocada na fila receptionQueue do Physical de destino e é agendado o seu tratamento para o tempo endReception.

4.3.2. Uma Implementação do Physical

O IEEE 802.11, operando no seu modo DCF (*Distributed Coordination Function*), é o padrão mais utilizado nos trabalhos de simulação de redes móveis ad hoc. Por isso este padrão foi escolhido para servir de base para este trabalho.

A implementação disponível no framework implementa uma versão bem simplificada da camada física do padrão IEEE 802.11, através das classes: WiFiPhyPDU (derivada da classe PhyPDU) e WiFiPhy (derivada da classe Physical). O prefixo "WiFi" utilizado no nome destas classes, vem de *Wi-Fi (Wireless Fidelity) network*, que são redes locais sem fio (WLAN) baseadas nos padrões IEEE 802.11b e 802.11a. A WiFiPhy utiliza a classe Air (derivada da classe Medium) como meio físico.

A classe WiFiPhyPDU acrescenta apenas um atributo: collision (booleano). Este atributo serve para marcar a WiFiPhyPDU que está em colisão.

A classe WiFiPhy possui o atributo radioRange para definir o alcance do sinal de rádio. Esta classe possui o método carrierSense(), que retorna um booleano, que informa se no instante simulado existe alguma transmissão em progresso no meio físico. A WiFiPhy simula uma transmissão de rádio *half-duplex* e suporta as mesmas taxas de transmissão do padrão IEEE 802.11b (1, 2, 5.5 e 11 Mb/s).

PUC-Rio - Certificação Digital Nº 0016060/CA

A classe WiFiPhy detecta colisão no receptor. Isto é feito da seguinte forma: quando uma WiFiPhyPDU é tratada, são verificadas todas as outras unidades de dados que estão na fila receptionQueue para checar se estão em colisão com esta. Isto é feito verificando os seus atributos beginReception e endReception. Se estiverem em colisão, o atributo collision de todas WiFiPhyPDU que estão em colisão é marcado.

A classe WiFiPhy antes de transmitir a WiFiPhyPDU, acrescenta ao seu tamanho os bits do preâmbulo.

A fila clientQueue do WiFiPhy tem o seu tamanho limitado a zero. O método acceptClientDataUnit() só aceita uma SDU se o WiFiPhy estiver ocioso.

4.4. A Subcamada MAC

O mecanismo básico de acesso do IEEE 802.11 é o DCF (*Distributed Coordination Function*). Este mecanismo é basicamente o método de acesso CSMA/CA (*Carrier Sense Multiple Access with Collision Avoidance*).

Neste trabalho foi modelada uma versão simplificada da subcamada MAC do padrão IEEE 802.11 DCF. O objetivo não é uma implementação completa do protocolo e sim de uma prototipação. O que interessa são os mecanismos de acesso ao meio e colisão. Dependendo da necessidade do usuário do framework, este pode estender as classes desta subcamada.

As classes que representam a subcamada MAC são:

- Classe WiFiMAC – derivada da classe Protocol.
- Classe MacAddress – derivada da classe Address.
- Quadros de dados (*Data Frames*): os quais são usados para transmissão de dados (Classe WiFiData – derivada da classe ProtocolDataUnit).
- Quadros de Controle (*Control Frames*): os quais são usados para controlar o acesso ao meio (Classes WiFiRTS, WiFiCTS e WiFiACK – derivadas da classe ServiceDataUnit).

PUC-Rio - Certificação Digital Nº 0016060/CA

Alguns exemplos das simplificações adotadas são:

- Os quadros de gerenciamento (*Management Frames*) não foram implementados. Estes são quadros que são transmitidos do mesmo jeito que os quadros de dados, para troca de informação de gerenciamento, mas não são encaminhados para as camadas de cima.
- Não é gerado nenhum tipo de campo de CRC nos quadros.
- Não foi implementada a função de *Power Saving*, porque esta função não é suportada pelo modo de operação "ad hoc" do IEEE 802.11.

4.5. A Camada de Rede

A partir do estudo dos diferentes protocolos de roteamento para redes móveis ad hoc (ver Apêndice), identificamos componentes comuns às duas principais classes de protocolos (*table-driven* e *on-demand*). Estes componentes e suas interações serão discutidas nesta seção.

PUC-Rio - Certificação Digital Nº 0016060/CA

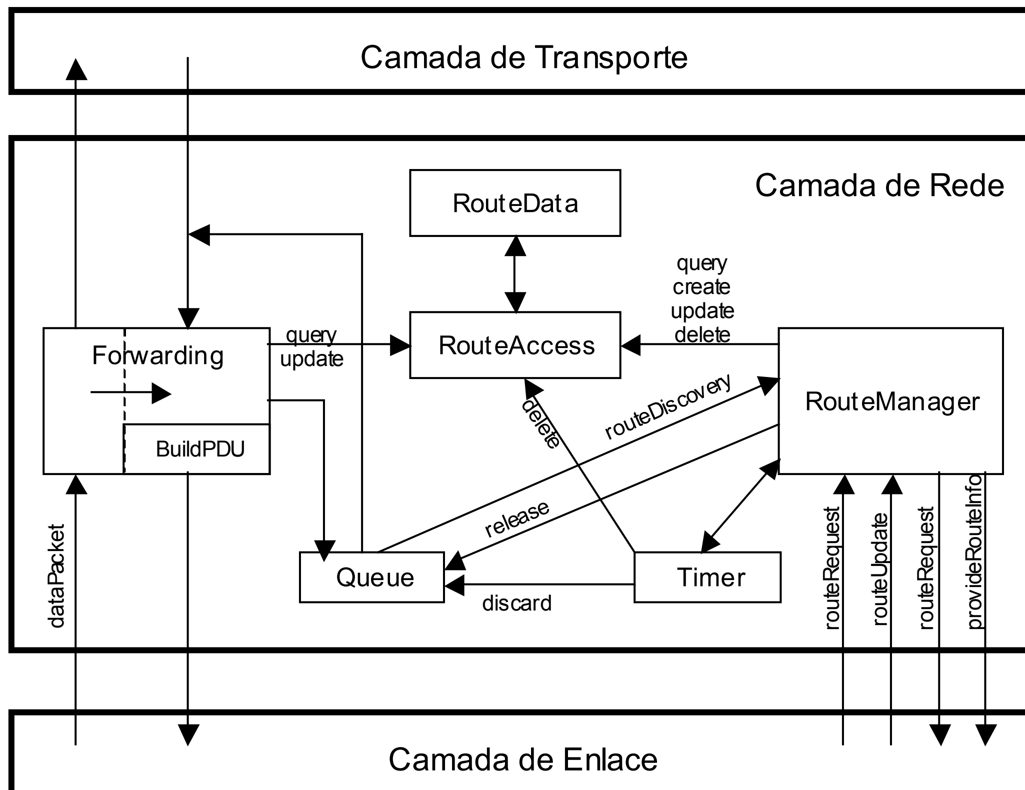

Figura 10 – Diagrama dos componentes de um protocolo de roteamento para MANET

O componente *Forwarding* serve para encaminhar os pacotes de dados que possuem uma rota conhecida. Os pacotes de dados entregues pelos protocolos clientes (p.ex., camada de transporte) são encaminhados para o protocolo fornecedor (camada de enlace). E os pacotes de dados recebidos da camada de enlace são repassados para a camada de transporte.

O componente *BuildPDU* serve para construir uma PDU da camada de rede para ser enviada para a camada de enlace.

O componente *Queue* possui uma fila onde são armazenados os pacotes que estão pendentes para encaminhamento devido falta de rota.

O *RouteManager* é o componente responsável pelo serviço de roteamento, isto é, é o responsável por descobrir, manter e remover rotas.

O componente *RouteData* possui uma ou mais estruturas de dados para armazenamento das rotas. Os protocolos *table-driven* normalmente possuem estruturas como tabelas e listas, e os protocolos *on-demand* podem possuir caches.

O componente *RouteAccess* provê uma interface para acessar as rotas armazenadas no *RouteData*, sem precisar expor qual é a estrutura de dados utilizada no *RouteData*.

O componente *Timer* é utilizado para a geração de eventos de tempo para o protocolo de roteamento.

PUC-Rio - Certificação Digital Nº 0016060/CA

4.5.1. Principais Interações entre os Componentes

Quando um pacote de dados é entregue pelos protocolos da camada de transporte, este é passado ao componente *Forwarding*. Este componente consulta (*query*) o *RouteAccess* para ver se possui uma rota para poder encaminhar o pacote. Se *RouteData* possuir a rota, o pacote é passado para o *BuildPDU*, que constrói a PDU de rede e envia para a camada de enlace.

Caso o *RouteData* não possua a rota, existem duas possibilidades, dependendo da classe do protocolo. Se for um *table-driven*, o pacote é simplesmente descartado. Se for um *on-demand*, o pacote é enfileirado no componente *Queue*. Este componente pede então ao *RouteManager* para descobrir a rota do pacote (*routeDiscovery*).

Para descobrir a rota, o *RouteManager* envia um pacote de requisição de rota (*routeRequest*) e aciona o *Timer* para controlar o *timeout* da requisição. Se chegar a informação da rota (*routeUpdate*) antes de estourar o tempo, então o *RouteManager* atualiza o *RouteData* e pede ao *Queue* para liberar (*release*) o pacote. O pacote liberado é enviado novamente para o *Forwarding*, que o envia para a camada de enlace. Caso ocorra o *timeout*, o *Timer* pede ao *Queue* para descartar o pacote (*discard*).

Quando a camada de enlace entrega um pacote de dados (*dataPacket*), este é processado pelo *Forwarding*. Se o pacote de dados for para o próprio nó, então o *Forwarding* entrega os dados à camada de transporte. Caso contrário, o *Forwarding* encaminha o pacote para o próximo nó. Os protocolos de roteamento, principalmente da classe *on-demand*, podem possuir uma otimização para aprender novas rotas com os pacotes de dados que chegam. Neste caso o *Forwarding* pode atualizar o *RouteData* (*update*).

A camada de enlace pode enviar um pacote de controle do tipo *routeUpdate*. Este pacote contém informação para atualização de rotas, e é processado pelo *RouteManager*, atualizando a informação no *RouteData*.

O *RouteManager* pode enviar dois tipos de pacotes de controle para a camada de enlace: *routeRequest* e *provideRouteInfo*. O *routeRequest* é normalmente utilizado pelos protocolos *on-demand* para descobrir novas rotas. O *provideRouteInfo* serve para divulgar informações do *RouteData* para outros nós.

O *provideRouteInfo* pode funcionar de diferentes formas. Nos protocolos da classe *on-demand*, é enviado um pacote *provideRouteInfo* como resposta ao recebimento de um pacote *routeRequest* e o *RouteData* possui a rota pedida. Ou quando recebe um pacote de controle com informação de atualização de rotas (*routeUpdate*) que deve ser encaminhado para outro nó. Normalmente neste caso, o *RouteManager* aproveita a informação que chegou no *routeUpdate*, e também atualiza o seu *RouteData*.

Nos protocolos da classe *table-driven*, são gerados pacotes do tipo *provideRouteInfo* devido à ocorrência de algum evento. Este evento pode ser a passagem de um tempo (*Timer*) desde a última vez que gerou um *provideRouteInfo*, ou porque já ocorreram mudanças significativas no seu *RouteData* desde a última vez que gerou um *provideRouteInfo*.

PUC-Rio - Certificação Digital Nº 0016060/CA

As informações que são enviadas num *provideRouteInfo* podem variar no grau de detalhe. Dependendo do protocolo, ou do momento em que são geradas, estas informações podem ser totais, isto é, toda a informação contida no *RouteData*, ou podem ser parciais.

O *Timer* pode ser usado para periodicamente remover (*delete*) dados obsoletos do *RouteData*.

4.5.2.
Uma Possível Implementação como Protocolo do MobiCS2

Uma possível forma de implementar um protocolo de roteamento no MobiCS2 pode ser a seguinte:

- Criar uma nova classe para o protocolo de roteamento, derivada da classe `Protocol`.
- Representar os componentes *Forwarding* e *RouteManager* como duas Interfaces Java, que devem ser implementadas pela classe do protocolo.
- O componente *BuildPDU* deve ser um método da classe (*Template Method*).
- A funcionalidade do componente *Timer* é conseguida através do modelo de programação de eventos de tempo da classe `Protocol`.
- Deve-se criar uma nova classe para cada um dos componentes: *Queue*, *RouteData* e *RouteAccess*. As instâncias destas classes devem ser incorporadas no protocolo, numa relação de composição.
- As interações entre os componentes são implementadas através de chamadas de métodos.

No próximo capítulo apresentaremos a implementação de um protocolo de roteamento que segue este modelo.

PUC-Rio - Certificação Digital Nº 0016060/CA