© 2005 by Adam J. Lee. All rights reserved.

TRAUST: A TRUST NEGOTIATION BASED AUTHORIZATION SERVICE FOR OPEN SYSTEMS

BY

ADAM J. LEE

B.S., Cornell University, 2003

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Abstract

Open computing systems aim to enable effective resource and information sharing between *authorized* users in multiple security domains. Making access control decisions in these systems is a difficult task, as a potentially unbounded number of users and resources exist in an environment with few guarantees regarding established trust relationships. Current access control mechanisms fail to adequately meet the needs of these systems due to design assumptions that are incompatible with the trust model used in open systems.

In this thesis we present Traust, a general purpose authorization service that uses trust negotiation to meet the needs of large-scale open systems. Traust allows users to establish relationships with previously unknown resource providers at runtime and negotiate for access to resources of interest. The Traust service can be integrated tightly with newer, trust aware resources or used to broker access to legacy services that do not natively support Traust. In either mode of operation, access policy maintenance overheads in the Traust system scale independently of the number of users in the system. We discuss our implementation of Traust and its performance, our experiences using Traust to broker access to legacy resources, and our progress developing a Traust-aware version of the GridFTP protocol. We also discuss Traust's resilience against attack.

To my family.

Acknowledgments

I would like to thank my advisor, Professor Marianne Winslett, for her insight and guidance throughout the duration of my studies at the University of Illinois. Jim Basney and Von Welch were invaluable sources of information and provided many useful comments during the design of the Traust system; their help was greatly appreciated. I would also like to thank Steph and the "kids" for putting up with me during times of stress and providing me with a means of escape from that which could have easily become all-consuming. Last, but certainly not least, I must thank every member of my family for their unconditional love and support; without them I would not have become the person that I am today.

Table of Contents

List of Figures . . . . **viii**

Chapter 1 Introduction **1**

Chapter 2 Design Requirements **4**

Chapter 3 Related Work **6**
3.1 Existing Solutions 6
3.2 Trust Negotiation 7

Chapter 4 Traust System Architecture **9**
4.1 Traust Servers 9
4.2 Traust Clients 9
4.3 Trust Negotiation Agents 10
4.4 Access Token Repository 10
4.5 Resources 11

Chapter 5 Protocol Overview **12**
5.1 Session Security 12
5.2 Message Types 12
5.3 Resource Access Protocol 14

Chapter 6 Implementation **17**
6.1 Implementation Details 17
6.2 Usage with Legacy Resources 18
6.3 Traust-Aware Resources 19
6.4 Performance 20

Chapter 7 Discussion **22**
7.1 Requirements Revisited 22
7.2 Security Evaluation 23
7.2.1 Session Security 23
7.2.2 Ensuring Valid Attribute Certificates 24
7.2.3 Single Point of Attack 25
7.2.4 Denial of Service 27
7.2.5 User Accountability 28

Chapter 8 Conclusions & Future Work **29**

Appendix A The Traust Protocol Specification **31**
A.1 Core Protocol 32
A.1.1 TCP Port Number 32
A.1.2 Traust Session Transport Layer 32
A.1.3 Basic Message Format 32
A.2 Currently Supported Messages 33
A.2.1 Get Info Command (COMMAND=0) 34
A.2.2 Initiate Trust Negotiation (COMMAND=1) 34
A.2.3 End Trust Negotiation (COMMAND=2) 35
A.2.4 Resource Request (COMMAND=3) 35
A.3 Supported Credential Types 37
A.3.1 Username/Password Credentials (TYPE=0) 37
A.3.2 X.509 Proxy Certificates (TYPE=1) 37
A.4 The Traust Resource Access Protocol 38
A.4.1 Valid Protocol Executions 38
A.4.2 Handling Error Conditions 39
A.5 A Sample Traust Session 39
A.6 A Sample PEM Encoded X.509 Proxy Certificate 41

Appendix B Traust Extensions for GridFTP **43**
B.1 The `TRAUST` Command 43
B.2 Allowing Permission Changes During a GridFTP Session 44
B.3 A Sample Traust-enabled GridFTP Session 46

References **48**

List of Figures

4.1 Traust system architecture 10

5.1 The Traust resource access protocol 14
5.2 A regular expression describing successful executions of the resource access protocol 16

6.1 The graphical Traust client 18

A.1 Traust protocol overview 32
A.2 A regular expression describing valid Traust interactions 38

Chapter 1

Introduction

Due to recent Internet trends—including peer-to-peer networks, grid computing, and corporations restructuring as virtual organizations—large-scale open systems are becoming ever more popular. Making intelligent access control decisions in these systems is a difficult task, as a potentially unbounded number of users and resources exist in an environment with few guarantees regarding pre-existing trust relationships. Traditional access control systems fail to work in these systems either because they cannot scale to such a large user base or make unrealistic assumptions about existing trust relationships in the system. As open systems continue to gain popularity, it is critical that the access control problem be addressed.

Trust negotiation is an active area of research aiming to help solve the problems surrounding authorization in open systems. In trust negotiation, access control decisions are made based on the attributes of the entity requesting access to a particular resource, rather than his or her identity. To determine whether an entity should have access to a resource, the entity and resource provider conduct a bilateral and iterative exchange of policies and credentials (used to certify attributes) to incrementally establish trust in one another.

To date, work in trust negotiation has focused primarily on the development of languages and strategies for trust negotiation (from at least eight research groups [BS04, BFS04, BS00, HMM+00, HHJS04, KM04, LM03, WWJ04, WSJ00, YWS03]) and the embedding of trust negotiation into commonly used protocols [HJM+02]. These research efforts have shown that the flexible nature of trust negotiation makes it a viable solution to the problem of authorization in open systems. If software engineers could easily redesign all major application protocols to support trust negotiation natively, the problem of making authorization decisions in open systems would be solved.

Unfortunately, redesigning and restandardizing existing protocols is a time-consuming process. To address this problem, we propose Traust, a stand-alone authorization service that allows for the adoption of trust negotiation in a modular, incremental, and grassroots manner, providing access to a wide range of resources without requiring widespread software or protocol upgrades.

In our approach, a collection of Traust servers act as brokers for the security tokens needed to gain access to the resources located in a given security domain. The format of these tokens is not restricted by Traust; tokens can be of any format, including ⟨username, password⟩ pairs, Kerberos tickets, SAML assertions, and X.509 certificates. Clients contact Traust servers and negotiate for access tokens for logical or physical resources including network servers, RPC methods, and organization-wide roles. The Traust service also provides clients in the system with an opportunity to establish trust in the service prior to the disclosure of their (potentially sensitive) resource access requests.

To the best of our knowledge, the Traust service represents the first general-purpose authorization system designed explicitly to meet the needs of large-scale open systems. In particular, the Traust system:

- allows clients to establish bilateral trust with previously-unknown resource providers on-the-fly and negotiate for access to new system resources at runtime;
- integrates transparently with newer, trust-aware resources while still maintaining compatibility with and allowing increased access to legacy resources;
- can broker access tokens in any format for any size security domain, ranging from single hosts (*e.g.*, in peer-to-peer systems) to entire organizations;
- has maintenance overheads that scale independently of the number of users in the system and the rates at which users join and leave the system.

The rest of this thesis is organized as follows. In Chapter 2 we highlight the defining characteristics of open systems and derive several important properties required of authorization systems designed for these environments. Chapter 3 examines related work in the area of access control relative to these requirements. Chapters 4 and 5 present the details of the Traust system architecture and resource access protocol, respectively. In Chapter 6 we describe our implementation of

the Traust system and its performance, discuss our experiences using Traust to broker access to existing legacy services, and comment on our progress developing a Traust-aware GridFTP client and server. In Chapter 7 we discuss how Traust meets the requirements identified in Chapter 2 and present a security analysis of the Traust system. We conclude and discuss our directions for future work in Chapter 8.

Chapter 2

Design Requirements

In large-scale open systems, resource providers choose to allow their resources to be shared across organizational boundaries in order to enable a greater number of authorized users to access these resources. Due to the large number of users in these environments, we cannot assume that resource providers will know the identities of all authorized clients that might possibly wish to access their resources *a priori*. In addition, we cannot assume that clients will know the set of resource providers that they might wish to interact with prior to the start of these interactions. Given that resource providers often wish to allow as many authorized users as possible to access these resources, we can immediately recognize four important requirements that must be satisfied by any authorization system designed for use in open computing systems.

Bilateral trust establishment To enable effective resource sharing, we cannot require pre-existing trust relationships between clients and resource providers; it is important to allow these entities to establish trust relationships with one another at runtime.

Runtime access policy discovery In large-scale open systems, clients cannot be expected to know the access polices protecting resources of interest *a priori*. Authorization systems used in these environments should allow clients to discover these policies as they are needed.

Preservation of privacy To protect clients and resource providers from malicious entities, their interactions should reveal as little information as possible. Clients and service providers should have some ability to control their disclosure of sensitive information, including their objectives, policies, identities, or attribute information.

Scalability Authorization systems used in open computing systems should be scalable both in terms of maintenance overhead and size of protection domain. Access policies should scale well in spite of a potentially unbounded number of users joining and leaving the system, while still maintaining an appropriate level of expressiveness. To accommodate the heterogeneity of these systems, the service should be light-weight enough for a single user (*e.g.*, a peer-to-peer client) to deploy on her local machine, yet robust enough to meet the demands of a large security domain.

In addition to these four requirements, it is important to include another, more practical, property:

Application Support Incorporating a new authorization service into existing open systems should not require a complete redesign of deployed applications, protocols, or the network infrastructure. The authorization system should support tight interaction with trust-aware applications designed to leverage its features explicitly, but also remain accessible to clients who wish to access legacy applications.

We do not make the claim that the above list of requirements is complete, as completeness will always depend on the *specific* needs of a system's participants. However, a system embodying these five requirements would allow resource providers to ensure that their resources (*e.g.*, data, computational clusters, or other services) are available to as many *authorized* users as possible without compromising the security of the resource itself. Such a system would also enable users to maximize their productivity by gaining access to new resources at runtime.

Chapter 3

Related Work

In this chapter, we examine related work in access control systems relative to the requirements presented in Chapter 2, in order to highlight the limitations of using these existing solutions in large-scale open systems. We then discuss trust negotiation, its current uses in open systems, and its relationship to Traust.

3.1 Existing Solutions

In many ways, it is natural to make access control decisions based on an authenticated identity. As such, a large number of access control systems are based on this notion, including the username/password authentication used in protocols such as Telnet and FTP, RADIUS [RWRS00], Kerberos [KN93, NT94], SESAME [AV99], structured [HFPS99] and unstructured [Zim95] systems based on certified public keys, and federated identity systems [KNe03, lib05, mic05, PW03]. While these systems perform well in some types of environments, they do not meet the needs of open systems.

These systems necessarily require that a user be known to the system ahead of time, as without this pre-existing knowledge, the system cannot provision resources to accept access requests from the user. In addition, identity-based access control systems require that security administrators manage access policies or role memberships on a per-user basis, a practice that does not scale well in environments with a potentially unbounded number of users. Lastly, these systems require that users prove their identity to the system without requiring that the system prove to the user that it is trustworthy.

In capability-based access control systems, resources make access control decisions based on authorization tokens presented by users of the system at the time of their request, rather than user identities. The concept of capabilities has long been used in operating systems [CJ75, Lev84, SJR86, SSF99] and distributed systems [BFK99, BFL96, TMvR86] security. Though these systems do not have the same limitations as identity-based systems, capability-based access control systems also do not meet the needs of open systems.

These systems do not support the bilateral trust establishment requirement discussed in Chapter 2, as they assume users have existing relationships with the system in order to acquire access capabilities. In addition, it is assumed that resource access policies are known *a priori*, enabling users to decide which capabilities should be presented in order to access a given resource. As was the case with the identity-based solutions discussed above, these authorization mechanisms are also unilateral, that is, users implicitly trust resources; this trust is not guaranteed to exist in open systems.

3.2 Trust Negotiation

Trust negotiation is a technique that has been proposed to address the shortcomings discussed above. In trust negotiation, the access policy for a resource is written as a declarative specification of the attributes that an authorized entity must possess in order to gain access to the resource. In these systems, credentials are also considered resources, so sensitive credentials can be protected by release policies of their own. In this way, an access request leads to a bilateral and iterative disclosure of credentials and policies between the user and resource provider. During this process, trust is established incrementally as more and more sensitive credentials are exchanged. There is a wide body of research regarding languages, strategies, and architectures for trust negotiation [BS04, BFS04, BS00, HMM+00, HHJS04, HJM+02, KM04, LM03, WWJ04, WSJ00, YWS03].

As an example trust negotiation, consider the case in which a user, Alice, wishes to access a service provided by Bob. After Alice requests access to Bob's service, Bob discloses the access policy for his service, which states that in order to use Bob's service, Alice must disclose her digital Visa credential. To protect herself from identity theft, Alice is only willing to disclose her Visa credential to members of the Better Business Bureau (BBB), so rather than disclose her Visa

credential, Alice sends Bob this release policy. Bob is in fact a member of the BBB and is willing disclose this credential to anyone. This satisfies Alice, who discloses her digital Visa credential to Bob and is then granted access to Bob's service. It is clear that these types of exchanges address the shortcomings discussed above by allowing mutually distrustful parties to gain trust in one another incrementally and bilaterally in a privacy-preserving manner.

To date, implementations of trust negotiation have been successfully embedded in commonly used applications and protocols (*e.g.*, [isr05]). While this clearly demonstrates the utility of trust negotiation, revising the protocols needed to access every resource used in open computing systems would be a daunting task. Rather than take this approach, Traust was designed to act as a stand-alone authorization service that allows users to use trust negotiation to establish trust relationships with new resource providers at runtime and to negotiate for access rights to any number of resources which may or may not support trust negotiation natively.

In the rest of this thesis, we describe the architecture and communications protocol used in the Traust authorization system and how Traust leverages the strengths of trust negotiation to act as a general-purpose authorization system that meets the needs of open systems.

Chapter 4

Traust System Architecture

Figure 4.1 illustrates the Traust system architecture. In the remainder of this chapter, we describe each component in greater detail.

4.1 Traust Servers

In our system, Traust servers act as brokers for the access rights to a set of resources in their security domain. A Traust server contains a protocol interpreter that is responsible for carrying out the steps of the Traust resource access protocol (see Section 5.3) and has some means of interacting with its Trust Negotiation Agent (or Agents). Each Traust server also maintains a repository of access tokens used to grant access to the resources that it protects; these tokens are issued to authorized users who negotiate for access to the resources protected by the Traust server.

4.2 Traust Clients

A Traust client is a process designed to acquire access tokens for resources of interest to its owner. Like a Traust server, a Traust client also contains a protocol interpreter and a means of contacting its Trust Negotiation Agent (or Agents). For systems in which resource requests could themselves be considered sensitive (*e.g.*, requests to access classified data), Traust clients have a means of determining the sensitivity classification (or classifications) of a request to determine its corresponding release policy (or policies). For maximum flexibility, a Traust client can be accessed either directly by a user or on the user's behalf by a Traust-aware application. Further information regarding these two modes of operation is presented in Chapter 6.

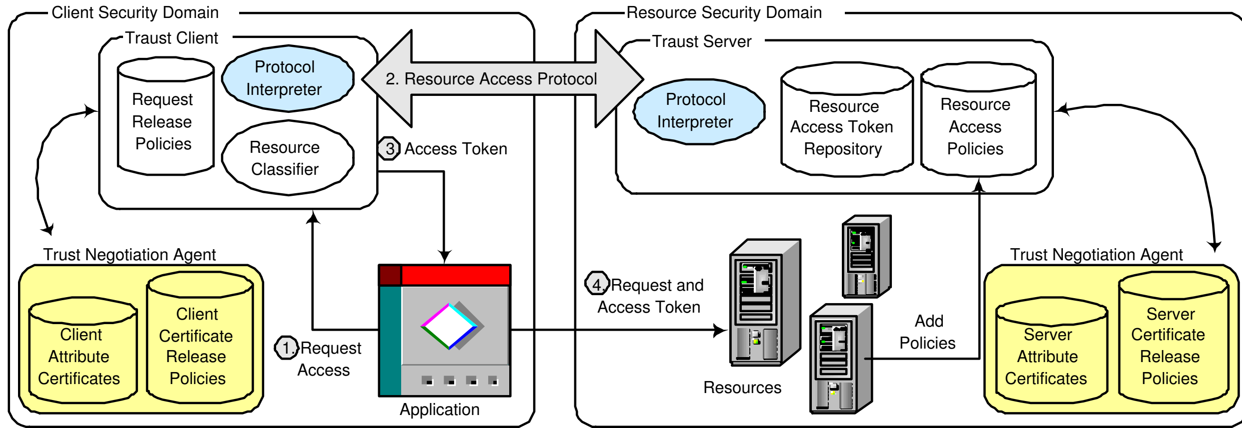

Figure 4.1: Traust system architecture

4.3 Trust Negotiation Agents

Both the Traust client and Traust server require access to one or more Trust Negotiation Agents. A Trust Negotiation Agent is responsible for understanding the protocol used for trust negotiation and carrying out trust negotiation sessions on behalf of the client or server processes that own it. In addition, a Trust Negotiation Agent manages its owners' attribute certificates and their corresponding release policies.

Logically, a Trust Negotiation Agent is part of the Traust client and server applications, though it need not run on the same physical machine and can be a shared resource for all of a user's processes. This allows for increased flexibility, as the overheads of running the agent can be shared across multiple processes. Allowing a Traust server to access multiple Trust Negotiation Agents also permits load-balancing during periods of high traffic.

4.4 Access Token Repository

Each Traust server maintains a repository of access tokens that can be used to access the services that it protects. This repository is not a repository in the traditional sense, which implies that it contains a static collection of tokens. Rather, the repository may contain static tokens, but may also contain instructions on obtaining or creating new tokens at runtime. For instance, the repository may create new local accounts used to access resources that it protects, acquire Kerberos tickets, generate SAML assertions, or be delegated proxy certificates from a MyProxy [NTW01]

server.

4.5 Resources

Resources are the logical and physical objects that Traust servers broker the access rights to. Some examples of resources include networks, individual machines, services (*e.g.*, web sites or file servers), RPC methods, web services, or organization-wide role memberships.

Chapter 5

Protocol Overview

In this chapter, we present an overview of the communication protocol used in the Traust system and discuss the ways in which Traust components interact during the execution of this protocol. We focus on our attention on message semantics and permissible sequences of messages; the full details of the Traust protocol, including message contents and formats, can be found in Appendix A, the Traust Protocol Specification.

5.1 Session Security

All communications between a client and Traust server occur inside of a TLS [DA99] tunnel used to provide confidentiality and integrity for the session. The tunnel itself is not used to provide any notion of authentication or authorization; one or more trust negotiation sessions are used for this purpose. We discuss these trust negotiation sessions in greater detail in Section 5.3.

5.2 Message Types

Messages in the Traust protocol can be divided into two categories: functional messages and trust establishment messages. The functional messages and their replies allow a Traust client to make requests of a Traust server and be provided with information in return. The current version of the Traust protocol supports two types of functional messages: *Get Information* and *Resource Request*.

Get Information (GI) The GI message allows Traust clients to request public meta-data regarding a Traust server with which they have an established connection. The server's response to

this message may include information such as software and protocol versions, a "message of the day," administrative points of contact, or other site-specific information. A GI request may only be sent by the client at the start of a Traust session.

Resource Request (RR) The RR message and its corresponding response embody the main functionality of the Traust service. RR messages contain a URI [BLFM05] and a series of optional ⟨attribute, value⟩ pairs describing a resource that the Traust client wishes to acquire access tokens for. This naming system is flexible enough to specify a wide variety of resources, including entire networks, enterprise-wide roles, or individual hosts, services, or method calls. The server response to this message contains either a failure notification or a collection of access tokens that can be used to access the requested resource.

To control the flow of sensitive information between clients and servers in the system, Traust supports three trust establishment message types: *Initiate Trust Negotiation*, *Trust Negotiation*, and *End Trust Negotiation*.

Initiate Trust Negotiation (ITN) The ITN message serves as a flag to indicate that a new trust negotiation session is about to begin. After receiving an ITN message, the Traust client (or server) will forward subsequent messages to one of its associated Trust Negotiation Agent processes for processing until an *End Trust Negotiation* message is received.

Trust Negotiation (TN) TN messages are used to encapsulate a trust negotiation session carried out between the Trust Negotiation Agent processes of the Traust client and Traust server. The policies and credentials exchanged between parties in the Traust system are encoded in the body of these messages. Several rounds of TN messages may be required for the initiating party to determine whether an acceptable level of trust has been gained in the responding party.

End Trust Negotiation (ETN) The ETN message serves as a flag to indicate that a trust negotiation session has just completed. Upon receiving an ETN message, the receiver will cease forwarding subsequent messages to their Trust Negotiation Agent.

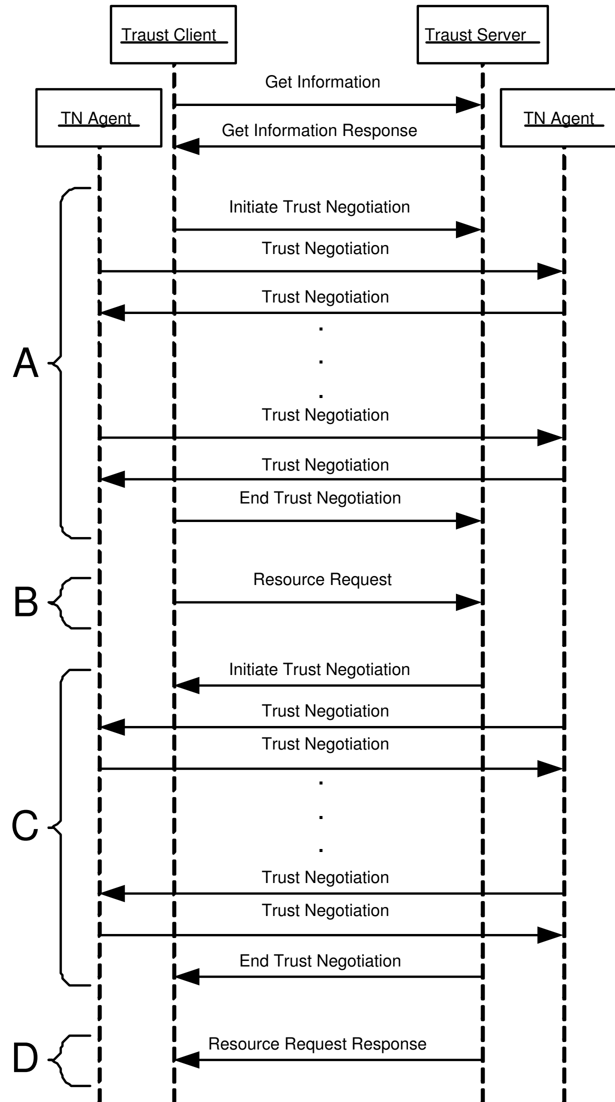

Figure 5.1: The Traust resource access protocol

Next, we describe how these messages are ordered to form the communication protocol used in the Traust system.

5.3 Resource Access Protocol

The Traust resource access protocol takes place in five stages: local classification, server trust establishment, request disclosure, client trust establishment, and response.

Prior to establishing a connection to a Traust server, the user's Traust client is provided with

the description of a resource that the user wishes to negotiate for access to. This description may be generated explicitly by the human user (*e.g.*, after reading a web page describing how to access a legacy service protected by a Traust server) or generated on-the-fly by a client application interacting with a Traust-aware resource. During the *local classification* stage, this resource description is examined using a local content classifier to determine its sensitivity classification or classifications. The Traust client then maps these sensitivity classifications into release policies which will be used in the server trust establishment phase.

During *server trust establishment*, indicated by the label 'A' in Figure 5.1, the Traust client initiates zero or more content-triggered trust negotiation sessions [HHJS04] with the Traust server—one for each release policy discovered during local classification. Alternatively, the client could initiate a single negotiation using the conjunction of these release policies. This process determines whether the Traust server is trustworthy enough to receive the resource request issued to the Traust client and prevents inadvertent disclosure of sensitive requests to unauthorized Traust servers.

Each trust negotiation session is initiated by the client sending an Initiate Trust Negotiation message to the server. The client's Trust Negotiation Agent then conducts an iterative exchange of Trust Negotiation messages with the server's Trust Negotiation Agent, reporting the results of this negotiation back to the Traust client. The Traust client terminates this phase by sending an End Trust Negotiation message to the Traust server. Should the client fail to establish trust in the server during any of these trust negotiation sessions, the Traust client closes its connection with the server and reports a failure to the user.

If the Traust client establishes trust in the server during each trust negotiation initiated by the Traust client, the Traust session enters the *resource disclosure* stage. At this point, the Traust client sends a Resource Request message to the Traust server describing the resource that the user wishes to access. This disclosure is indicated by the label 'B' in Figure 5.1.

Upon receiving the Traust client's Resource Request message, the Traust server examines it to determine the access policy that protects the requested resource. The server then begins the *client trust establishment* phase, indicated by the label 'C' in Figure 5.1, by sending an Initiate Trust Negotiation message to the client. The Traust server's Trust Negotiation Agent then carries out a negotiation with the client's Trust Negotiation Agent via an iterative exchange of Trust Negotiation

$$(GI_C GI_S)?(ITN_C (TN)^* ETN_C)^* RR_C (ITN_S (TN)^* ETN_S)? RR_S$$

Figure 5.2: A regular expression describing successful executions of the resource access protocol

messages. When the negotiation is over, the Traust server sends an End Trust Negotiation message to the Traust client to indicate this fact.

In the *response* phase, indicated by the label 'D' in Figure 5.1, the Traust server indicates the status of the resource access protocol. If the server failed to establish trust in the client, the client is sent a failure notification in the Resource Request response message. If the Traust server did establish trust in the client, however, it obtains the access tokens needed for the client to access the requested resource and passes these tokens to the client in the Resource Request response message. Obtaining an access token could be as simple as looking up a static token, or could involve generating a new local account or interacting with a Kerberos, MyProxy, or CAS [PWF+02] server to obtain the needed access tokens.

At a high level, the Traust resource access protocol maps users' attributes into access tokens that are meaningful in the local security domain of the resource that is to be accessed. Figure 5.2 is a regular expression describing successful executions of the Traust resource access protocol. The message abbreviations are those used in Section 5.2 and the subscripts indicate whether a particular message was sent by the client (C) or server (S). Note that we include the optional transmission of a Get Information message and its response, though it was not discussed in this section. Any sequence of messages not described by this expression is considered invalid; a correctly functioning participant in the resource access protocol should immediately close any connection upon which an invalid execution has been detected.

Chapter 6

Implementation

In this chapter, we discuss our implementation of the Traust system. In addition, we discuss our experiences using Traust to broker access to legacy resources (*e.g.*, password-protected web sites), comment on our progress developing a Traust-aware GridFTP client and server, and address the performance of our implementation.

6.1 Implementation Details

We have developed a prototype implementation of the Traust service using the Java programming language. We provide a client API that can be embedded into applications that wish to interact directly with a Traust server. In addition, we have developed both command line and graphical Traust clients which allow human users to interact with a Traust server to request access tokens for legacy services whose clients do not natively support Traust. We also provide an extensible resource classification API which allows users to develop custom request sensitivity classifiers. Because defining these types of classifiers can be difficult, a user's organization (*e.g.*, their employer or credential issuer) is likely to supply them with the classifiers for sensitive requests. In our implementation, a resource classifier based on substring matching is used by default.

Our server implementation provides an extensible API which can be used to interface with a wide variety of access token repositories. We have developed a simple, yet flexible, token repository that allows the server to obtain the tokens needed to access a given resource by either (1) accessing tokens stored directly in the repository, (2) referencing files located on the local file system, or (3) interfacing with external processes. We have used the latter mechanism to generate one-time-use

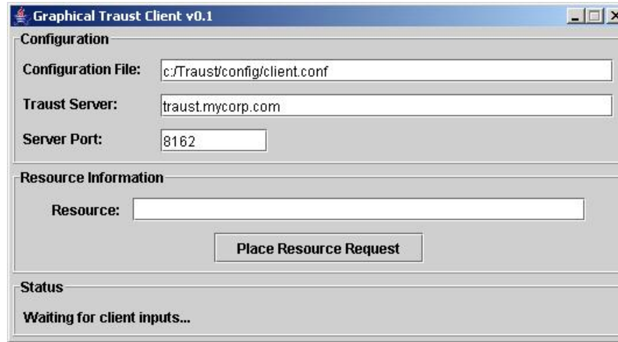

Figure 6.1: The graphical Traust client

passwords, delegate X.509 proxy certificates, and create temporary local accounts.

Both the client and the server currently utilize Trust Negotiation Agents based on the TrustBuilder framework for trust negotiation [WYS+02]. TrustBuilder currently supports the use of X.509 attribute certificates for credentials and the IBM Trust Policy Language [HMM+00] for access policy specification, with future support for other policy languages and credential types. TrustBuilder has been successfully integrated with a number of protocols and applications [HHJS04, HJM+02, RZN+05], making it a good choice for use in the Traust system.

6.2 Usage with Legacy Resources

In some computing environments, it is considered acceptable to require that clients manually acquire resource access tokens prior to using networked services. For instance, at many universities, users wishing to access student records must first acquire a Kerberos ticket using a stand-alone client application (*e.g.*, `kinit`). For these environments, we have developed command line and graphical Traust clients that allow users to manually request access tokens for legacy services that do not natively support Traust interaction. Figure 6.1 shows a screen shot of our graphical Traust client.

As an example of how Traust might be used in this type of environment, consider the case of a rescue dog handler who hears a newscast about a building that has collapsed and wishes to help in the recovery effort. The newscast gives the URL of a web-based information portal that will be used to coordinate the recovery effort. The user browses to this web site and is presented with a login form and resource descriptor to pass into his Traust client that will allow him to negotiate for

a temporary login and password to the portal. The Traust interaction allows the client to establish trust in the server (*e.g.*, that the server is a state-sponsored disaster response coordinator, not a hoax) and allows the server to verify the user's credentials (*e.g.*, that he is a certified rescue dog handler with up-to-date vaccinations, not a news reporter looking for a hot story). The Traust server then returns a temporary login and password for the web site, which the client application displays to the user.

We have built a testbed information portal for this application and successfully used Traust to allow previously unknown, but authorized, users to gain access to the information contained within. In addition, Traust has been used in a similar fashion to issue X.509 proxy certificates that control access to a file server.

6.3 Traust-Aware Resources

In addition to using Traust to control access to legacy resources, we wish to allow for the development of services that support Traust natively. These applications can embed Traust interactions in their access protocols, allowing users' client applications to carry out any necessary Traust interactions without requiring the user to initiate this process. As an example of how to permit this form of tight interaction with Traust, we have proposed two modifications to GridFTP, a secure mass data transfer protocol used heavily in the context of grid computing. In this section, we overview these proposed modifications; the technical details of the modifications are discussed in greater detail in Appendix B.

The first of our modifications allows a user's GridFTP client application to query the GridFTP server (prior to login) for the name of the Traust server that brokers access to this server. The client application can then execute the Traust resource access protocol with this server without user intervention. If the Traust server authorizes the client to access the GridFTP server, the Traust server will issue the client an X.509 proxy certificate that the client can use to log into the the GridFTP server using the existing Grid Security Infrastructure [WSF+03].

Our second modification defines a simple syntax for access "hints" that can be provided by a GridFTP server. In the event that a command issued by a client fails due to insufficient access rights, the server can embed an access hint in the error message returned to the client application.

These hints identify a Traust server to contact, and a corresponding resource request. A Traust-aware GridFTP client may, without user intervention, execute the resource access protocol with this Traust server in order to obtain a new a new X.509 proxy certificate. This certificate can be used to re-authenticate to the GridFTP server and successfully retry the previous request without terminating the current session. This allows GridFTP servers to enforce the principle of least privilege [SS75], as clients' access rights can be changed as they perform different operations on the server. We are currently modifying the GridFTP server distributed with the Globus toolkit[1] v3.9.5 to incorporate these additions.

6.4 Performance

We now comment briefly on the performance of our implementation in two representative usage scenarios. In the first scenario, the client releases its resource request to the Traust server without requiring a trust negotiation. The Traust server then initiates a single-round trust negotiation with the client in which the client demonstrates proof of ownership of one attribute certificate. This case is indicative of Traust interactions that might be seen in corporate environments where users are asked to show their digital employee ID card or role certificate to gain access to a particular resource. We ran this scenario across our department's network at midday and found that it executed in 2.77 seconds, on average over 10 runs. The two major components of this time are connection establishment (1.12 seconds) and configuring the client Trust Negotiation Agent and carrying out the trust negotiation (1.33 seconds).

The second scenario uses the disaster response information portal discussed in Section 6.2. In this scenario, the client is only willing to disclose his access request to Traust servers that can prove that they are operated by a state-sponsored disaster response coordinator, and uses the server trust establishment phase of the resource access protocol to enforce this. The Traust server is able to prove ownership of an attribute credential indicating this fact, which satisfies the client, who then discloses his request for access to the information portal.

At this point, the server requests that the client prove that he is a certified rescue dog handler, is over the age of 18 (by showing a state-issued driver's license), has a recent tetanus vaccination

[1] See `http://www.globus.org/`.

(the record of which is issued by a state-certified board of health), and that his dog has a recent rabies vaccination (the record of which is issued by a state-certified county). In response to this request, the client demonstrates proof of ownership of his rescue dog handler certificate and discloses the release policies protecting his driver's license and both vaccination records. The release policy protecting his driver's license requires that the server disclose a privacy policy issued by an accrediting organization, while the release policy protecting both vaccination records requires that the server prove that it is operated by a department of some U.S. state. The Traust server is willing to disclose this information, at which point the client sends over the remaining credentials required by the server. In all, these two trust negotiations took place over three rounds and involved the disclosure of nine credentials (including supporting credentials for certification chains). On average, this scenario executed in 4.04 seconds over our department's network at midday. The main components of this time are connection establishment (1.12 seconds), configuring the client Trust Negotiation Agent and carrying out the first negotiation (1.58 seconds), and the second trust negotiation (1.34 seconds).

The Traust resource access protocol takes longer than using a more traditional means of acquiring resource access tokens (*e.g.*, obtaining a Kerberos ticket), but the extra amount of time is reasonable. As the access tokens issued by a Traust server will be valid for some interval, the acquisition cost can be amortized over multiple resource interactions. Additionally, the client used in these tests created a new Trust Negotiation Agent at each invocation, a process which takes .93 seconds on average; configuring the client to use a stand-alone Trust Negotiation Agent would eliminate this overhead. Further, to the best of our knowledge, the TrustBuilder system has not undergone a performance evaluation study, so there exists opportunity for optimization within this system. The benefits of allowing previously unknown users to negotiate for access to resources outweigh the modest cost of the negotiation.

Chapter 7

Discussion

In this chapter, we discuss the security properties of the Traust system. First, we describe the ways in which Traust meets the needs of large-scale open systems by addressing each of the requirements presented in Chapter 2. We then present an informal security analysis of the Traust system and address possible attacks against the system.

7.1 Requirements Revisited

Chapter 2 introduced five requirements for authorization systems to be used in open systems: bilateral trust establishment, runtime access policy discovery, preservation of privacy, scalability, and application support. The Traust system architecture and resource access protocol were designed to address these goals from the start. Bilateral trust establishment and runtime access policy discovery are attained through the use of trust negotiation in the server and client trust establishment stages of the resource access protocol. To help preserve privacy, these negotiations can leverage negotiation strategies that limit the disclosure of sensitive credentials. In environments where some requests themselves could be considered private, clients may also enforce their own request release policies. Traust's use of trust negotiation implies that access policies are specified in terms of the attributes that an authorized user must possess, thus these policies scale independently of the number of users joining and leaving the system. The performance of the Traust service prototype is reasonable (roughly 4 seconds on average for a complex interaction). Lastly, Traust integrates with both legacy and Traust-aware resources.

7.2 Security Evaluation

Though Traust adequately addresses the five requirements discussed in Chapter 2, these properties say very little about the security of the system. In this section, we discuss the security properties of the Traust system and address several potential attacks against Traust.

7.2.1 Session Security

In the Traust system, session security is provided through the use of the TLS protocol. In [DA99], the authors present a security analysis of the TLS protocol under the assumption of an active attacker [DY83] with the ability to intercept, modify, delete, and replay messages sent over the communication channel. In the case that the public key of one party in the protocol is authenticated, the authors show that the TLS channel is secure against man-in-the-middle attacks, thus the two parties can be assured of the confidentiality and integrity of the messages transmitted using TLS. In situations where a Traust server is run by an organization such as a university, research center, or corporation, the server can be issued a certified public key much in the same way that World Wide Web servers are issued certified keys today. In these cases, the security evaluation presented in [DA99] applies to the session security of Traust.

In environments where neither the client nor the server has a certified public key (*e.g.*, peer-to-peer networks), the TLS protocol is vulnerable to a man-in-the-middle attack during session establishment. The implication of this attack is that an unauthorized third party can read and alter messages sent through the TLS tunnel, unknown to the client and server. The SSH [YL05] protocol is subject to the same such attack, as it is rarely the case that SSH servers have certified public keys. In SSH, the threat of this attack is usually mitigated by caching previously-used public keys. In this way, unless the man-in-the-middle attack occurs during the first connection between the client and server, it can be detected, as the cached public key of the legitimate server and the public key returned by the man-in-the-middle will not match. We argue that the threat of this attack can be reasonably mitigated in Traust by using the same practices as are used in SSH. As in SSH, however, highly sensitive non-certified public keys should be verified out-of-band to prevent this attack.

Given that it is possible to prevent man-in-the-middle attacks against the TLS protocol, we

argue that the security analysis presented in [DA99] ensures that messages exchanged during the Traust resource access protocol can be viewed only by their intended recipients. However, ensuring that the content of these messages (*e.g.*, attribute certificates exchanged during trust negotiation) actually belongs to the parties involved in the protocol requires extra attention.

7.2.2 Ensuring Valid Attribute Certificates

A naive implementation of the Trust Negotiation Agents used in the Traust system leaves parties open to an attack in which a malicious party can demonstrate "proof" of ownership for attributes that it does not possess by conducting interleaved executions of the resource access protocol with multiple parties. We now describe the attack in greater detail and present a solution that eliminates the possibility of this attack from the Traust system.

To illustrate this attack, consider the following example in which Alice is carrying out an execution of the Traust resource access protocol with Mallory. Alice wishes to access Mallory's Music Warehouse so that she can download songs to listen to during her commute to work. During the client trust establishment stage of the resource access protocol, Mallory requests that Alice submit her digital Visa card (presumably so that Mallory can bill Alice for the music that she downloads) and demonstrate proof of ownership. To protect herself from identity theft, Alice is only willing to disclose this credential to businesses who are members of the Better Business Bureau (BBB). Alice sends this release policy to Mallory along with a challenge for him to sign using the private portion of his BBB credential. Mallory is not a member of the BBB, though he wishes to trick Alice into believing that he is.

Mallory then opens a Traust connection to Bob's Books. Bob runs a Traust server to allow certain groups of users, such as school teachers, to negotiate for access to discount coupons for the books that he sells. During the server trust establishment phase of the resource access protocol, Mallory informs Bob that he is only willing to interact with members of the Better Business Bureau and submits Bob a challenge (Alice's challenge!) to sign with the private portion of his BBB credential. Bob is a member of the BBB and willingly signs the challenge, returning it along with the public portion of his BBB credential to Mallory. Mallory then closes his connection with Bob and forwards Bob's BBB credential and signed challenge to Alice, who is now convinced that

Mallory is a member of the BBB.

This attack is possible because a naive implementation of the Trust Negotiation Agent has no means of associating an attribute credential with a particular identity. In the Traust system, however, parties can form a loose notion of "session identity" by requiring a binding between attribute credentials and the public key used to establish the underlying TLS tunnel. To prevent the attack discussed above, we can require that parties not sign the challenge sent by the other negotiating party directly. Rather, the Trust Negotiation Agent first concatenates this challenge with a hash of the public key that was used to establish the TLS tunnel used in this interaction and signs the resulting bit-string. This prevents the surreptitious forwarding of attribute credentials as in the above example, as long as the actual owner of the attribute credential is trustworthy.

Note that the above solution does not prevent the collusion of malicious entities who wish to pool their resources to appear as a single, more privileged entity. Techniques such as hidden credentials and oblivious signature-based envelopes [HBSO03, LDB03] can be used to prevent collusion, though this system requires that parties know the identity used by the other negotiating party in order to obtain their identity-based public keys. Another possible solution involves the use of attribute-based encryption [SW05], though this requires that all attributes of interest to the negotiation be issued by the same authority. Unfortunately, this makes attribute-based encryption unappealing in the open system environment for which Traust was designed. Systems such as *idemix* [CH02] can also be used to prevent this type of collusion, though at the expense of embedding a master secret into each credential. The difficult nature of solving the collusion-resistance problem makes it an exciting area for future work.

7.2.3 Single Point of Attack

In addition to attacks on the Traust resource access protocol, we must also consider attacks on the Traust server itself. If a single Traust server brokers access tokens for a large number of resources, it will be an appealing target for attack, as a successful attacker could possibly gain access to a large number of resources by compromising a single Traust server. We now discuss several potential solutions to this problem.

Small protection domains

The Traust server that controls access to a given resource can be run on the same physical machine as the resource itself. In this case, a compromise of the machine that the Traust server is running on only grants the attacker the access tokens needed to access the single resource that the server was protecting. In addition, these tokens are of no value, as the attacker implicitly gains access to that resource by compromising the machine on which the resource is located.

This Traust server configuration model clearly prevents an attacker from gaining access to multiple resources by compromising a single node, and motivates the use of Traust in peer-to-peer systems. We next consider two arrangements of Traust servers for use in organizations wishing to maintain a more structured organizational model.

Hierarchical arrangement

Here we consider arranging the Traust servers protecting access to an organization's resources in a hierarchical fashion. In this model, the Traust servers at the upper level of the hierarchy broker access rights for a large number of low-sensitivity resources. As we proceed down the hierarchy, Traust servers broker access to fewer resources of higher sensitivity levels. The leaves of the hierarchy broker access to only a single, highly-sensitive resource.

In addition to the distribution of access rights discussed above, high-level Traust servers also know which lower-level servers broker access rights to other resources in the network. This knowledge allows higher-level servers to redirect client traffic to an appropriate lower-level server, making the organizational infrastructure easier to navigate for the client. We believe that this allows an adequate trade-off between the granularity at which Traust servers are deployed and the consequences of compromising one of these servers.

Secret sharing

For organizations not willing to pay the administrative costs associated with maintaining a hierarchy of Traust servers, we now discuss a protection strategy based on secret sharing [Bla79, Sha79]. In this model, an organization deploys multiple Traust servers, the exact number of which is determined by the organization's unique needs. A client wishing to access a given resource would

contact one of these servers and carry out the resource access protocol to attempt to gain access. Rather than being returned the token needed to access that resource, an authorized client is given a *share* of the access token and a list of other Traust servers. Upon completing the resource access protocol with a preset threshold, k, out of the n Traust servers, the client will have the ability to reconstruct the access token. In this model, an adversary needs to compromise multiple Traust servers to gain access to any resource.

In both the hierarchical and secret sharing Traust server deployment models, it is important that Traust server administrators keep several things in mind. To reduce the number of potential vulnerabilities that could be used to compromise a Traust server, only a minimal set of services should be configured. For instance, in [NTW01] the authors suggest that a MyProxy server be run on a "tightly secured host (*e.g.*, comparable to a Kerberos Domain Controller)"; we too feel that this minimum precaution should be taken to protect any Traust server brokering access to highly valuable resources. Additionally, if multiple Traust servers are to be deployed, their hardware and software configurations should be as heterogeneous as possible [ZVA+01, OS04] to prevent the threat of a single exploit compromising multiple Traust servers.

Note that it is possible to compose the hierarchical and secret sharing Traust server deployment models to meet the needs of a particular organization or resource provider. This flexibility allows administrators to effectively manage the trade-offs that exist between server maintenance overheads, the arrangement of servers within an organization, and the effects of compromising a Traust server.

7.2.4 Denial of Service

The potentially centralized nature of a Traust deployment along with the relatively heavyweight process of trust negotiation make Traust servers interesting targets for denial of service attacks. To prevent malicious users from extending the number of rounds required to reach a decision during a trust negotiation, Ryutov *et al.* integrate TrustBuilder with the GAA-API and leverage the GAA-API's mechanisms for responding to changes in system context [RZN+05]. They show how negotiation strategies and policies can be adapted at runtime in response to suspicion of

denial of service, thereby increasing system availability. Production implementations of the Trust Negotiation Agent used by the Traust server should use a mechanism such as this to help mitigate the threat of denial of service attacks based on trust negotiation.

Another, more standard, technique to help reduce the risk of denial of service attacks on Traust servers is replication. Since a Traust server can be decoupled from the resources to which it brokers access, high-traffic Traust servers can be replicated to prevent them from becoming bottlenecks. An organization that wishes to both allow their Traust servers to remain available during denial-of-service attacks and prevent the compromise of some number of Traust servers from allowing unauthorized access to their resources could use a k-of-n secret sharing scheme as described above; this ensures that as long as k Traust servers are available, authorized users can obtain the tokens necessary to access resources provided by the organization.

7.2.5 User Accountability

Another interesting problem related to systems based on trust negotiation involves the trade-offs that exist between openness and accountability. If a resource protected by an identity-based access control system is compromised or attacked, the user whose identity was used to attack the system can be held accountable, or at the very least investigated for clues as to who the real attacker might have been. Since trust negotiation is based on the exchange of attribute—rather than identity—information, this is not always possible.

As this problem is present in attribute-based trust negotiation systems in general, rather then Traust specifically, we did not address it during our system design. However, we see several possible solutions to this problem, including the use of third-party psuedonymity certifiers. These entities could be used to provide a legal means of recourse for resource providers whose resources were abused by a user with a pseudonym attribute issued by that certifier. It is not clear how such a system would function in truly open environments, though it is an interesting avenue for future research.

Chapter 8

Conclusions & Future Work

In this thesis, we addressed the problem of designing an authorization service that adequately meets the needs of large-scale open systems. We discussed how the unique requirements of these systems make the access control problem difficult to solve and showed how existing authorization and authentication systems do not meet the needs of open systems. To address these shortcomings, we proposed Traust, a general purpose authorization system based on trust negotiation.

Traust servers act as brokers which allow users in the system to negotiate for access rights to various resources in its protection domain. The resource access protocol used between clients and servers in the Traust system allows clients to establish trust in previously unknown Traust servers prior to any sensitive interaction. This enables clients to discover new resource providers at runtime and interact with them without inadvertently disclosing sensitive resource requests to malicious Traust servers. Servers also leverage the strengths of trust negotiation to establish trust in previously unknown clients prior to disclosing resource access tokens to them. All communications take place inside of a TLS tunnel to ensure confidentiality of the session.

Our implementation of the Traust service is written in Java and utilizes the TrustBuilder framework for trust negotiation. We discussed the components of our implementation and our experiences using Traust to protect access to legacy resources (*i.e.*, those resources that do not natively support interaction with Traust servers). We described the Traust-aware version of GridFTP, a data-transfer service used heavily in grid computing, which we are developing.

Lastly, we discuss the ways in which the Traust architecture and resource access protocol meet the needs of large-scale open systems in a secure manner. We show that Traust is resistant to man-in-the-middle attacks and surreptitious forwarding of attribute certificates, and discuss several

potential configurations of Traust servers that prevent the compromise of a single server from allowing an attacker to gain access to multiple resources and lessen the chances of a successful denial of service attack on Traust.

In the future, we plan to examine how Traust can be extended to support third-party negotiations for the purposes of obtaining new attribute certificates at runtime. We can imagine many situations in which a Trust Negotiation Agent is asked to show an attribute certificate that it does not possess, but could likely obtain. Extending Traust to support attribute certification hints would allow one Trust Negotiation Agent to tell another how to use Traust to locate attribute certificates that it does not currently possess. This however, would also allow malicious entities to waste the time of their negotiation partners, making this an interesting area to investigate. In addition, we plan to further explore how to securely manage the access tokens stored in the repository of a Traust server.

Appendix A

The Traust Protocol Specification

Open computing systems aim to enable effective resource and information sharing between *authorized* users in multiple security domains. Making access control decisions in these systems is a difficult task, as a potentially unbounded number of users and resources exist in an environment with few guarantees regarding established trust relationships. Current access control mechanisms fail to adequately meet the needs of these systems due to design assumptions that are incompatible with the trust model used in open systems.

Traust is a general-purpose authorization service that uses trust negotiation to meet the needs of large-scale open systems. Traust allows users to establish relationships with previously unknown resource providers at runtime and negotiate for access to resources of interest. The Traust service can be integrated tightly with newer, trust aware resources or used to broker access to legacy services that do not natively support Traust. Traust was designed to enhance the ease with which authorized users can access resources within large-scale heterogeneous open systems. For example, Traust can automate the dissemination of access credentials in Grid computing environments.

Figure A.1 presents an overview of the interactions that occur when a client wishes to access a resource in the protection domain of a Traust server. Step one of Figure A.1 shows the Traust client acting on behalf of the human user executing the resource access protocol with the Traust server which protects access to the resource of interest. This protocol provides a means for the user to dynamically discover the access policies protecting the resource and attempt to satisfy these policies via automated trust negotiation [WSJ00]. If the Traust client can satisfy the access policy, the Traust server will issue the client zero or more credentials needed to access that resource. The format of these credentials is not restricted by Traust; they can be of whatever format is

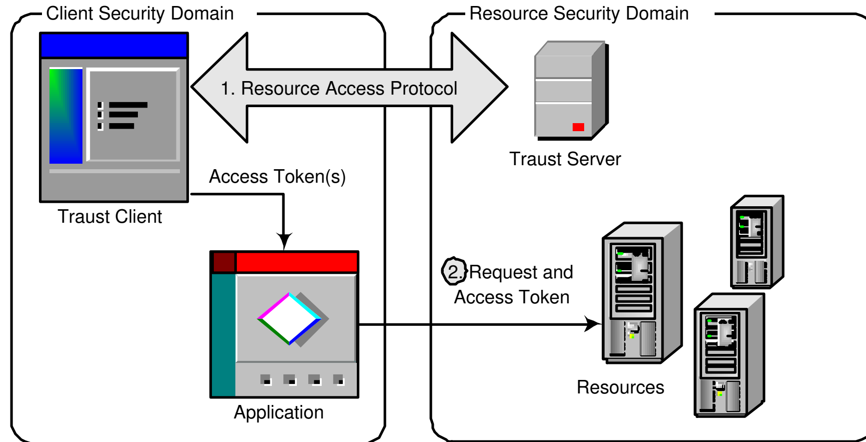

Figure A.1: Traust protocol overview

required by the resource in question (*e.g.*, SAML assertions, X.509 certificates, Kerberos tickets, username/password pairs, etc.). In this way, Traust maps users' attributes into access credentials that are locally meaningful to the resource to be accessed. In step two of Figure A.1, the client uses the newly obtained credentials to gain access to the resource. We now fully specify the protocol used by the Traust system.

A.1 Core Protocol

A.1.1 TCP Port Number

The Traust server can be configured to listen on any open TCP port. The default port is 8162.

A.1.2 Traust Session Transport Layer

Traust uses TLS as its underlying transport layer. As such, all communications discussed in this specification take place confidentially between the client and the service in a tamper-proof manner.

A.1.3 Basic Message Format

The basic message format used by Traust borrows greatly from that of the MyProxy system [NTW01]. Commands are written as lines of ASCII text, each of which terminates with the ASCII newline character, '\n' (0x0a). A command is terminated by two newline characters. Basic commands have

the form:

```
COMMAND=<integer>
<command_body_line1>
.
.
<command_body_lineN>
```

Responses to these commands can take one of two forms: a successful response or a failure response. Successful responses have the format:

```
COMMAND=<integer>
RESPONSE=0
<response_line1>
.
.
<response_lineN>
```

Failure responses have the following format:

```
COMMAND=<integer>
RESPONSE=1
ERROR=<text_line1>
.
.
ERROR=<text_lineN>
```

The `ERROR` lines should be concatenated together (separated by '\n' characters) by the client process before presentation to the user. After sending an `ERROR` response, the server must close the communication channel that it shares with the client immediately.

It is important to note that in both the successful and failure response cases the `<integer>` in the response `COMMAND` line is the same as the `<integer>` in the `COMMAND` line of the command that generated this response.

A.2 Currently Supported Messages

In this section, we discuss the commands supported by version 0.1 of Traust. Any Traust implementation that reports a version of 0.1 must support all of the following commands.

A.2.1 Get Info Command (COMMAND=0)

The Get Info command is used by a client to obtain various meta-information regarding the server. To request this information, the client sends the following command:

```
COMMAND=0
```

The server's response to this command is as follows:

```
COMMAND=0
RESPONSE=0
ATTRIB=(VERSION,0.1)
ATTRIB=(<attrib>,<value>)  \
.                           \ optional
.                           /
ATTRIB=(<attrib>,<value>)  /
```

In this response, the optional lines that begin with the `ATTRIB` prefix can be used to return various other information regarding the Traust server. Clients should be able to parse `ATTRIB=(CONTACT,(<name>,<email>))` and `ATTRIB=(MOTD,<msg>) ATTRIB`s. These `ATTRIB`s disclose the contact point for questions about this server and the server "message of the day," respectively. As with `ERROR` responses, multiple `MOTD` lines should be concatenated by the client (separated by '\n' characters) before presentation to the user. Servers may include site-specific `ATTRIB`s in their Get Info responses, though no assumptions should be made as to whether or not clients will know how to interpret these additional `ATTRIB`s.

A.2.2 Initiate Trust Negotiation (COMMAND=1)

The Initiate Trust Negotiation command is sent to indicate that the sending party wishes to conduct a trust negotiation with the other party. Clients may send this command prior to disclosing a resource request (see Section A.2.4) in order to establish trust in the Traust server before disclosing a sensitive request. Servers may send this command after receiving a Resource Request command from the client, to establish trust in the client prior to disclosing access credentials for the requested resource.

Upon the receipt of this command, the receiving party should use its trust negotiation agent to

parse all subsequent incoming communication until such time as it receives an End Trust Negotiation command. An Initiate Trust Negotiation command has the following format:

```
COMMAND=1
```

As this command is simply an indicator, the body is left blank. It should also be noted that there is no "response" version of this command, as it serves simply as an indicator to the other negotiating party and does not require any response on their behalf.

A.2.3 End Trust Negotiation (COMMAND=2)

The End Trust Negotiation command is sent after the trust negotiation initiated by an Initiate Trust Negotiation message has ended. At this point, the receiving party will cease to use its trust negotiation agent to parse incoming communication. An End Trust Negotiation command has the following format:

```
COMMAND=2
```

As with the Initiate Trust Negotiation command, the body of the End Trust Negotiation command is left blank and there is no "response" version of this command.

A.2.4 Resource Request (COMMAND=3)

The Resource Request command invokes the credential lookup functionality of the server. With this command, clients indicate a desire to access a given resource. A Resource Request has the following format:

```
COMMAND=3
<resource URI>
ATTRIB=(<attribute>,<value>)  // zero or more
```

The body of this command is a URI [BLFM05] identifying the resource that the client wishes to access. In many cases, this URI will take the form of a URL [BLMM94] specifying a networked service, though it may also take the form of a URN [Moa97] describing a more generic resource (such as a client's desire to activate a site-wide role). The URN syntax is also convenient for servers

wishing to attach opaque identifiers to Traust resources in hopes of hindering information gathering attacks. The optional `ATTRIB` lines allow a resource request to contain additional information, in the form of ⟨attribute,value⟩ pairs.

If the server receives an invalid resource request (for example, a request for a non-existent resource), it should return an error of the following form:

```
COMMAND=3
RESPONSE=1
ERROR=Invalid request
```

Upon receiving a valid Resource Request, the server may optionally send an Initiate Trust Negotiation command and conduct a trust negotiation session to determine whether or not the client is authorized to access the given resource. Should the trust negotiation session fail to allow the server to establish trust in the client, the server should respond with an error response to the Resource Request. This message has the format:

```
COMMAND=3
RESPONSE=1
ERROR=Client not authorized
```

If the trust negotiation session allows the server to establish trust in the client (or was not required), the server will return the access credential (or credentials) needed to access the requested resource by embedding them in a Resource Request Response. These messages have the following format:

```
COMMAND=3
RESPONSE=0
BEGIN_CREDENTIAL       \
TYPE=<integer>          \
<credential_data>        \
.                         > Zero or more
.                        /
<credential_data>       /
END_CREDENTIAL         /
```

As shown above, zero or more credentials can be disclosed in each Resource Request Response. Each such credential occurs between a matched pair of `BEGIN_CREDENTIAL` and `END_CREDENTIAL` lines. The `TYPE` line indicates (by way of an integer code) the type of credential contained in the

immediately following `CRED` lines. The format of the `CRED` lines is `TYPE` dependent and is discussed in Section A.3 of this document.

A.3 Supported Credential Types

This section describes the minimum set of credential types that must be supported by any Traust server whose version is reported as 0.1. Additional credential types will be incorporated in future versions.

A.3.1 Username/Password Credentials (TYPE=0)

Username and password credentials will be transmitted using the following format:

```
BEGIN_CREDENTIAL
TYPE=0
<plaintext username>
<plaintext password>
END_CREDENTIAL
```

It should be noted that plaintext username/password disclosure is permissible, as the TLS channel that exists between the client and Traust server provides confidentiality and integrity for all data transmitted.

A.3.2 X.509 Proxy Certificates (TYPE=1)

X.509 proxy certificates will be transmitted using the following format:

```
BEGIN_CREDENTIAL
TYPE=1
<first line of the PEM encoded proxy certificate>
.
.
<Nth line of the PEM encoded proxy certificate>
END_CREDENTIAL
```

If the proxy certificate being transmitted was generated using the `grid-proxy-init` command, the lines between `TYPE` and `END_CREDENTIAL` should contain, line for line, the contents of the `/tmp/x509up_u<uid>` file generated as the output from `grid-proxy-init`. An example PEM encoded X.509 proxy certificate is provided in Appendix A.6.

$$(GI_C GI_S)?(ITN_C (TN)^* ETN_C)^* RR_C (ITN_S (TN)^* ETN_S)? RR_S$$

Figure A.2: A regular expression describing valid Traust interactions

A.4 The Traust Resource Access Protocol

The messages presented in Section A.2 form the foundation of the Traust resource access protocol. This protocol is the core of the Traust system and allows clients to request access to resources protected by a particular Traust server. In this section, we formally describe the valid executions of the Traust resource access protocol. We then discuss several errors that can occur during executions of this protocol and how participants should respond to any errors which they detect.

A.4.1 Valid Protocol Executions

Figure A.2 presents a regular expression describing valid executions of the Traust resource access protocol. The abbreviations used in Figure A.2 stand for the message types presented in Section A.2: *GI* stands for the Get Info command (or a Get Info Response if sent by the server), *ITN* stands for the Initiate Trust Negotiation command, *TN* is a placeholder for the trust negotiation messages parsed by the client and server trust negotiation software, *ETN* stands for the End Trust Negotiation command, and *RR* stands for the Resource Request command (or a Resource Request Response if sent by the server). The subscripts indicate whether a message was sent by the client (C) or the server (S).

As shown in Figure A.2, the resource access protocol begins with the optional transmission of a Get Info command by the client and its corresponding server response. The client then has the option of initiating a trust negotiation session with the server prior to disclosing a potentially sensitive Resource Request. This trust negotiation consists of an Initiate Trust Negotiation command sent by the client, one or more pairs of Trust Negotiation messages sent by the client and server, and an End Trust Negotiation command sent by the client. This process can be repeated zero or more times. Once this is complete, the client will disclose a Resource Request to the Traust server. The server may then initiate an optional trust negotiation session with the client to determine whether or not the client satisfies the access policy of the requested resource. Once the

server determines whether the client satisfies the access policy of the resources, a Resource Request Response is sent to the client. This message either contains the credentials needed to access the resource or a notification of failure.

A.4.2 Handling Error Conditions

We now identify several broad classes of errors that could arise during the execution of the resource access protocol, and discuss how the participants in the protocol should react to any errors that they detect.

Invalid Protocol Executions Figure A.2 describes valid executions of the Traust resource access protocol. Any exchange of messages which deviates from this regular expression is considered to be an invalid protocol execution.

Unknown Messages An unknown message error occurs when a party receives a Traust message that does not match one of the messages described in Section A.2. For instance, an old Traust server might detect this type of error if a client sends a message described in a newer version of the Traust specification than that supported by the server.

Malformed Messages Malformed messages occur any time that a message of a valid type is constructed incorrectly. For instance, an error response to a Resource Request which contains both an error message and a credential disclosure would be considered malformed.

Any participant in the Traust resource access protocol detecting one or more of the above error types should terminate its connection to the other protocol participant immediately. This is a preventative measure meant to minimize information leakage and other attacks that might occur as a result of improper protocol execution. In addition to terminating the connection, the protocol interpreter should log an error message to aid the human user in determining the cause of the protocol failure.

A.5 A Sample Traust Session

In this section we present a sample interaction between a Traust server and a user, Alice, wishing to connect to the GridFTP server `gridftp.foo.org:2811`. The first thing that happens is that

Alice opens a TCP connection to the Traust server `traust.foo.org` on port 8162. After the TCP three-way handshake, the client and server form a TLS tunnel.

```
+--- KEY ----------------------------------------------------+
|                                                            |
|  ----> : Single message exchange                           |
|  ====> : Multi-message exchange (with details omitted)     |
|                                                            |
+------------------------------------------------------------+

  Alice                                                        Server
    <====== TCP 3-way handshake with traust.foo.org:8162 ========>

    <========= TLS handshake and session establishment ==========>
```

Alice is only willing to disclose Resource Request commands to Traust servers that have a "Traust" credential issued by `foo.org`. She invokes a trust negotiation to request that the server disclose such a credential, if it wishes to continue servicing Alice's request.

```
  Alice                                                        Server
    COMMAND=1 --------------------------------------------------->

    <=========== TN session (get "Traust" credential) ===========>

    COMMAND=2 --------------------------------------------------->
```

The server was able to satisfy this request, so Alice is willing to disclose her Resource Request. Alice issues the following Resource Request, which asks for a credential that allows her access to `gridftp.foo.org:2811` with the role "earthquake":

```
  Alice                                                        Server
    COMMAND=3                                     \------------->
    gsiftp://gridftp.foo.org:2811?role=earthquake  /
```

At this point, a trust negotiation is started by the server to ensure that Alice is a valid member of the earthquake research team. Such a negotiation could involve the disclosure of an identity certificate and a "Project Member" credential issued by the PI for the earthquake project.

```
  Alice                                                        Server
    <---------------------------------------------------- COMMAND=1

    <========= TN session (prove that Alice can assume ==========>
```

```
                     the "earthquake" role)

    <---------------------------------------------------- COMMAND=2
```

Alice can indeed prove that she is entitled to assume the “earthquake” role, so the server sends her the following credential as a Resource Request Response:

```
  Alice                                                           Server
                                  /  COMMAND=3
                                 /   RESPONSE=0
                                /    BEGIN_CREDENTIAL
                               /     TYPE=1
    <-------------------------/      -----BEGIN CERTIFICATE-----
                              \      .
                               \     .
                                \    -----END CERTIFICATE-----
                                 \   END_CREDENTIAL
```

At this point, the server terminates the connection.

A.6 A Sample PEM Encoded X.509 Proxy Certificate


```
-----BEGIN CERTIFICATE-----
MIICTzCCAbigAwIBAgIECmItzzANBgkqhkiG9w0BAQQFADBxMQ0wCwYDVQQKEwRH
cmlkMRMwEQYDVQQLEwpHbG9idXNUZXN0MSIwIAYDVQQLExlzaW1wbGVDQS1yb3Nl
LmNzLnVpdWMuZWR1MRQwEgYDVQQLEwtjcy51aXVjLmVkdTERMA8GA1UEAxMIQWRh
bSBMZWUwHhcNMDQxMTE1MjAwMzQ0WhcNMDQxMTE2MDgwODQ0WjCBhTENMAsGA1UE
ChMER3JpZDETMBEGA1UECxMKR2xvYnVzVGVzdDEiMCAGA1UECxMZc2ltcGxlQ0Et
cm9zZS5jcy51aXVjLmVkdTEUMBIGA1UECxMLY3MudWl1Yy5lZHUxETAPBgNVBAMT
CEFkYW0gTGVlMRIwEAYDVQQDEwkxNzQyMDY0MTUwXDANBgkqhkiG9w0BAQEFAANL
ADBIAkEAtz5vF2WNMlpR1u+JELCNX+GuhnXhSoL/2JmMcKCHAFW+MN7tMtXaf8uY
xfVPpyJ1wrvqF/jzsaYXN+VPsFnHmQIDAQABoyMwITAfBgorBgEEAZtQAYFeAQH/
BA4wDDAKBggrBgEFBQcVATANBgkqhkiG9w0BAQQFAAOBgQDVtBqHfP++NVTidwlo
5WXjTy0/prmm+YA95IyeakcNXCnhap7pUcQIVRx7p7uSNGPU4ObQAhiBYyIc6Kf6
7inxj3drI6ArLkmwwOm0Nmwui0/nBwW/vnPTfp5w5b9wtxXqzwGFiV5V8VWauATy
y4nxtrdan8TnDY8wsSKGXPkDFA==
-----END CERTIFICATE-----
-----BEGIN RSA PRIVATE KEY-----
MIIBOwIBAAJBALc+bxdljTJaUdbviRCwjV/hroZ14UqC/9iZjHCghwBVvjDe7TLV
2n/LmMX1T6cidcK76hf487GmFzflT7BZx5kCAwEAAQJBAJlWZG3mq73b+Knsbf0K
UHNQjdOOpt7MOj3NV8kTe4TZ0L1rIb1cs6ZRuBL7CeFQk7Mgd6fzAsUgyydgummf
nxUCIQDZepgZdrpKzRSwB/vcs/tTJQBS+eE9dPbjLhQmqn0wlwIhANezeyoXW0CD
l/QbAgePVYMbHZUzox6ZiweBIyopBp9PAiEAkrr1DhEd5cPyVkY8tw6z3cgUL0et
AWL+BA8dx2y15GcCIE9mJeQsjM6GohydBHY78MI97PnK9DSDWIX+py8RwvpJAiAx
```

xChT9GhOlpL0ABx/3vVMR/8rr1Orjd+9zWHUnYfWWA==
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
MIICXzCCAcigAwIBAgIBAjANBgkqhkiG9w0BAQQFADBjMQ0wCwYDVQQKEwRHcmlk
MRMwEQYDVQQLEwpHbG9idXNUZXN0MSIwIAYDVQQLExlzaW1wbGVDQS1yb3NlLmNz
LnVpdWMuZWR1MRkwFwYDVQQDExBHbG9idXMgU2ltcGxlIENBMB4XDTA0MDkxNTIw
MzY1NVoXDTA1MDkxNTIwMzY1NVowcTENMAsGA1UEChMER3JpZDETMBEGA1UECxMK
R2xvYnVzVGVzdDEiMCAGA1UECxMZc2ltcGxlQ0Etcm9zZS5jcy51aXVjLmVkdTEU
MBIGA1UECxMLY3MudWl1Yy5lZHUxETAPBgNVBAMTCEFkYW0gTGVlMIGfMA0GCSqG
SIb3DQEBAQUAA4GNADCBiQKBgQDefx+5/InGSaOCYUXfPRhOVLcqWBNPxWihFxme
+RbRZ45gZbjxrLuPlWttT4Pm8BL5JyRaQExmYGvNeFRMFruq3lce+XcxLzUTZoCG
O9WO9WP8AIj4LcrGOwgJDOlybE2c5ugDfh+7EV4lJAWHJeR2XxY0qwFgp+ugyhtb
SES5BQIDAQABoxUwEzARBglghkgBhvhCAQEEBAMCBPAwDQYJKoZIhvcNAQEEBQAD
gYEAk/QGTqp3iV6HwpRBCErLjZq6fya9i8xSu5Rvr0CBN/scn/4fpwaji/O67Oss
KWi3PJHWnvBpx2OOumO3aq//u1XbTrHehS3AOPi2+Zd+NK5vq3zKqKp/2W/3CIRO
PFI8VI2JsiAiA+NIp1UVhuCGoZ6U9v6epVSkSTC3Do3lmUk=
-----END CERTIFICATE-----

Appendix B

Traust Extensions for GridFTP

Here we discuss a way in which GridFTP can be extended to support dynamic access credential discovery via interaction with a Traust server. This will allow users of the GridFTP server to obtain the credentials necessary to access data provided by the server at run-time and eliminate the need for users to establish accounts on the server itself.

The main goal in this design was to use the existing GSI authentication mechanisms as much as possible as to avoid repeating unnecessary work. As such, this solution is rather lightweight and easy to implement.

B.1 The TRAUST Command

To facilitate the incorporation of Traust into GridFTP, we define a new FTP command, `TRAUST`. To initiate a GridFTP session supporting TRAUST interaction, the client opens an FTP connection to the GridFTP server and then issues the command:

```
TRAUST
```

This command should be entered prior to logging in to the system. The GridFTP server's response to this command will be response code 200. The body of this response will be as follows:

```
200-TRAUST <Traust server name or IP address>:<port>
200-<resource URI>
200-ATTRIB=(<attribute>,<value>)    // zero or more
200 <authentication method>
```

In this response, the port specification is optional and will be assumed to be 8162 when not present. The `<resource URI>` is a URI [BLFM05] that can be embedded in a Traust Resource Request command and the optional `ATTRIB` lines further specify the resource. At this point, the client contacts the Traust server described in the first line of the GridFTP server's 200 response and requests access to the resource described on the second line of the 200 response.

Should the exchange with the Traust server succeed, the Traust server will then issue the client whatever credentials are needed to log into the GridFTP server. The client then uses the authentication method listed on the third line of the GridFTP server's response to the TRAUST command along with the newly obtained credentials to log into the server. Acceptable authentication method descriptions to embed in the 200 response include "USERPASS" and "GSI", which represent the standard FTP username/password login and GSI authentication, respectively.

The `TRAUST` command only provides clients with the information necessary to obtain credentials that will allow them access to the GridFTP service at a base permission level. After issuing the `TRAUST` command, clients then use the information provided in the `TRAUST` response to carry out a Traust interaction with the indicated server to negotiate for the access credentials needed to log into the GridFTP server. These newly-obtained credentials are then used with an existing authentication mechanism (*e.g.*, GSI authentication) to establish a secure GridFTP session.

B.2 Allowing Permission Changes During a GridFTP Session

Although the `TRAUST` command is useful to determine whether or not a given client should be granted to access a particular GridFTP server, it will rarely be the case that all clients authorized to access a particular server should have access to all files stored on the server. To address this problem, we wish to allow clients the ability to negotiate for new access rights dynamically as they traverse the file system. We accomplish this through the use of "access hints" embedded in GridFTP server error messages.

For instance, if we would like to enforce access lists on a per-directory basis (as in the Andrew file system), we could imagine that the act of changing directories might cause authorization errors. For instance, if the client issues a `CWD` or `CDUP` command to the GridFTP server, they might be returned an error code 550 (access denied). This message usually takes the following form:

```
550 Access Denied
```

Rather than returning the standard FTP error code 550, a GridFTP server supporting the TRAUST command could return the following access denied message with an embedded access hint:

```
550-TRAUST <Traust server name or IP address>:<port>
550-<resource URI>
500-ATTRIB=(<attribute>,<value>)    // zero or more
550-<authentication method>
550 Access Denied
```

The `<resource URI>`, optional ATTRIB lines, and `<authentication method>` fields above carry the same meaning as those returned in response to the Traust command and, again, the port specification is optional. This enhanced error message will be interpreted by the client as an indicator that the current request failed, though successful interaction with the indicated Traust server could lead to the access credentials necessary to allow the failed operation to succeed. In this case, the client could take the following actions:

1. The client's Traust API is used to contact the indicated Traust server and request access to `<resource URI>`.

2. If the Traust interaction is successful, the client will be issued new access credentials by the Traust server that will allow access to the requested directory.

3. The client then uses the newly obtained credentials in a re-authentication exchange with the GridFTP server initiated by using the GridFTP AUTH command. This exchange will take place using the method specified in the access hint provided to the client.

4. The CWD or CDUP command can then be reissued and will succeed.

It is reasonable to assume that a good GridFTP client protocol interpreter could automate steps 1–4 after receiving an error message containing an access hint. In this case, the user of the GridFTP client would not need to manually intervene to trigger any of the interactions with the Traust server and would only notice a slight lag in the connection as the access credentials are re-negotiated. To enforce the directory level access controls discussed in this section, servers would need to return error messages containing access hints not only with failed CWD and CDUP commands,

but also with failed commands that attempted to access files in directories other than the current working directory (*e.g.*, commands such as `RETR`, `STOR`, and `LIST`).

One of the interesting features of this approach is that we can allow access hints to be issued with any error message. The client protocol interpreter need only parse error responses for the "`TRAUST`" string to know that a Traust interaction might possibly fix the error. This flexibility allows a GridFTP server to enable access hints that can enforce a wide variety of access control policies on the server's resources. For instance, since the server is free to embed different access hints in `RETR` or `STOR` requests for the same file, there is no reason that this system cannot be used to enforce different read and write access controls for a single file or directory.

B.3 A Sample Traust-enabled GridFTP Session

```
+--- KEY -------------------------------------------------+
|                                                          |
|  ----> : Single message exchange                         |
|  ====> : Multi-message exchange (with details omitted)   |
|                                                          |
+----------------------------------------------------------+


Traust.foo.org                   Alice                     GridFTP.foo.org
--------------                   -----                     ---------------

                                 <============= Connection Est. =========>

                                 --------------- TRAUST ----------------->

                                 <- 200-TRAUST traust.foo.org:8162 ------
                                    200-gsiftp://gridftp.foo.org
                                    200 GSI

<==== Traust session negotiating ====>
    for access to gridftp.foo.org

---------- X.509 proxy cert --------->
          allowing login to
           gridftp.foo.org

                                 <==== GSI AUTH exchange using newly ====>
                                         obtained proxy certificate
```

```
                                 <-------------- 234 OK -----------------

                                 ------------ CWD /earthquake ----------->

                                 <- 550-TRAUST traust.foo.org:8162 -----
                                    550-urn:gridftp.foo.org:resource:earthquake
                                    550-GSI
                                    550 Access Denied

<==== Traust session negotiating ====>
        for access to resource
urn:gridftp.foo.org:resource:earthquake

---------- X.509 proxy cert --------->
          allowing access to
              /earthquake

                                 <====== GSI AUTH exchange using =======>
                                      earthquake proxy certificate

                                 <-------------- 234 OK -----------------

                                 ------------ CWD /earthquake ----------->

                                 <--------------- 250 OK ----------------

                                 .
                                 .
                      [rest of GridFTP session]
                                 .
                                 .
```

References

[AV99] Paul Ashley and Michael Vandenwauver. *Practical Intranet Security: Overview of the State of the Art and Available Technologies*. Kluwer Academic Publishers, 1999.

[BFK99] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. KeyNote: Trust management for public-key infrastructures (position paper). *Lecture Notes in Computer Science*, 1550:59–63, 1999.

[BFL96] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *IEEE Conference on Security and Privacy*, May 1996.

[BFS04] Elisa Bertino, Elena Ferrari, and Anna Cinzia Squicciarini. Trust-X: A peer-to-peer framework for trust establishment. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):827–842, Jul. 2004.

[Bla79] G. R. Blakley. Safeguarding cryptographic keys. In *AFIPS Conference Proceedings*, volume 48, pages 313–317, 1979.

[BLFM05] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform resource identifier (URI): Generic syntax. IETF Request for Comments RFC-3986, Jan. 2005.

[BLMM94] Tim Berners-Lee, Larry Masinter, and Mark McCahill. Uniform resource locators (URL). IETF Request for Comments RFC-1738, Dec. 1994.

[BS00] Piero Bonatti and Pierangela Samarati. Regulating service access and information release on the web. In *7th ACM Conference on Computer and Communications Security*, pages 134–143, 2000.

[BS04] Moritz Y. Becker and Peter Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *5th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 159–168, 2004.

[CH02] Jan Camenisch and Els Van Herreweghen. Design and implementation of the *idemix* anonymous credential system. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 21–30, 2002.

[CJ75] Ellis Cohen and David Jefferson. Protection in the Hydra operating system. In *5th ACM Symposium on Operating Systems Principles*, pages 141–160, 1975.

[DA99] Tim Dierks and Christopher Allen. The TLS protocol version 1.0. IETF Request for Comments RFC-2246, Jan. 1999.

[DY83] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, Mar. 1983.

[HBSO03] Jason Holt, Robert Bradshaw, Kent E. Seamons, and Hilarie Orman. Hidden credentials. In *2nd ACM Workshop on Privacy in the Electronic Society*, Oct. 2003.

[HFPS99] Russell Housely, Warwick Ford, Tim Polk, and David Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. IETF Request for Comments RFC-2459, Jan. 1999.

[HHJS04] Adam Hess, Jason Holt, Jared Jacobson, and Kent E. Seamons. Content-triggered trust negotiation. *ACM Transactions on Information System Security*, 7(3), Aug. 2004.

[HJM+02] Adam Hess, Jared Jacobson, Hyrum Mills, Ryan Wamsley, Kent E. Seamons, and Bryan Smith. Advanced client/server authentication in TLS. In *Network and Distributed Systems Security Symposium*, Feb. 2002.

[HMM+00] Amir Herzberg, Yosi Mass, Joris Michaeli, Dalit Naor, and Yiftach Ravid. Access control meets public key infrastructure, or: assigning roles to strangers. In *IEEE Symposium on Security and Privacy*, May 2000.

[isr05] Internet security research lab–projects. Web Page, May 2005. ⟨http://isrl.cs.byu.edu/TrustBuilder.html⟩.

[KM04] Hristo Koshutanski and Fabio Massacci. Interactive trust management and negotiation scheme. In *2nd International Workshop on Formal Aspects in Security and Trust (FAST)*, pages 139–152, Aug. 2004.

[KN93] John Kohl and B. Clifford Neuman. The Kerberos network authentication service (version 5). IETF Request for Comments RFC-1510, Sep. 1993.

[KNe03] Chris Kaler and Anthony Nadalin (editors). Web services federation language (WS-Federation). Specification, Jul. 2003. ⟨http://www-106.ibm.com/developerworks/webservices/library/ws-fed/⟩.

[LDB03] Ninghui Li, Wenliang Du, and Dan Boneh. Oblivious signature-based envelope. In *22nd ACM Symposium on Principles of Distributed Computing*, pages 182–189, Jul. 2003.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, Mar. 1984. ⟨http://www.cs.washington.edu/homes/levy/capabook/⟩.

[lib05] Liberty alliance project – digital identity defined. Web Site, Apr. 2005. ⟨http://www.projectliberty.org/⟩.

[LM03] Ninghui Li and John Mitchell. RT: A role-based trust-management framework. In *Third DARPA Information Survivability Conference and Exposition*, Apr. 2003.

[mic05] Microsoft .NET passport. Web Site, Apr. 2005. ⟨http://www.passport.net⟩.

[Moa97] Ryan Moats. URN syntax. IETF Request for Comments RFC-2141, May 1997.

[NT94] Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. In *IEEE Communications*, volume 32, pages 33–38, Sep. 1994.

[NTW01] Jason Novotny, Steven Tuecke, and Von Welch. An online credential repository for the grid: MyProxy. In *Tenth International Symposium on High Performance Distributed Computing (HPDC-10)*, Aug. 2001.

[OS04] Adam J. O'Donnell and Harish Sethu. On achieving software diversity for improved network security using distributed coloring algorithms. In *11th ACM Conference on Computer and Communications Security*, Oct. 2004.

[PW03] Birgit Pfitzmann and Michael Waidner. Federated identity-management protocols– where user authentication protocols may go. In *11th Cambridge International Workshop on Security Protocols*, Apr. 2003.

[PWF+02] Laura Pearlman, Von Welch, Ian Foster, Carl Kesselman, and Steven Tuecke. A community authorization service for group collaboration. In *IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*, 2002.

[RWRS00] Carl Rigney, Steve Willens, Allan Rubens, and William Simpson. Remote authentication dial in user service (RADIUS). IETF Request for Comments RFC-2865, Jun. 2000.

[RZN+05] Tatyana Ryutov, Li Zhou, Clifford Neuman, Travis Leithead, and Kent E. Seamons. Adaptive trust negotiation and access control. In *10th ACM Symposium on Access Control Models and Technologies*, Jun. 2005.

[Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, Nov. 1979.

[SJR86] Robert D. Sansom, Daniel P. Julian, and Richard F. Rashid. Extending a capability based system into a network environment. In *ACM SIGCOMM Conference on Communications Architectures and Protocols*, pages 265–274, 1986.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep. 1975.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *17th ACM Symposium on Operating Systems Principles*, Dec. 1999.

[SW05] Amit Sahai and Brent Waters. Fuzzy identity based encryption. In *Eurocrypt 2005*, May 2005.

[TMvR86] Andrew S. Tanenbaum, Sape J. Mullender, and Robbert van Renesse. Using sparse capabilities in a distributed operating system. In *6th International Conference on Distributed Computing Systems*, pages 558–563, 1986.

[WSF+03] Von Welch, Frank Siebenlist, Ian Foster, John Bresnahan, Karl Czajkowski, Jarek Gawor, Carl Kesselman, Sam Meder, Laura Pearlman, and Steven Tuecke. Security for grid services. In *Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*, Jun. 2003.

[WSJ00] William H. Winsborough, Kent E. Seamons, and Vicki E. Jones. Automated trust negotiation. In *DARPA Information Survivability Conference and Exposition*, Jan. 2000.

[WWJ04] Lingyu Wang, Duminda Wijesekera, and Sushil Jajodia. A logic-based framework for attribute based access control. In *2nd ACM Workshop on Formal Methods in Security Engineering (FMSE 2004)*, pages 45–55, Oct. 2004.

[WYS+02] Marianne Winslett, Ting Yu, Kent E. Seamons, Adam Hess, Jared Jacobson, Ryan Jarvis, Bryan Smith, and Lina Yu. The TrustBuilder architecture for trust negotiation. *IEEE Internet Computing*, 6(6):30–37, Nov./Dec. 2002.

[YL05] Tatu Ylonen and Chris Lonvick. SSH transport layer protocol. IETF Network Working Group Internet-Draft, Mar. 2005. ⟨`http://www.ietf.org/internet-drafts/draft-ietf-secsh-transport-24.txt`⟩.

[YWS03] Ting Yu, Marianne Winslett, and Kent E. Seamons. Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation. *ACM Transactions on Information and System Security*, 6(1), Feb. 2003.

[Zim95] Philip R. Zimmerman. *The Official PGP User's Guide*. MIT Press, May 1995.

[ZVA+01] Yongguang Zhang, Harrick Vin, Lorenzo Alvisi, Wenke Lee, and Son K. Dao. Heterogeneous networking: A new survivability paradigm. In *2001 Workshop on New Security Paradigms*, pages 33–39, 2001.