Implementação do Protocolo de *Time-Stamp* da Infra-Estrutura de Chaves Públicas X.509

Eduardo Arantes Rocha de Oliveira[1], Luiz Antônio da Frota Mattos[1]

[1]Universidade de Brasília (UnB) – Instituto de Ciências Exatas – Departamento de Ciência da Computação (CIC) – Campos Universitário – Asa Norte – CEP 70910-900 – Brasília, DF

eduardo.oliveira@persocom.com.br, frota@cic.unb.br

Abstract. *Public Key Infrastructure based solutions that need evidence that some document or digital signature existed before a particular time should employ the time-stamp protocol (TSP). A programming library which supports the protocol's data structures, rules and procedures is made needed to comply this requirement. Such library must implement the specification correctly, providing an elegant and robust architecture to its users, and also worry about related specifications. Thus the library usage becomes easy and results in simple and reliable application code.*

Resumo. *Soluções baseadas em infra-estrutura de chaves públicas que necessitam de evidência da existência de um documento ou assinatura digital antes de um determinado horário devem utilizar o protocolo de* time-stamp *(TSP). Para isso, precisam de alguma biblioteca de programação que forneça suporte às estruturas de dados, regras e procedimentos definidos pelo protocolo. Tal biblioteca deve implementar corretamente a especificação, oferecendo uma arquitetura elegante e robusta a seus usuários, além de se preocupar com especificações correlatas. Dessa forma, seu uso se torna fácil e resulta em código de aplicação simples e confiável.*

1. Introdução

Documentos em formatos digitais têm obtido grande destaque e importância em serviços outrora disponíveis apenas a documentos registrados em papel. A substituição da forma em que documentos são armazenados para arquivos eletrônicos pode ajudar a reduzir custos, agilizar a busca de informações e facilitar o gerenciamento de uma grande quantidade de documentos, de acordo com De Rolt e Soares (2003). Essa transição, entretanto, é um caso típico onde a solução de um problema traz consigo a criação de outros. Documentos registrados em papel têm a propriedade de dificultar a alteração de seu conteúdo sem que seja percebido por um especialista, enquanto os dados de documentos digitais podem ser facilmente modificados. Quando o documento em papel está envolvido em algum processo onde exista algum prazo de submissão, ele precisa ser protocolado em cartório, onde recebe um carimbo de data e horário, além da assinatura do tabelião. A evidência de que o documento foi criado antes de determinada data está na confiança do carimbo e da assinatura que foram apostos ao documento. O mecanismo aplicado a documentos digitais que possui essa característica chama-se *time-stamp.*

Haber e Stornetta (1991) foram os primeiros a descrever um mecanismo de *time-stamp* de documentos eletrônicos a receber maior atenção. Explicando como um esquema ingênuo funcionaria, os autores identificam os pontos fracos e propõem

soluções aos problemas encontrados. A idéia é que exista um provedor de serviço de *time-stamp* confiável, ou seja, as pessoas envolvidas no protocolo acreditariam que o provedor nunca usasse de má fé na geração de *time-stamps.* Além disso, o esquema aplica o *time-stamp* sobre o cálculo do *hash* do documento, assinando digitalmente a cadeia de *bytes* gerada. Eles ainda sugerem um protocolo que encadeia *time-stamps* linearmente e outro que distribui a confiança entre os usuários do protocolo, desfazendo-se da necessidade de um provedor confiável. Massias e Quisquater (1997) reforçam a idéia do encadeamento e distribuição da confiança, enquanto Costa, Custódio, Dias e De Rolt (2003) propõem o uso de encadeamento em forma de árvore binária.

O grupo de trabalho da *Internet Engineering Task Force* (IETF) responsável pela definição de padrões relacionados à infra-estrutura de chaves públicas (PKIX) lançou o *Time-Stamp Protocol* (TSP) [Adams, Cain, Pinkas, Zuccherato] em 2001. Como o protocolo sofreu uma série de revisões até chegar a sua redação final, seu uso é extremamente recomendado em aplicações relacionadas à infra-estrutura de chaves públicas. Além disso, o formato de assinaturas eletrônicas de longa duração [Pinkas, Ross, Pope 2001] utiliza o TSP em sua especificação. O grupo de trabalho do IETF responsável por serviços notários (LTANS) também menciona o TSP em um documento inicial de requisitos de um serviço de arquivamento em longo prazo [Wallace, Brandner, Pordesh 2004]. Portanto, as empresas que lidam com esse tipo de infra-estrutura e as que fornecem tecnologia nessa área devem procurar implementar o TSP.

O esforço de implementação desse protocolo resultou na escrita do presente artigo. Patrocinado por uma empresa especializada em soluções de infra-estrutura de chaves públicas, o objetivo do trabalho era a criação de uma biblioteca de programação que permita a utilização do TSP em aplicações de segurança que envolvam protocolação de documentos ou assinaturas digitais.

O artigo apresenta as principais preocupações que uma biblioteca que implemente o TSP deve ter, relacionando os requisitos do protocolo com requisitos criados a partir de documentos correlatos. Posteriormente, é mostrada a solução obtida para os problemas apresentados, ilustrada com trechos do código que formam a biblioteca construída.

2. O Protocolo de *Time-Stamp*

A especificação do *Time-Stamp Protocol* (TSP) mostra um protocolo semelhante ao proposto inicialmente por Haber e Stornetta, utilizando o esquema de um provedor de serviço confiável. O protocolo em si é bastante simples: um usuário do serviço envia uma requisição com formato bem definido a um provedor (chamado de *Time-Stamp Authority* - TSA), que retorna a resposta contendo o *time-stamp.* A especificação define as mensagens e estruturas de dados utilizadas no protocolo e descreve como utilizar o TSP sobre alguns protocolos de transporte.

O primeiro requisito da construção de uma biblioteca desse tipo é a implementação correta e completa do protocolo. A especificação deve ser analisada e entendida por inteiro, e os documentos referenciados também devem ser estudados, como foi o caso da Sintaxe de Mensagens Criptográficas (CMS) [Housley 1999], que empresta a definição de algumas estruturas de dados ao TSP. A pesquisa por

documentos correlatos ao protocolo também é importante, podendo trazer à tona novos requisitos à obtenção de uma biblioteca realmente consistente e funcional. Nessa procura, foi encontrada a especificação de requisitos de uma política a ser implementada por uma TSA [Pinkas, Pope e Ross 2003], documentada no RFC 3628, e um perfil (*profile)* do protocolo [ETSI 2002], que limita o uso de alguns campos das estruturas de dados do TSP. Portanto, a partir da redação do TSP, das especificações referenciadas e dos documentos correlatos foram descobertos os possíveis atores da biblioteca e os casos de usos de como o protocolo poderia ser utilizado no contexto da infra-estrutura de chaves públicas.

Pensando no protocolo como um tipo de serviço, temos que o solicitante de *time-stamp* é um assinante, a TSA é um provedor de serviço, e aqueles que consideram o *time-stamp* gerado uma prova de que algum dado existia antes de determinada data são dependentes do serviço, ou simplesmente dependentes. O desafio da implementação era abranger o máximo de casos de uso desses usuários, além de permitir que o código cliente da biblioteca fosse robusto e elegante. Além do fluxo tradicional do envio da requisição do assinante ao provedor e a respectiva geração do *time-stamp*, também foram levados em conta a verificação do *time-stamp* por um dependente e o uso do *time-stamp* em assinaturas digitais. Comum a todos esses procedimentos está a validação das mensagens e estruturas do protocolo, que também abrange o conceito de perfis.

O primeiro problema que surge é a manipulação da notação abstrata ASN.1. Deveria ser possível a transformação das estruturas de dados definidas pelo protocolo em código-fonte, através de algum tipo de compilador. Outro aspecto é a implementação das regras do protocolo, o que não pode ser feito por algum autômato. Regras como "se o campo 'certReq' [da requisição] (...) for falso, então o campo 'certificates' da estrutura SignedData não deve estar presente na resposta" só podem ser implementadas manualmente.

Um conjunto especial de regras do protocolo é justamente a validação das mensagens e estruturas definidas. O TSP define apenas duas mensagens (requisição e resposta), porém uma terceira estrutura de dados também deve ser validada independentemente: o *token* de datação, que corresponde ao *time-stamp* definido pelo protocolo. O esquema de validação pode ser estendido para dar suporte aos perfis do TSP, verificando se as mensagens ou o *token* contém apenas valores limitados por determinado perfil que assinante e/ou provedor se comprometem a seguir. Apesar de ser um tópico separado do TSP, utilizar a idéia de perfis pode facilitar em muito a implementação tanto de uma aplicação cliente quanto de uma aplicação de servidor.

A grande preocupação, entretanto, é relativa à camada de transporte. O único protocolo de transporte em que o TSP especifica mais de dois tipos de mensagens é aquele baseado em *socket* (`tcp`). Os outros protocolos citados (`ftp`, `http`, e e-mail) devem apenas implementar o padrão requisição-resposta, cada qual com alguma peculiaridade: requisições `http`, por exemplo, devem ter o *Content-Type* definido como `application/timestamp-query`, enquanto a resposta deve ter esse atributo definido como `application/timestamp-reply`. Dessa maneira, quando um assinante envia uma requisição via `ftp` ou `http`, a conexão com o provedor deve permanecer ativa até que uma resposta seja enviada de volta. Como recursos de rede nem sempre são abundantes, ambas as partes do protocolo sofrem com a manutenção da conexão, principalmente o servidor, que deve ser capaz de atender várias requisições ao mesmo tempo. A especificação do TSP via *socket* é a única que contorna esse problema,

lançando mão da idéia de *polling*, ou seja, requisições de tempos em tempos, em intervalos definidos pelo servidor, para saber se a resposta já está pronta. Dessa forma, não há necessidade de manter a mesma conexão com o servidor: depois de terminado o tempo de espera, o assinante dispara uma requisição `tcp` com a referência obtida na solicitação anterior, e procede assim sucessivamente até o recebimento da resposta. Essa é uma característica fundamental da concepção do TSP e teve grande importância na solução apresentada.

3. Implementação do protocolo

A resolução do primeiro problema apresentado - o de transformar as estruturas de dados em código-fonte - foi resolvido utilizando-se um compilador proprietário da empresa patrocinadora do projeto. Apesar de existirem diversos compiladores comerciais disponíveis, a utilização de um compilador próprio permite maior flexibilidade à empresa em futuras extensões do uso da ferramenta. O compilador utiliza-se da definição ASN.1 anexa à especificação do protocolo como entrada, e gera uma classe[1] para cada estrutura de dados definida. As definições ASN.1 importadas também são utilizadas como entrada e cada definição utilizada pelo compilador gera classes agrupadas em pacotes distintos. Assim, tendo a base da implementação o pacote `pkix.tsp`, o padrão é ter `pkix.tsp.asn1` como o nome do pacote das classes geradas a partir da definição do TSP.

Resolvido o problema inicial, o passo seguinte era a implementação das regras do protocolo. O padrão adotado foi o de não modificar as classes geradas automaticamente, construindo do zero as classes que implementam o protocolo. As classes ASN.1, então, seriam utilizadas apenas na codificação e decodificação de mensagens em conteúdo binário. Caso o gerador de código seja modificado (para suportar um novo tipo de codificação – DER, BER, CER, XML, por exemplo), a mudança não causará impacto nessas novas classes – desde que o código gerado mantenha a mesma interface. A Tabela 1 mostra o mapeamento entre as classes ASN.1 e as classes que realmente implementam o protocolo. O nome das classes ASN.1 é o mesmo nome das estruturas especificadas pelo TSP. Além disso, nota-se a relação de um-para-um entre as duas colunas.

Tabela 1. Mapeamento entre as classes que implementam o protocolo (classes principais) e as classes ASN.1.

Classes Principais `pkix.tsp`	**Classes ASN.1** `pkix.tsp.asn1`
`TSPRequest`	`TimeStampReq`
`TSPMessageImprint`	`MessageImprint`
`TSPPolicy`	`TSAPolicyId`
`TSPResponse`	`TimeStampResp`
`TSPStatusInfo`	`PKIStatusInfo`
`TSPStatus`	`PKIStatus`
`TSPFailureInfo`	`PKIFailureInfo`
`TSPToken`	`TimeStampToken`
`TSPTokenInfo`	`TSTInfo`
`TSPAccuracy`	`Accuracy`

Alguns padrões adotados na implementação dessa parte da biblioteca merecem destaque. O primeiro foi evitar construtores do tipo "JavaBeans", ou seja, instanciar

[1] Toda a implementação foi feita em Java.

uma classe por meio de um construtor sem argumentos e depois modificar os atributos internos através de métodos `set`. Dessa maneira, tanto a interface (estrutura estática) quanto o código de execução (estrutura dinâmica) da classe se tornam mais consistentes. Um exemplo típico é a criação de um *token* de datação: instanciá-lo e só depois configurar o algoritmo de assinatura, o algoritmo de *hash* e o certificado utilizado permitiria que a classe tivesse estados inválidos entre as chamadas:

```
TSPToken token = new TSPToken();
token.setEncryptionAlgorithm(“RSA”);
token.setMessageDigestAlgorithm(“SHA1”);
token.setPrivateKey(key);
token.setSigningCertificate(certificate);
```

Um construtor que obtém todas as informações necessárias para a manutenção do estado correto do *token* substitui essa série de instruções:

```
public TSPToken(TSPTokenInfo info, String messageDigestAlgorithm,
String encryptionAlgorithm, PrivateKey key, X509Certificate
signingCert)...
```

Além disso, toda manipulação de atributos internos de uma classe é feita pela própria classe, com exceção do *framework* de validação, que faz uso de alguns métodos `get` disponíveis. A criação das mensagens de requisição e resposta pode ser feita através de fábricas configuradas para gerar mensagens com os mesmos campos, simplificando o código do cliente da biblioteca. A classe `RequestFactory`, por exemplo, possui atributos comuns à classe `TSPRequest`, que são utilizados para fabricar novas requisições, podendo ser considerada um protótipo. Depois de configurar a fábrica com os parâmetros desejados, basta chamar o método criador, que calcula o *hash* da cadeia de *bytes* passada como parâmetro:

```
RequestFactory factory = new RequestFactory();
factory.setCertReq(true);
factory.setReqPolicy(new TSPPolicy(“0.4.0.2023.1.1”));
InputStream in = new FileInputStream(file);
TSPRequest request = factory.createRequest(in);
```

Outra facilidade é a implementação de *time-stamping* de assinaturas digitais utilizando a sintaxe de mensagens assinadas do CMS. Enquanto o *time-stamp* de um documento é apenas o *token* de datação, em uma mensagem assinada (`SignedData`) o *token* é um atributo não autenticado do assinante da mensagem (`SignerInfo`). A biblioteca fornece suporte a esse mecanismo, evitando que o cliente tenha esforço extra na codificação de aplicações que manipulam tais estruturas.

3.1. Validação

Para resolver o problema da validação, foi criado um *framework* que utiliza dados internos às classes principais da biblioteca através de métodos `get`, quebrando de certa forma o padrão adotado de maximizar o encapsulamento. Essa decisão foi tomada devido ao caráter estático das estruturas do protocolo (que dificilmente serão modificadas por documentos posteriores) e à dificuldade que seria promover a validação de diversos aspectos de uma estrutura em uma única classe. Além disso, como o *framework* foi projetado para dar suporte à idéia de perfis, novas validações seriam possíveis por meio de polimorfismo. Essa flexibilidade é promovida pelo fato de que

cada validação corresponde a uma classe, que lança uma exceção específica, encapsulando o fato gerador de erro de forma particular. Exemplificando, a parte do *framework* responsável pela validação da requisição - do ponto de vista de um perfil em particular - pode lançar uma exceção trazendo a mensagem de resposta pronta, contendo todos os erros encontrados na requisição. Essa resposta, então, é enviada ao cliente.

Ainda sobre o *framework* de validação, existem três tipos de dados que podem ser validados: a requisição (basicamente apenas do ponto de vista de um perfil), a resposta (verificação da compatibilidade entre seus dois únicos campos, `status` e `timeStampToken`) e o *token* de datação (baseado em perfil e nas regras do protocolo). Para cada um desses tipos existe uma interface de validação específica:

```
public interface RequestValidator extends Validator {

    void validateRequest(TSPRequest request) throws
ValidatorException;

}

public interface ResponseValidator extends Validator {

    void validateResponse(TSPResponse response) throws
ValidatorException;

}

public interface TokenValidator extends Validator {

    void validateToken(TSPToken token, TokenValidatorParams
params) throws ValidatorException;

}
```

Tomando o *token* como exemplo, existem várias classes que desempenham sua validação, implementando `TokenValidator`. A fim de tornar o processo de validação mais simples do ponto de vista do usuário da biblioteca, foi construída uma classe (`TokenMainValidator`) que implementa o padrão de projeto *Composite* [Gamma, Helm, Johnson e Vlissides 2000], agrupando diversos validadores em apenas um. Dessa maneira, o usuário da biblioteca pode utilizar uma única classe validadora para desempenhar diversas verificações, e ainda assim obter exceções particulares (caso ocorram). Essa classe é usada diretamente pelo *token* de datação (`TSPToken`) para validá-lo:

```
public void validate(TokenValidatorParams params) throws
ValidatorException {

    new TokenMainValidator().validateToken(this, params);

}
```

3.2. Implementação da camada de transporte

Ao lado do *framework* de validação, a implementação da camada de transporte é um dos destaques da biblioteca TSP. Ao contrário da implementação das regras do protocolo e do mecanismo de validação, a camada de transporte não concerne dependentes do protocolo, apenas assinantes e provedores. A implementação foi projetada sob a visão de cada um desses atores independentemente, já que possuem requisitos específicos. Entretanto, a necessidade de se obter uma interface única para as implementações de vários protocolos de transporte era um requisito comum tanto ao código responsável pela parte do cliente quanto pela parte do servidor. Os detalhes de cada mecanismo ficam transparentes ao usuário da biblioteca, que enxerga sempre a mesma interface. A

classe `Requester`, por exemplo, é implementada por `HttpRequester` para fornecer suporte ao cliente enviar requisições para servidores escutando `http`:

```
public interface Requester {
    TSPResponse query(TSPRequest request) throws
RequesterException;
}
```

`HttpRequester` não disponibiliza nenhum outro método público, apenas possui um construtor específico para suportar URL's.

Dentre os quatro protocolos citados pela especificação, apenas o `http` foi inteiramente implementado, enquanto que a implementação do `tcp` não chegou a ser exaustivamente testada. Devido à natureza da especificação do TSP quanto à permanência de conexões `http` ou `ftp` até que o servidor responda uma requisição, uma aplicação cliente deve estar preparada para lidar com bloqueio de I/O durante as requisições. Mal projetada, a aplicação pode ter a interface gráfica bloqueada até a chegada da resposta, caso a conexão seja realizada na mesma *thread* responsável pela atualização dos elementos gráficos. Para resolver essa dificuldade, uma classe que implementa os padrões de projeto *Adapter* e *Observer* foi criada. Quando um objeto dessa classe (`ObservableRequesterAdapter`) é utilizado, ele cria uma nova *thread* e dentro dela delega o processo de requisição ao objeto que realmente desempenhará a comunicação com o servidor:

```
public void performQuery(TSPRequest request) throws
RequesterException {
    new Thread(this).start();
}
```

Todos os objetos interessados na chegada da resposta gerada pelo servidor, ou seja, os observadores, devem se registrar na classe adaptadora (o sujeito observado). No momento em que a resposta chega, o sujeito notifica todos seus observadores, disponibilizando a resposta através de um método `get`. Um caso típico de um observador é a controladora de interação de uma aplicação *desktop*: ao receber a notificação do recebimento da resposta, a controladora poderia promover alguma atualização da interface gráfica, indicando que a requisição enviada foi atendida.

No caso de conexões via `tcp`, esse tipo de arquitetura não é tão necessário, já que a especificação foi projetada para o servidor fornecer uma resposta – contendo uma referência para pesquisa posterior – assim que receba alguma requisição. O TSP baseado em `tcp` possui tipos de mensagens exclusivas a conexões via *socket*. O uso dessas mensagens depende do estado em que se encontra a transação, o que sugere a utilização do padrão de projeto *State Machine*, ou Máquina de Estados. A Figura 1 mostra a máquina de estados de uma transação TSP via `tcp` do ponto de vista do cliente. Dois detalhes devem ser observados na figura: o primeiro é o *loop* formado quando o cliente recebe sempre uma mensagem com referência de *polling* (`pollReq`); o segundo é outro *loop* formado quando é recebida uma resposta parcial (`partialMsgRep`), que também contém uma referência de *polling*.

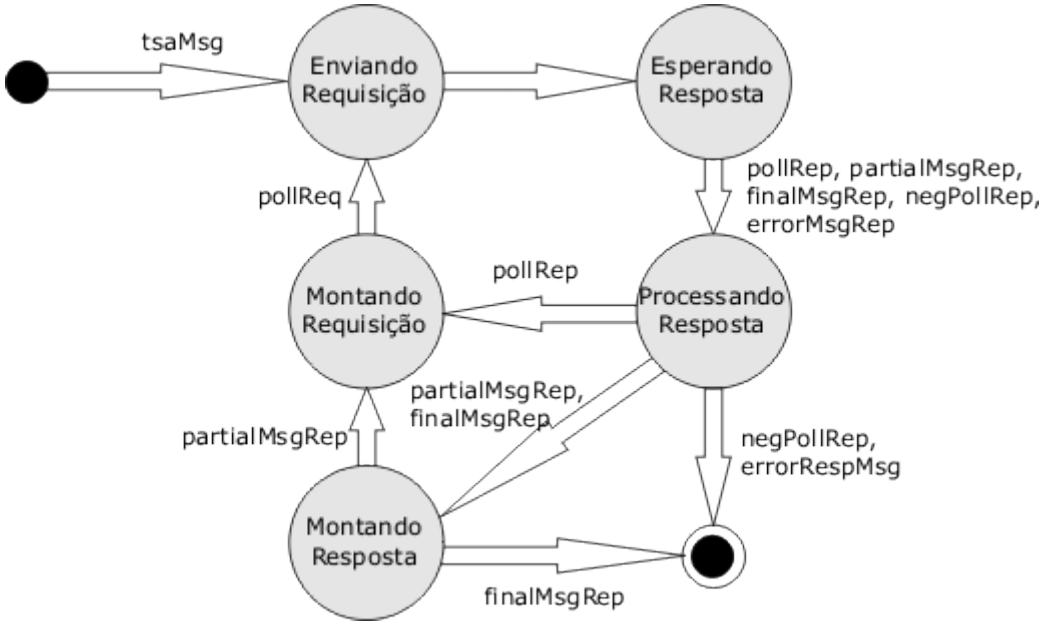

Figura 1. Máquina de estados de uma transação TSP via *socket*, do ponto de vista do cliente.

Essa estratégia é comum tanto ao código do cliente quanto ao código do servidor, porém é invisível pelo usuário da biblioteca, que lida apenas com as classes de alto nível da requisição, da resposta, e da respectiva interface de comunicação. Além do grupo de classes que representam os estados, cada mensagem do mini-protocolo `tcp` também possui uma classe que a representa. O aspecto interessante é o uso do padrão *Marker Interface*, que significa implementar uma interface sem métodos para promover segurança de tipo entre classes com o mesmo propósito. Esse padrão também foi implementado no *framework* de validação, onde a classe `Validator` é tal interface.

Nesse caso, existem duas interfaces marcadoras: `RequestMessage` e `ResponseMessage`, que são implementadas inclusive pela requisição (`TSPRequest`) e resposta (`TSPResponse`) do TSP. Todas as mensagens `tcp` que partem do cliente implementam a primeira interface, enquanto as mensagens que partem do servidor implementam a segunda. Esse tipo de mecanismo foi fundamental para a implementação `tcp` manter a mesma interface dos outros protocolos.

A codificação da parte do servidor foi guiada por em esquema de delegação definido na fase de análise. Com o objetivo de proteger a máquina responsável pelas assinaturas dos *tokens* de datação (que deve ter acesso à chave privada do sistema, possivelmente em um módulo criptográfico seguro - HSM), o servidor não atenderia às requisições diretamente, e sim por meio de *gateways*. A idéia é que a máquina principal não tenha qualquer porta disponível para conexões à rede externa, requisito viável quando se utiliza um *firewall* na arquitetura da rede. Dessa forma, apenas os *gateways* poderiam se conectar a essa máquina, enquanto que os assinantes do serviço obteriam *tokens* através dos *gateways*.

O protocolo sugerido para a comunicação com a máquina central é o `tcp`, por ser o mais robusto e melhor especificado pelo TSP. Nesse tipo de arquitetura, é necessária inclusive a existência de um *gateway* escutando conexões `tcp`, já que a máquina central (cerne) não tem acesso público disponível. Na implementação da biblioteca, esse cenário foi o que direcionou a maioria das decisões de projeto, porém a

utilização mais simplista também não foi ignorada (por exemplo, um servidor `http` sem uso de *gateways*). A consideração desses aspectos pela arquitetura da biblioteca proporciona bastante flexibilidade na implantação de um servidor TSP. A Figura 2 mostra o esquema sugerido.

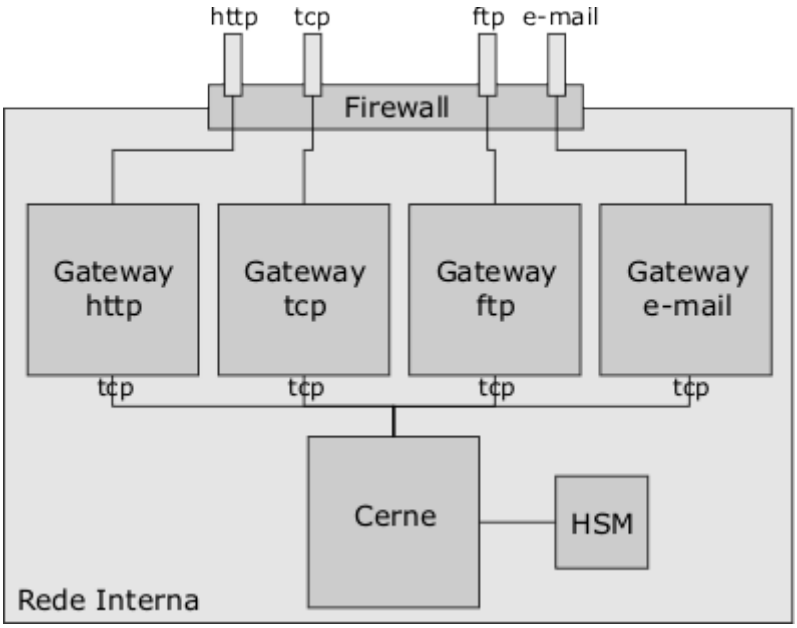

Figura 2. Esquema sugerido para implantação do servidor, onde uma máquina central (Cerne) é acessada apenas através de *Gateways*.

Para escutar requisições de um determinado protocolo de transporte, a biblioteca utilizou a idéia de *containers*¸ abstração análoga aos *requesters* utilizado no lado do cliente. Como a plataforma Java provê uma especificação padrão de um *container* `http` (tecnologia Servlet), a biblioteca baseou a implementação desse protocolo em tal tecnologia. O *container* `http`, portanto, é alguma implementação da especificação Servlet, como o Apache Tomcat. Em vez de processar as requisições eles mesmos, os *containers* foram projetados para delegar essa tarefa a `ContainerDelegate`'s, que utilizam uma nova *thread*. Depois de terminado o processamento, os *delegates* respondem aos assinantes e devolvem a *thread* utilizada para o *pool* que o *container* possui.

4. Conclusão

O trabalho realizado como esforço de pesquisa resultou na construção de uma biblioteca de classes pronta para ser utilizada em projetos relacionados ao protocolo de *time-stamp* (TSP). A biblioteca pode ser usada na implantação de um servidor TSP, em aplicações cliente - que geram requisições a servidores - e também em aplicações que dependam da existência de um *token* de datação no formato TSP.

A construção dessa biblioteca mostra que a implementação de protocolos de segurança envolve conhecimentos sólidos em engenharia de software e redes de computadores. A procura por soluções elegantes e a visão da biblioteca sob a ótica de seus possíveis usuários tornou a arquitetura das classes mais complexa, e a curva de aprendizado para dominá-la acaba sendo mais acentuada. Entretanto, depois de compreendida as soluções adotadas, seu uso se torna fácil e resulta em código simples,

conciso e robusto. Além disso, o usuário poderá encontrar facilidades além da própria especificação do protocolo, como a implementação da idéia de perfis.

Referências Bibliográficas

Wallace, C., Brandner, R., Pordesh, U. (2004), “Long-term Archive Service Requirements”, Disponível em <http://www.ietf.org/internet-drafts/draft-ietf-ltans-reqs-00.txt>, último acesso em Março/2004, expira em Agosto/2004.

Pinkas, D., Pope, N. e Ross, J. (2003) "Policy Requirements for Time-Stamping Authorities (TSAs)", RFC 3628. Disponível em <http://www.ietf.org/rfc/rfc3628.txt>, último acesso em Novembro/2003.

Costa, V., Custódio, R.F., Dias, J.S. e De Rolt, R.C. (2003) "Protocolização Digital de Documentos Eletrônicos". Disponível em <http://www.bry.com.br/downloads/artigo/Protocolacao Digital de Documentos Eletronicos.pdf>, último acesso em Novembro/2003.

ETSI (2002) "Time Stamp Profile", ETSI TS 101 861 V1.2.1. Disponível em <http://docbox.etsi.org/EC_Files/EC_Files/ts_101861v010201p.pdf>, último acesso em Novembro/2003.

De Rolt, R.C. e Soares, L. H. (2002) "Protocolização Digital no Sistema de Peticionamento Eletrônico". Disponível em <http://www.bry.com.br/downloads/artigo/Protocolização Digital no Sistema de Peticionamento Eletronico.pdf>, último acesso em Novembro/2003.

Pinkas, D., Ross, J., Pope, N. (2001) “Eletronic Signature Formats for long term eletronic signatures”, RFC 3126. Disponível em <http://www.ietf.org/rfc/rfc3126.txt>, último acesso em Março/2004.

Adams, C., Cain, P., Pinkas, D. e Zuccherato, R. (2001) "Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)", RFC 3161. Disponível em <http://www.ietf.org/rfc/rfc3161.txt>, último acesso em Dezembro/2003.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (2000) "Padrões de Projeto", Editora Bookman, Porto Alegre.

Housley, R. (1999) "Cryptographic Message Syntax (CMS)", RFC 2630. Disponível em <http://www.ietf.org/rfc/rfc2630.txt>, último acesso em Novembro/2003.

Massias, H. e Quisquater, J.-J. (1997) "Time and cryptography". Disponível em <http://citeseer.nj.nec.com/massias97time.html>, último acesso em Novembro/2003.

Haber, S. e Stornetta, W.S. (1991) "How to Time-Stamp a Digital Document". Disponível em <http://www.surety.com/docs/1sttime.pdf>, último acesso em Novembro/2003.