# Protection of Information and Communications in Distributed Systems and Microservices

Antti Myyrä

Master's Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, October 31, 2018

HELSINGIN YLIOPISTO | HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

| Tiedekunta – Fakultet – Faculty | Laitos – Institution – Department |
|---|---|
| Faculty of Science | Department of Computer Science |

Tekijä – Författare – Author

Antti Myyrä

Työn nimi – Arbetets titel – Title

Protection of Information and Communications in Distributed Systems and Microservices

Oppiaine – Läroämne – Subject

Computer Science

| Työn laji – Arbetets art – Level | Aika – Datum – Month and year | Sivumäärä – Sidoantal – Number of pages |
|---|---|---|
| Master's Thesis | October 31, 2018 | 64 |

Tiivistelmä | Referat | Abstract

Distributed systems have been a topic of discussion since the 1980s, but the adoption of microservices has raised number of system components considerably. With more decentralised distributed systems, new ways to handle authentication, authorisation and accounting (AAA) are needed, as well as ways to allow components to communicate between themselves securely. New standards and technologies have been created to deal with these new requirements and many of them have already found their way to most used systems and services globally.

After covering AAA and separate access control models, we continue with ways to secure communications between two connecting parties, using Transport Layer Security (TLS) and other more specialised methods such as the Google-originated Secure Production Identity Framework for Everyone (SPIFFE). We also discuss X.509 certificates for ensuring identities. Next, both older time-tested and newer distributed AAA technologies are presented. After this, we are looking into communication between distributed components with both synchronous and asynchronous communication mechanisms, as well as into the publish/subscribe communication model popular with the rise of the streaming platform.

This thesis also explores possibilities in securing communications between distributed endpoints and ways to handle AAA in a distributed context. This is showcased in a new software component that handles authentication through a separate identity endpoint using the OpenID Connect authentication protocol and stores identity in a Javascript object-notation formatted and cryptographically signed JSON Web Token, allowing stateless session handling as the token can be validated by checking its signature. This enables fast and scalable session management and identity handling for any distributed system.

ACM Computing Classification System (CCS):

Security and privacy
- Security services
  - Authentication
  - Authorization
- Systems security
  - Distributed systems security

Computer systems organization
- Architectures
  - Distributed architectures

Avainsanat – Nyckelord – Keywords

AAA, distributed systems, microservices, secure communication

Säilytyspaikka – Förvaringställe – Where deposited

Muita tietoja – Övriga uppgifter – Additional information

# Table of Contents

1. Introduction....2
2. Authentication, Authorisation & Accounting....4
3. Access Control Models....6
    3.1. Mandatory Access Control....6
    3.2. Role-Based Access Control....7
    3.3. Discretionary Access Control....8
    3.4. Attribute-Based Access Control....8
4. Secure Communication over Networks....10
    4.1. Diffie-Hellman Key Exchange....10
    4.2. Public Key Infrastructure....11
    4.3. Transport Layer Security....13
5. Distributed Systems and Microservices....16
    5.1. Overlays and Service Meshes....17
    5.2. Secure Communication Mechanisms Between Services....18
        5.2.1. Mutual TLS....19
        5.2.2. Secure Production Identity Framework for Everyone....20
6. Distributed Authentication, Authorisation & Accounting....21
    6.1. Traditional Mechanisms of Distributed AAA....23
        6.1.1. Simple Authentication Security Layer....24
        6.1.2. Kerberos....24
        6.1.3. Security Assertion Markup Language....26
    6.2. New distributed authentication & authorisation models....27
        6.2.1. JSON-based standards....27
        6.2.2. OAuth 2.0....29
        6.2.3. OpenID Connect....32
        6.2.4. Macaroons....33
7. Communication Mechanisms Between Microservices and Distributed Systems....36
    7.1. Synchronous Mechanisms....36
        7.1.1. Remote Procedure Calls....37
        7.1.2. HTTP-Based Methods....38
    7.2. Asynchronous mechanisms....39
        7.2.1. Advanced Message Queuing Protocol....41
        7.2.2. Message Queue Telemetry Transport....41
        7.2.3. Message Broker Specific Protocols....42
8. Authentication, Session Management and Identity Propagation for Microservices....44
9. Conclusions....46
References....48
Attachment: Ambassador AuthService....53
    A.1. Main....53
    A.2. Login....55
    A.3. Auth....59

# 1. Introduction

Any system that can be used by more than a single user must have some form of *authentication, authorisation and accounting* (AAA) taking place. Some part of AAA can happen outside of a computer system, for example having a guard standing in front of the room housing computers that are not connected to external networks. When users have different roles and privileges, a more sophisticated AAA setup needs to be in place and the guard might have a list of people and what they are allowed to do. With connected systems that can be accessed from anywhere on the planet, validating users, their permissions and logging their actions automatically is vital for any system or software in order to function properly.

As the amount of software used in the world constantly grows, so does the amount of flaws and security vulnerabilities available for attackers. With European Union's *General Data Protection Regulation* (GDPR) and similar legal frameworks around the world, organisations are looking into securing their systems with emerging priority. The AAA is a key part of these endeavours: while we want to provide users their data, we are also trying to prevent unauthorised access by anyone else. With the number of system integrations rising constantly, more often another system ends up using the service instead of the end user. As systems use each others' data, it is also critical to know where the copies of the data end up and to what ends the data will be used in the future. In this regard privacy of users' own data is also a growing concern.

As systems grow more complex, data itself is often stored separately from the user accounts. To complicate this, decisions on who is allowed to read and modify the data depend on their identity. Modern AAA provides the solutions for both tying the user back to her data, as well as giving service the tools to verify that the user is also the one she claims to be. Another current development is the increasing complexity of system architectures in the *Internet of Things* (IoT), where a sensor or probe might not only produce data, but can also use data gathered by others to enhance its own functionality. As complexity and amount of data grows, so do demands of load that systems are expected to handle. Either authentication or authorisation cannot become a bottleneck that restricts development.

In Chapters 3, 4 and 5 we discover current and time-tested technologies in securing and verifying data and communications between systems, with focus on access control, the *Transport Layer Security* (TLS), *Public Key Infrastructure* (PKI) and communication frameworks for distributed systems. In Chapter 6 we are looking into frameworks and technologies that have been traditionally used to enable AAA for distributed systems. Chapter 7 consists of communication models between separated system components and systems with focus on the synchronous and asynchronous communication, as well as the publish/subscribe communication model.

In Chapter 8 we are presenting an example of how new technologies can be put together, with an authentication service implementation for an open source Ambassador API gateway [Amb17]. The service offers authentication by acting as *OpenID Connect* authentication protocol client, storing authentication details in cryptographically signed claim-representations, *JSON Web Tokens* (JWT). These JWTs can also be validated without earlier knowledge of them on each incoming request, as they carry the proof of authentication with them and they can be validated from their signature. The service can also revoke signed tokens and operate both by itself or scale in parallel, supporting any number of users without too much effort.

# 2. Authentication, Authorisation & Accounting

In this chapter we cover the basics of authentication, authorisation and accounting (AAA). The emphasis will be on definitions of AAA concepts, as well as the most common access control models currently in use and examples of their usage.

Authentication involves validating that the user is who she claims to be [MET99]. Authentication is often done with username and password, or by transferring the identity from another service that has authenticated the user. Other methods involve using certificates and secondary tokens, but these are usually used together with passwords. We will cover cryptography-based authentication and especially certificates more thoroughly in section 3.

Once identity has been confirmed, each action the user takes must be validated to be within their assigned privileges. Authorisation covers the validation of actions according to system-specific conditions. Principles for secure authorisation design were presented in 1975 by Saltzer and Schroeder [SaS75]. Despite their work coming from an age where the Internet or distributed systems did not exist, it is still surprisingly current in its design suggestions. One example of these suggestions is the *Complete Mediation* principle, which states that every access to every object within the system must be checked for authority each time it is accessed. This guarantees that changes to user privileges are in effect immediately, not only after the user logs in next time. Other good basis for secure systems is the *Least Privilege* principle, meaning that all users, roles or groups should always start without any rights and only ones needed to complete the task should be given.

After user has been correctly identified and each action authorised, actions taken must also be logged and this is where accounting comes in. Accounting covers collecting the information of user actions, which can then be used for logging, billing and possible security alerts if user is doing something that normally should not be done, even with the correct privileges. In secure systems, logs should also be sent to a separate location so accounting is preserved even in the event of a hacking incident or a catastrophic failure where the system is wiped out. Accounting is not covered too broadly in this thesis, although it is a key part of any secure system. The reason for this is the fact that account-

ing itself is useless without authentication and authorisation, but it can be fitted into any of the models that we cover. A little bit more about accounting can be found in Chapter 6, where we look into the simplicity of accounting with different authentication and authorisation technologies.

# 3. Access Control Models

Access control is the enforcement of rights after the user has been authorised and authenticated: to whom the access to a resource is given and who should not be allowed. Access control models differ based on whether we want to target individual users, user groups or perhaps both. Starting from simple mandatory access control, models have evolved as use cases have grown more complex [KHD10]. Currently numerous models exist for different scenarios, but most of them can be categorised into four different classes that we cover more deeply in the following subchapters. To simplify differences and present different use cases, we're using an example of hospital IT with all the models.

## *3.1. Mandatory Access Control*

*Mandatory access control* (MAC) gives control to an administrator of the resources. The creator of a file or the end user, to whom access is given, cannot delegate access further. Its history in the US military data protection, MAC has strict organisational hierarchy and two different security models known as *Biba* and *Bell-LaPadula* [Lin06]. Biba is a more open model, where users at lower clearance levels can read what users at higher levels have written, but cannot themselves modify this information. Bell-LaPadula on the other hand, is the more standard access model where users at higher clearance levels can read and modify anything others have written at lower levels, but cannot read (or modify) anything created in clearance levels higher than their own. Bell-LaPadula is the access control model most often used when handling critical information.

The MAC can be used to create simple rules that can easily cover every use case within the organisation, but it can also freeze organisational structures, as changes to a more non-hierarchical structure are not supported by the used access control model.

In our example, a hospital using MAC might be using Bell-LaPadula, as performance reviews of personnel, done by their superiors, should not be readable by their coworkers. This allows more senior physicians on a higher authorisation level to read and modify drug prescriptions and medical reports created by doctors with lower seniority, but would not allow nurses to view them while attending to a patient. If we do not need

patients to have access to the system, the Biba model, where younger doctors and nurses can see the info for all patients, but cannot modify them without action from the attending physician, might be a more suitable candidate as the access control model. Accounting is also critical in protection of patient records and each access or modification should be recorded and later validated to prevent confidential knowledge from spreading.

## 3.2. Role-Based Access Control

*Role-based access control* (RBAC) does not concern itself with organisational structure nor individual users, but the roles they possess [SCF96]. For example, user with the role "employee" might see only her own information, whereas user with the role "HR" can see other people's information as well, but cannot delete anything as this is reserved only for the user with the "administrator" role. In RBAC, access can also be given to a combination of roles so only HR people within a specific department might be able to access the personal information of people within their department, but not personal information from other department's personnel. Similarly to MAC, in RBAC users often cannot delegate access rights outside their own groups and rights are often given based on automated policies.

With increased complexity, role delegation also requires more planning and auditing from time to time, with the time period depending on the confidentiality of the protected information. To reduce complexity, RBAC also includes sub model categories that limit allocated roles based on the login session [SCF96]. With the *Core RBAC* (or Flat RBAC) model, user is allowed only one role per session to prevent complex role management. The *Hierarchical RBAC* model is similar to Core RBAC in the sense that the user will be equipped with the highest role in the role hierarchy according to her own roles. As access management in hierarchical RBAC is based on roles and subroles, with user being given the highest role that she has, without all the underlying roles. Between single-role models and unlimited roles, there's also the *Constrained RBAC* model that limits the number of roles based on organisational policy.

The RBAC for our example hospital might be based on hierarchical RBAC model, so personnel can have access to patient information within their own department, but not to patients in other departments of the same hospital. RBAC also allows for patients to ac-

cess the system as the patient role allows seeing their own medical records.

### 3.3. Discretionary Access Control

Known especially from the standard access control models of Unix and Linux, *Discretionary Access Control* (DAC) is the most flexible of the four models. In file systems, DAC allows the owner of a file or folder to completely control access to her resources and even to change user and group ownership of the file to someone else [OSM00]. DAC also prevents unauthorised users from seeing object details, such as file size or modification dates.

Flexibility makes DAC a compelling option as the administration does not have to deal with user requests for access rights to information, but it also comes with a cost of security concerns. With DAC, any user can give out access to protected resources on purpose or by accident, which both can be leveraged by malware or badly written software running with user's access rights [OSM00].

As DAC can be quickly implemented, our hospital might initially use it with their shared filesystems. With this, an accountant for the hospital might have write access to all the bills her department sends, but have only read access to bills from other departments. When the accountant accidentally opens a new bill for the department that contains ransomware, it would encrypt only the parts of the shared drive that can be written into, leaving the fate of department bills to depend on how well their backup policy is working. Bills from other departments would be safe as the accountant does not have write access to them.

### 3.4. Attribute-Based Access Control

*Attribute-Based Access Control* (ABAC) is based on the idea of combined roles in RBAC, but taken much further. The ABAC policies can include numerous rules with simple Boolean Logic (if, else). For example, a policy can be used to check if the user is over 18 years old and if she has completed necessary safety courses and has a driver's license in order to rent a car [YuT05]. Other simple use case for ABAC are university courses where student has to have necessary knowledge before participating to a higher-level course.

Common downside of ABAC is the added complexity that can lead to someone gaining access to wrong resources. Prevention of these cases requires sane defaults, well-thought out rules and continuous auditing.

ABAC policies in the example hospital environment could be related to individual performed operations. For example, a surgeon might have read and write access to a surgery report for an operation that she is performing, but only read access to a one that is done by one of her colleagues. After the surgery has been completed and report written, write access might be revoked to maintain auditable records of past events.

# 4. Secure Communication over Networks

For almost any single network request over the Internet, tens or hundreds of switches, routers and servers are involved. Traffic can be listened in, captured or modified at any point of its journey by malicious actors. Because of this, cryptography is required to ensure both confidentiality and integrity of digital communication between remote parties.

Public-key cryptography is a system that uses pairs of keys, public and private, to encrypt communication between parties. Public-key cryptography, also known as asymmetric cryptography, was invented in two places [Ell99]. In 1970 a British governmental cryptographer James H. Ellis presented his work in an internal document. Without knowledge of Ellis' work, that was then a British government secret, two US cryptographers, Whitfield Diffie and Martin Hellman invented a similar solution and published it in 1976. Diffie and Hellman presented the key exchange algorithm named after them as Diffie-Hellman key exchange that later became a starting point for most of secure communication happening over the Internet. Ellis never received recognition for his work as it was made public only after his death in 1997.

## *4.1. Diffie-Hellman Key Exchange*

With key exchange algorithms, two parties can set up a secure way of communicating without knowledge of each others' private keys, and without anyone else being able to observe the conversation by listening even when the whole traffic session has been copied.

In the Internet, one of the most common use cases for public-key cryptography is the key exchange in *Transport Layer Security* (TLS). The TLS is a protocol enabling encrypted *Hypertext Transport Protocol* (HTTPS) traffic using a key-exchange algorithm known as *Diffie-Hellman key exchange* [DVW92]. TLS, that also has support for other key-exchange protocols, is in focus at Chapter 4.3. Diffie-Hellman (DH) is an implementation of modular arithmetics, where an interceptable modulus and base are used together with secret integers that each party keeps only to themselves. With secret integers, parties can set up a shared secret that is not feasible to calculate for anyone listening in. This exchange is covered in Figure 1.

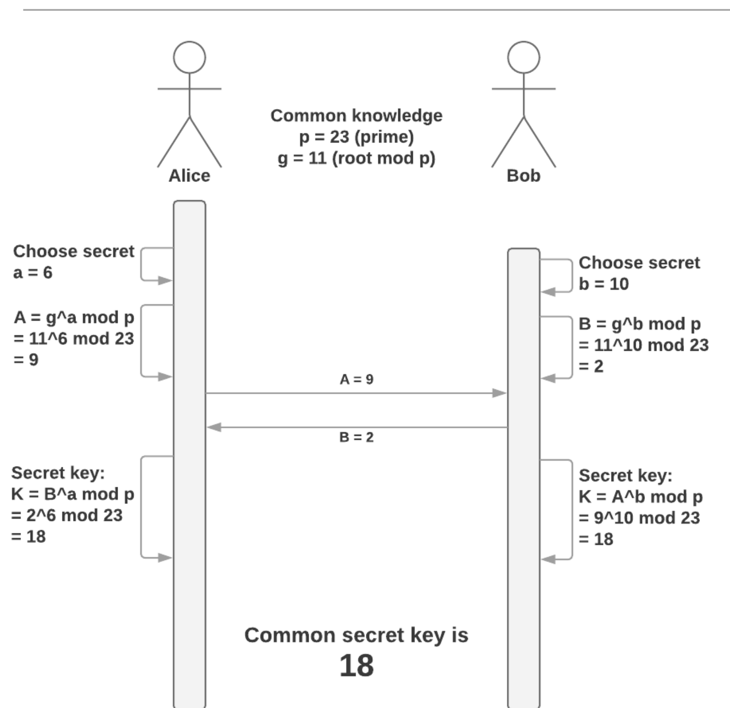


*Figure 1: Diffie-Hellman key exchange (simplified)*

As both sides discard their secret integers after the session has ended, it is not possible for a listener to decrypt the conversation afterwards. It is however possible to execute a man-in-the-middle (MITM) attack, with attacker performing Diffie-Hellman key exchanges with both parties, making them think that they are communicating only with each other. To prevent this, public key certificates are used to verify identity of the server, and in some cases to verify identity of the client as well.

The DH key exchange shown in Figure 1 is a simplified one and is presented in this form as a toy example. Modern DH has considerably bigger values as secrets and prime p. Currently recommended primes and secrets are 2048 bits in length [GiB17], making possible combinations close to impossible to calculate for an eavesdropping attacker. In addition elliptic curves are often used in place of modular arithmetics in a key-exchange protocol known as *Elliptic Curve Diffie-Hellman* (ECDH) [Tur14].

## 4.2. Public Key Infrastructure

Methods and physical means for signing, distributing, using, storing, distributing and revoking certificates are collectively known as *Public Key Infrastructure* (PKI), and the

standard format for certificates is known as X.509 [RFC5280]. The PKI defines how trust to a specific certificate is allocated and how it propagates. Public key certificates are proofs of key ownership, issued by an external trusted party known as a *Certificate Authority* (CA) and validated by a *Registration Authority* (RA). PKI's use in certificate acquisition is presented in Figure 2.

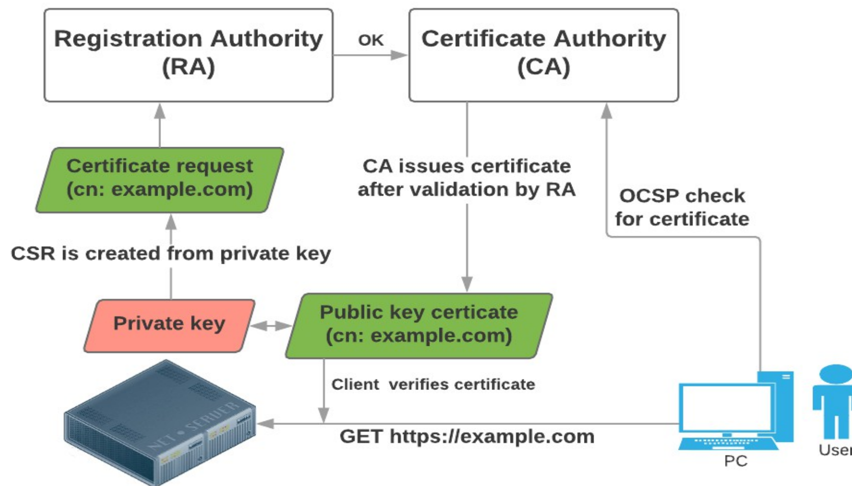


*Figure 2: Public Key Infrastructure for SSL certificates*

In a typical client-server context, server has a self-created private key that is kept secret and never transmitted towards a client. In HTTPS, certificate subject is a certain host-name, with a signed certificate proving that the private key belongs to the verified party. The process involves a *Certificate Signing Request* (CSR) that includes owner information such as company name, country and city, as well as *Common Name* (CN) and *Subject Alternative Names* (SAN) that are the domain names that require a certificate. Certificate authority processes the CSR and checks that the requesting party actually controls the domain names that the certificate is requested for. The checks vary based on the level of validation required for the certificate, but often a DNS record or file under a specific path of the domain is required to grant a certificate for that domain.

Validation process varies depending on the type of certificate [RFC3647]. With domain validation, only the control of the domain(s) mentioned in CSR is checked through a HTTP request, email or a new DNS record and is the most common certificate valida-

tion type as it can be easily automated. Organisation validation involves confirming the certificate with a person that is listed as a contact for the organisation in an official company register, in addition to domain validation. Extended validation covers both of these and also requires a written intent by an official representative of a organisation requesting the certificate, as well as verification call from the RA to the representative to confirm the request.

Once the RA has finished validation, it will give a permission for the CA to issue a certificate for the requested domain names. The certificate has a property that as it was created based on the specific CSR, it can also be used only with the private key used to create the CSR. As browsers include the CA's root certificates, a new certificate is trusted without updates for the browser. Certificates have a validity period covering from a few months to even 3 years, preventing stolen certificates to be used indefinitely. If any compromise is detected, certificates can be revoked faster in two different ways. The original method was by using a *Certificate Revocation List* (CRL) that is maintained and signed by the CA. The CRL is updated regularly by the browser and then used locally, so no external request is required for each check. CRLs are considered problematic as they do not provide an instantaneous way to revoke a compromised certificate. Because of this an alternative method, known as the *Online Certificate Status Protocol* (OCSP) is also used [RFC6960]. The CAs operate OCSP endpoints that have knowledge of every signed certificate, allowing users to check certificate validity in real time.

## 4.3. Transport Layer Security

*Transport Layer Security* (TLS) is a collection of cryptographic protocols that enable secure communication over the Internet [Tur14]. Besides Diffie-Hellman, TLS also supports other key exchange algorithms and aims to be a standard for all secure communication over networks, fitting to almost every communication use case involving TCP. TLS was originally developed in 1999 to replace the *Secure Sockets Layer* (SSL) de facto standard that had been in use previously. Currently, an earlier SSL version 3.0 and first TLS version 1.0 implementation are considered insecure and their use is discouraged. TLS versions currently supported are 1.1, 1.2 and the new version 1.3, which was accepted as a standard in August 2018 [RFC8446].

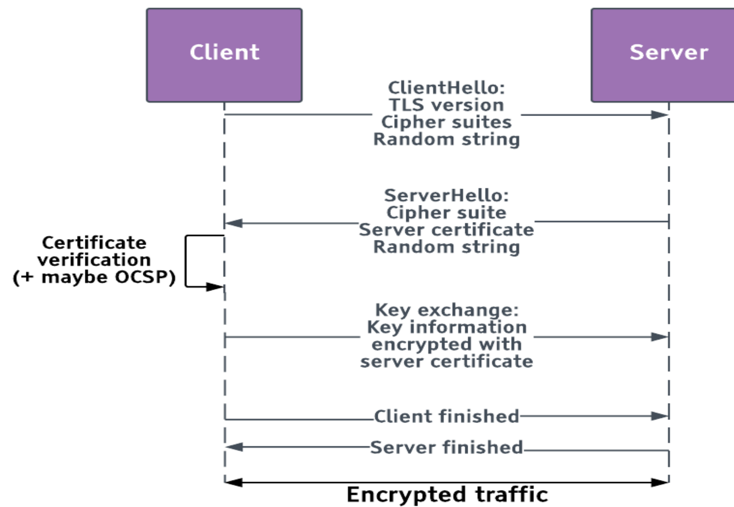


*Figure 3: The TLS 1.2 three-way handshake. Diffie-Hellman randomised strings shown in Figure 2 are sent as part of ClientHello and ServerHello.*

As illustrated in Figure 3, when a client contacts the server with a list of cipher suites and TLS versions supported, the server selects the best ones it supports and replies with them accompanied by its public certificate and its public key. After verifying certificate validity (signed by a known CA, has not been expired, not in revocation list and checked with OCSP) to prevent MITM attacks, the client encrypts the response using the server's public key, so it is decryptable only with the server's private key. Client-generated *Pre-shared key* (PSK) is also included. Pre-shared key is used to set up the shared secret used in the actual TLS session. Shared secret is agreed by using some form of the Diffie-Hellman key exchange. As both parties delete the unique shared secret after the session is over, the TLS sessions are not decryptable even if servers' private key is exposed later. The undecryptability of older sessions is known as *forward secrecy*.

Besides key exchange, TLS is also protecting the established connection. During key exchange, parties also agree on the used cipher that will be symmetrically encrypted using the established shared secret. In addition, TLS also protects connection reliability as it checks each incoming message with message authentication code, preventing the tampering of encrypted data [RFC8446].

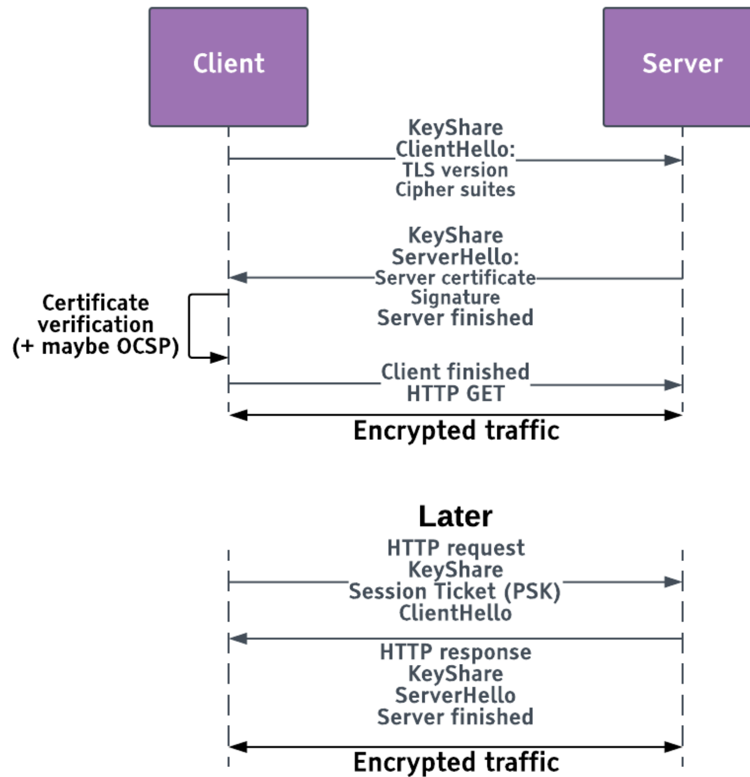


*Figure 4: TLS 1.3 handshake & 0-RTT resumption later*

The older stable version, TLS 1.2 was published in 2008 and had served for exactly 10 years when it was superseded by TLS 1.3 [RFC8446], that brings in new supported ciphers and obsoletes older ones. Perhaps the biggest change from the old standard is the shorter handshake, shown in Figure 4: instead of the standard three-way handshake shown in Figure 3, TLS 1.3 uses a two-way handshake to establish the connection, saving time every time user connects to a new server endpoint. TLS 1.3 can can also resume previous TLS sessions with an earlier shared key *(Session Ticket)*, allowing the client to send encrypted data along the first TCP packet. This feature is known as *Zero Round Trip* (0-RTT).

TLS forms the backbone of secure communication between networked hosts. In Chapter 5.1, we continue with the same TLS 1.3 standard to see how it can also be used to verify connecting clients or any two parties wanting to share data securely.

# 5. Distributed Systems and Microservices

As the term suggests, distributed systems consist of physically distributed components that communicate with each other over a network. An example of such setup can be seen in Figure 5. The scope of distributed systems varies, with some having all of their components within a single datacenter, while others can have a global presence with data stored and transferred between endpoints all around the world.

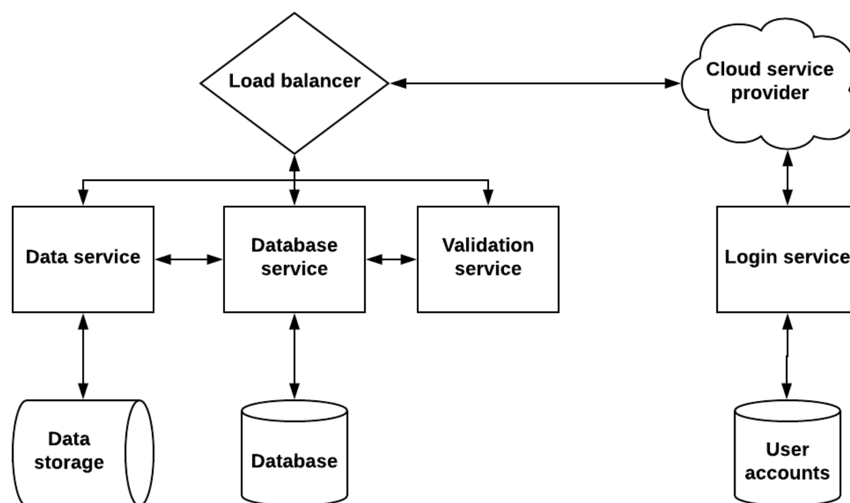


*Figure 5: An example of microservice platform with user accounts stored in another service*

Even though application with its data residing on a separate server can also be defined as a distributed system, we are focusing more on systems that have their data stored in different locations. For example, user accounts and passwords can be stored in a separate location or service from the one used for the application. The application can then call for the authentication service to verify user credentials.

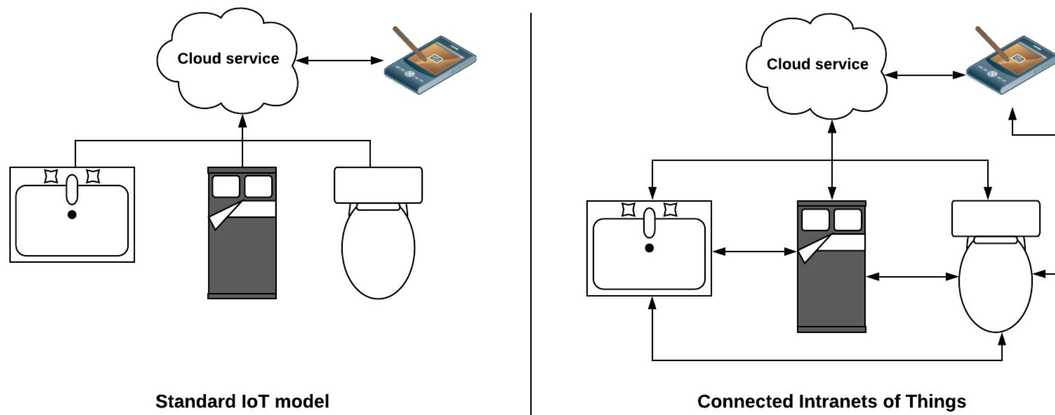


*Figure 6: The movement of data within the IoT and Connected Intranets of Things paradigms*

The paradigm of disappearing master authority of data is clearly visible in the fields of Internet of Things (IoT) and data processing. Traditionally, sensors have been the ones publishing data and other systems have subscribed to their data feeds. This has lead to a clear structure and direction for the system, even with many systems subscribing to the same data [EFG03]. A newer, emerging paradigm that is shown in Figure 6 can have sensors subscribing to each others' data and doing their own collecting based on that data. The model, named as *Connected Intranets of Things* [RZL13], requires new types of authentication and authorisation schemes as a single compromised sensor or probe could otherwise be used to tamper and steal huge amounts of data. For example, if the bed presented in Figure 6 could tell toilet to flush, it might do so repeatedly, wasting huge amounts of water.

## 5.1. Overlays and Service Meshes

In a constantly changing environment the idea of servers having a dedicated IP address or hostname is evaporating fast, as any service can exist only when it's needed or have 10 replicas of it to handle the incoming load. As virtualised or containerised applications can be started on any host, networking has evolved to find the correct destination for requests. As shown in Figure 7, overlay network operates through virtual network interfaces within compute nodes, and passes its packets encapsulated through a standard physical network.

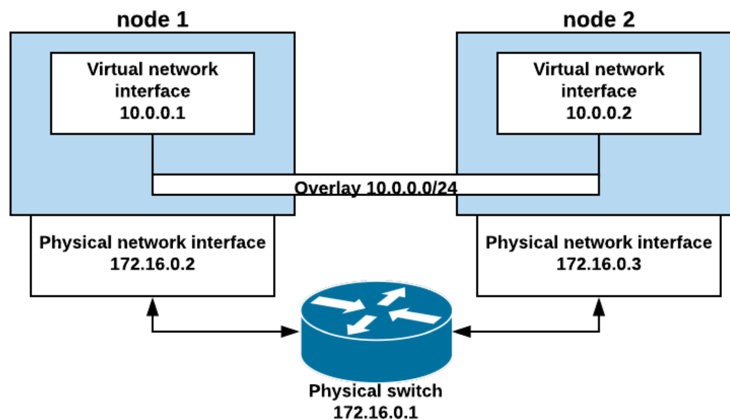


*Figure 7: Overlay network flows through the standard physical network*

Overlay networks can have the knowledge of each service endpoints location and can forward traffic accordingly [AKB01], allowing services to talk with each other independently of their physical location. With the addition of security features such as request auditing and firewalling, the concept of overlay networks has evolved into service meshes that handle all communication between different distributed services. In Chapter 5.2.2, we are taking a closer look at *Secure Production Identity Framework for Everyone* (SPIFFE) for network request authorisation and auditing and observe a real world example of SPIFFE's usage in Chapter 8.

## 5.2. Secure Communication Mechanisms Between Services

The basis for secure networking has long been that private networks are secure and communication within them is safe from prying eyes. With constantly changing distributed systems and changing attack vectors, this approach has become a liability and new approaches are needed to secure the data [GiB17].

Often the first step in securing internal networks is the same we have been using for communication over the Internet: TLS X.509 certificates. While this works fine on a standard system where front end services query back end services, and they in turn query databases, it may become problematic when services are subscribing and publishing data to each other, similar to the Connected Intranets of Things model. *Publish-sub-*

*scribe*, explained in more detail in Chapter 7.2, requires both the sending and receiving end to be sure about the other's identity. Below we are describing two ways to deal with client authentication: mutual TLS using client certificates and a modern SPIFFE framework using X.509 certificates in requests for authentication.

## 5.2.1. Mutual TLS

The TLS specification includes support for client certificates, allowing the server to ask for the client to verify itself through some trusted CA that the server knows and trusts. This allows an external party to verify clients and removes the need for password authentication, trading password management for certificate management.

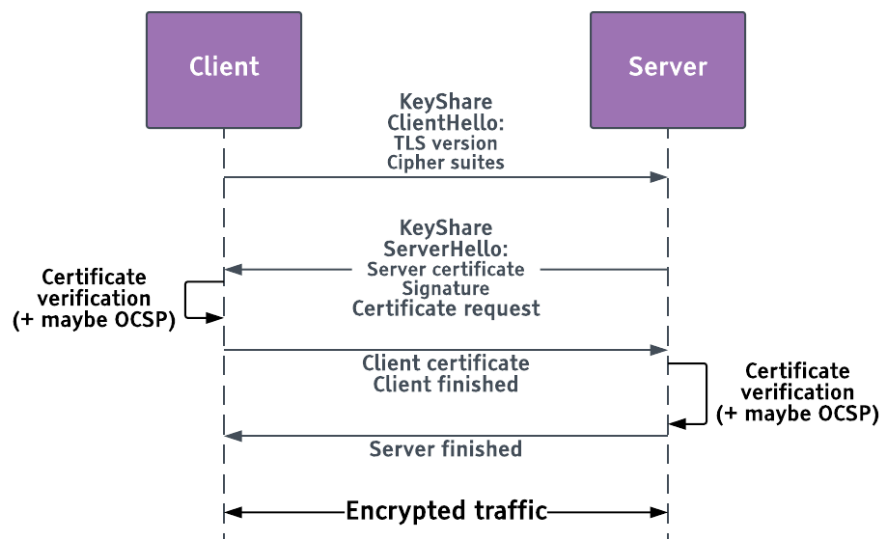


*Figure 8: Mutual TLS handshake in TLS 1.3*

Client certificates are requested during the initial handshake by the server, when it adds CertificateRequest to its first response [RFC8443], as shown in Figure 8. This adds a step to the handshake as the client needs to send its certificate to the server, thus making handshakes in TLS 1.3 a three-way and in TLS 1.2 a four-way one.

In untrusted networks, mutual TLS is a good choice for preventing brute-force and denial of service attacks, as the attacker is not able to send any data without a valid certificate. The disadvantage of using client certificates is the added overhead of certificate management and renewal, although with the correct software this can be automated.

### 5.2.2. Secure Production Identity Framework for Everyone

*Secure Production Identity Framework for Everyone* (SPIFFE) is an open standard initiated by Google for service identities in a fastly changing distributed environments [SPI17]. It is comprised of three components: a specification for naming identities (SPIFFE ID), *SPIFFE Verifiable Identity Document* (SVID) for authenticating the service in requests, and the Workload API specification to generate SVIDs for services, making it possible to create, renew, revoke and verify SVID certificates. Workload API functionality is for validating identity for a single service, and it is shown in Figure 9 along with other SPIFFE components.

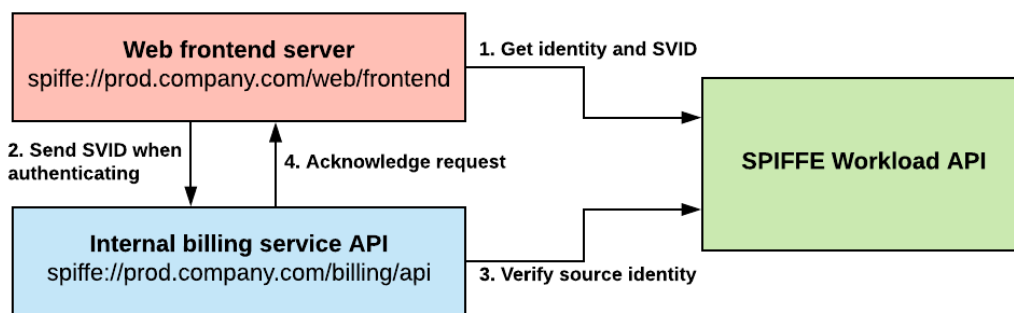


*Figure 9: SPIFFE communication between a web front end and an internal API*

SPIFFE IDs are *Uniform Resource Identifiers* (URIs) including the protocol (spiffe://), trust domain (for example, prod.company.com) and a path that either identifies the service (for example, /billing/backend), service owners (for example, /group/accounting/user/bill) or is just a unique identifier for it (for example /543ca66a-3215-4f4b-a835-1771fe64279d). The trust domain does not need to be the actual address of the platform, but it should remain the same for every service, with only the path changing depending on the service.

The SVID is a certificate conforming to the X.509 standard [RFC6960], giving it the same cryptographical properties as standard HTTPS certificates and making it possible to verify SVIDs outside of SPIFFE. SVID is used with requests, making it possible for the accessed service to verify the requesting party's identity and also to log it, making auditing on a request-level a possibility.

# 6. Distributed Authentication, Authorisation & Accounting

Distributed Authentication, Authorisation and Accounting (AAA) is not a new theme, with research and implementations existing from the beginning of 1990's. These implementations usually include a single point of failure in the authentication step, although authorisation can in some cases be done in a distributed manner.

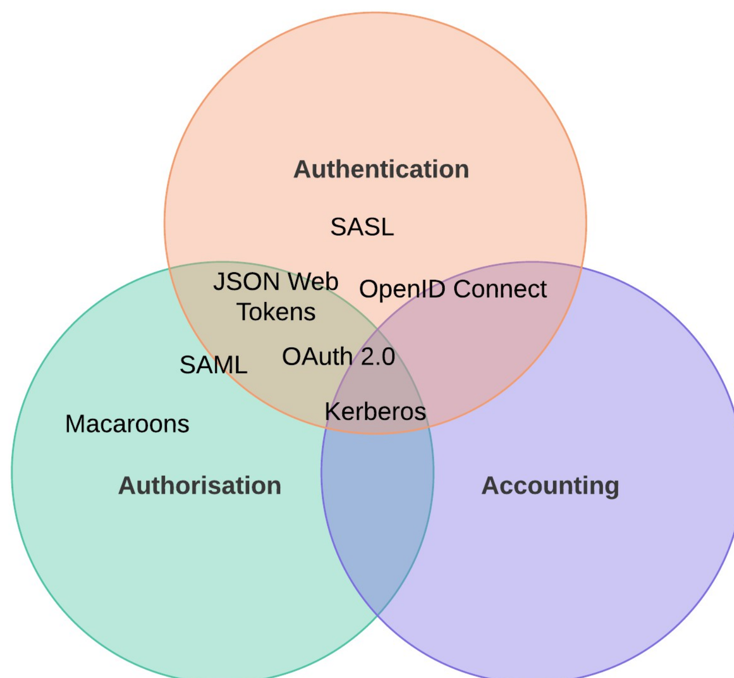


*Figure 10: Technologies and their relation to AAA*

Figure 10 illustrates technologies presented in this chapter and their relation to authentication, authorisation and accounting. The noticeable lack of accounting technologies is due to the fact that accounting by itself does not do anything, and it is always related to authentication or authorisation, or even both. Different technologies are organised in a way that shows how simply accounting can be tied to their usage. Each one of the presented technologies can be made to support accounting, but technologies such as Kerberos are built in a way that makes accounting easier to provide.

As illustrated in Figure 11 below, authentication and authorisation in distributed systems can be done in four different ways. Each one of them includes different possibilit-

ies on how authentication and authorisation can be arranged in distributed systems, and each one has its own benefits and drawbacks.

| | Local auths | Centralised auths |
|---|---|---|
| Local authn | Traditional way<br>To each service its own<br>Not good for distributed systems | Not a valid approach in any context |
| Centralised authn | Service-specific control<br>Easier to maintain fine-grained object permissions | Best control, no need to duplicate changes<br>Need for clear roles (RBAC, ABAC) to work efficiently |

*Figure 11: Possibilities for distributed or centralised authentication (authn) and authorisation (auths)*

The first example is the traditional one of each service doing their own authentication and authorisation. When a monolithic application is broken down to smaller services without re-thinking the AAA infrastructure, the same credentials database can be used by all services. But as services grow and start requiring their own databases, services start to maintain their own credentials along with object permissions for user authorisation. While this is an understandable situation when a service is growing and evolving, it will eventually become too difficult to maintain and other models are needed.

Moving from local AAA to a more distributed direction, a second option is to do both authentication and authorisation centrally, with services only abiding by the given permissions. In the case of user-specific access controls, this model might make it hard to maintain more fine-grained object permissions, but with role or attribute based access control (RBAC, ABAC) introduced in Chapter 2 it offers a single point for permission management and policies.

A compromise between the two models of everything local or everything centralised, is to do authentication centrally and keep object permissions within each service. This enables support for more fine-grained user-specific object permissions, but makes centralised permission policies harder to implement. Even though RBAC or ABAC cannot be centrally implemented with this model, it might seem tempting as adding new permissions to a single service can be more easily done.

The fourth model is to have centralised authorisation and localised authentication, with each service validating the user on its own, but fetching access rights from a centralised location. This model is arguably the worst of the four, and is only mentioned because of completeness. As authentication is done locally, each service would need to keep track of local user database, but also maintain a relationship with a centralised rights database. This would lead to a lot of duplicate code and would allow new attack vectors if any of the authentication implementations were to be faulty.

Judging from a legal standpoint, the second approach might seem the most feasible, as AAA is done centrally and user access to each service endpoint is validated before the request is allowed to the specific service. While bugs and misconfigurations in the central AAA service could be fatal, this reduces the attack surface as other services are more protected from harm.

With either centralised authentication or authorisation, system needs a separate gateway to vet requests before passing them onwards so the service can be certain that provided details of the user or her rights are not fabricated. This can also be achieved with cryptographic checks, as is done with JSON Web Tokens presented in Chapter 6.2.1, but a gateway to check incoming requests can provide value with accounting and other tasks in these cases as well. The related API gateway model is presented more thoroughly in Chapter 8.

## 6.1. Traditional Mechanisms of Distributed AAA

As we have mentioned before, AAA is not a new theme in distributed environments. Earliest technologies were originally created in 1980s, and even though they have seen development their architecture has not changed much. We are presenting a few that are still in use today and provide time-tested AAA to the systems they are guarding.

### 6.1.1. Simple Authentication Security Layer

The *Simple Authentication Security Layer* (SASL) was originally created by John Gardiner Myers of Carnegie Mellon University in 1997 as RFC 2222, and updated in 2006 [RFC4422]. SASL allows a whole variety of services to be used as authentication endpoints with the same protocol, effectively decoupling authentication from application protocols with the provided abstraction. This enables server software to use many different authentication backends without implementing each of them separately, just by supporting SASL authentication.

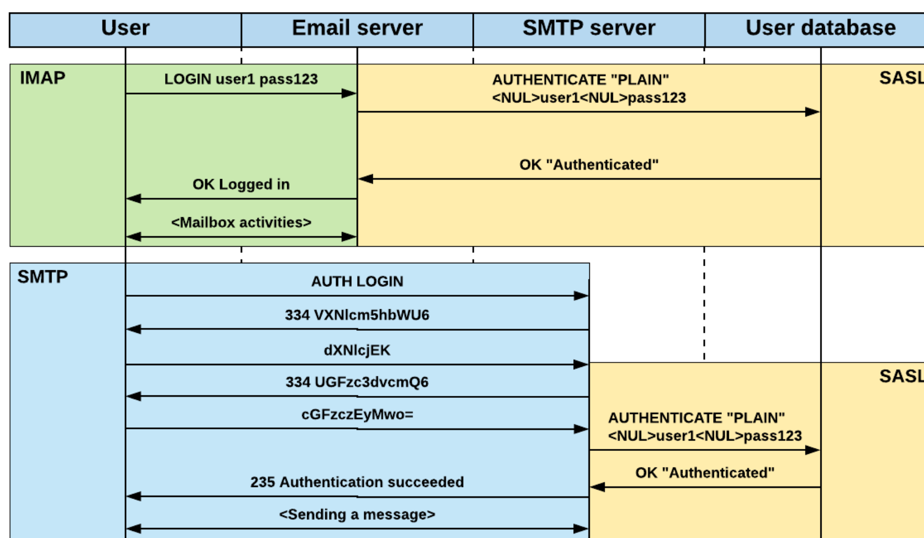


*Figure 12: IMAP and SMTP with SASL back end login*

SASL authentication can be established by a trusted service that processes user credentials and replies with results. One example of a common use case for SASL is a distributed email system, presented in Figure 12. Within the system, each component needs to verify user credentials when processing mailbox logins with *Internet Message Access Protocol* (IMAP) or sending email with *Simple Mail Transfer Protocol* (SMTP). With this setup, all email servers can share an authentication backend where login credentials are queried with SASL before user is allowed to access resources.

### 6.1.2. Kerberos

Kerberos is one of the earliest authentication protocols that is still widely used in many distributed systems and computer organisations. Originally developed within Massachu-

setts Institute of Technology, its first non-internal version 4 was published in the late 1980s, and version 5 eventually became a standard as RFC 1510 in 1994 [KNT94]. The standard was made obsolete in 2005 with the renewed RFC 4120, but it still maintains backwards compatibility for the older Kerberos version 5.

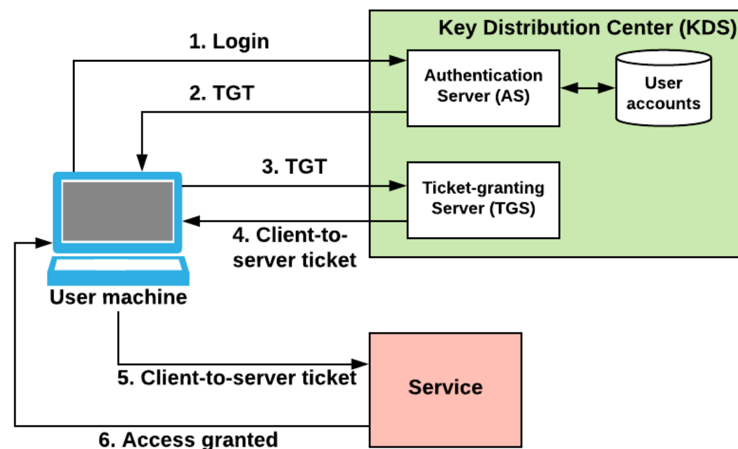


*Figure 13: Kerberos authentication to a single service. As long as the TGT remains valid, only steps 3-6 need to be done for other services.*

Kerberos operates on tickets to allow nodes to prove their identity without trusting each other with the help of a trusted *Key Distribution Center* (KDC), shown in Figure 13. Other Kerberos-authentication-related objects are the *Authentication Server* (AS) and the *Ticket-Granting Service* (TGS) that are both often found from the same KDC. Kerberos client transmits the user's id to the AS, that sends back a session key encrypted with the users password or public key, along with an encrypted *Ticket Granting Ticket* (TGT) to be used in the next step. If client has the correct password or key for the user, it is able to decrypt the *Client/TGS Session Key* and use that to decrypt the TGT from the message.

After validating the user, client sends a request to TGS with the ID of the service it wants to authenticate with, along with TGT. After decrypting the TGT with previously agreed Client/TGS Session Key, TGS sends back encrypted Client-to-Server Ticket including client information and a newly created *Client/Server Session Key* that client needs to use to communicate with the service. With the Client/Server Session Key and Client-to-server ticket (that was encrypted using the service's secret key), service is able

to ascertain identity of the user that client is representing and can provide service to the client. On the other side, connecting client can also be sure of the server's identity. As long as TGT has not expired, it can be reused to get Client-to-Server Tickets for other services as well.

Because Kerberos has a trust model based on symmetric key cryptography, it has been able to withstand the test of time with updates to the cryptographic algorithms from DES, which was originally used, to for example AES and other more secure algorithms. Similar updates can keep Kerberos alive in the future as well. Another aspect that helps keep Kerberos model current is the usability of the model in an automated manner. As there is no *Single Sign-On* (SSO) endpoint for the user to interact with, systems can authenticate and communicate between each other without a human operator with just stored credentials. With Internet of Things and distributed systems, Kerberos will likely retain its role as trust provider for numerous different services.

### 6.1.3. Security Assertion Markup Language

*Security Assertion Markup Language* (SAML) is a standard and an XML-based markup language for exchanging authentication and authorisation data between identity and service providers [HEL05]. One notable SAML-based entity is the Shibboleth framework that is used by numerous public service organisations and universities across the world. Login functionality is shown in detail in Figure 14.

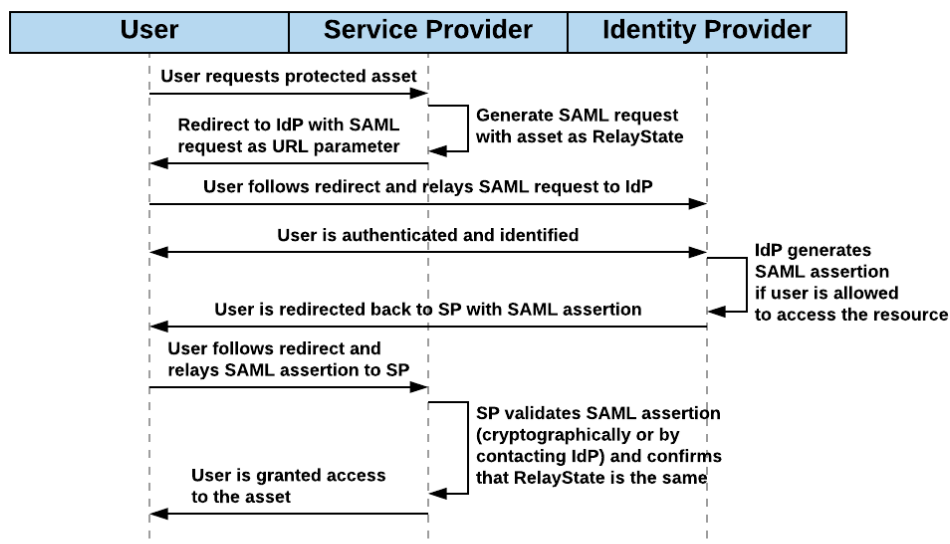


*Figure 14: Login to a service through SAML single sign-on (SSO)*

The SAML authentication begins with the service provider redirecting the user's browser to the *identity provider* (IdP), along with an assertion XML document defining the requested action. SAML assertion contains the requested action, for example an AuthnRequest when authenticating users, as well as the information about the issuer and RelayState to point to a resource, as is done in Figure 14. After authentication, IdP redirects the user back to the service with the SAML response XML document included in the redirect. Response contains original ID that was sent from the service in a SAML assertion, as well as other details that can be used when the response is verified from the IdP. Other possibility for verification is to include a cryptographically signed token (for example, a JSON Web Token as explained in section 5.2.1) in the response that can be used to validate the response without contacting the IdP, but by just knowing its public key.

## *6.2. New distributed authentication & authorisation models*

While the models presented in section 5.1 have existed for quite some time, distributed systems themselves have evolved in many ways. Previously distribution happened mostly with centralised account management, with accounts separately accessible from the systems providing the service. Newer development has seen the rise of microservices, where a single system can consist of tens or even hundreds of small services. Many of these can process user information and need to independently make decisions whether to give out requested data or to deny the request. As such services can operate in another datacenter or even in another continent, calling home and asking for confirmation is not always an option. In this chapter we present technologies that are well suited for distributed use cases found in modern applications.

### 6.2.1. JSON-based standards

The *Internet Engineering Task Force* (IETF) published a set of JSON-based standards in 2015, including *JSON Web Signature* (JWS), *JSON Web Encryption* (JWE) and *JSON Web Token* (JWT) [Jon11]. Together they present a group of standards usable with the Javascript API, making it approachable for browsers and server software alike. JSON Web Token is a prime candidate for distributed authentication, as it can be signed

by the authentication service and validated by other services with its public key. In the new 5G standard, JWTs are currently specified to be used as OAuth 2.0 access tokens [GPP18].

In addition to just a username, JWT can hold all kinds of information, including token signing time, expiration time, user roles and all sorts of private claims that need to be agreed on only by the token issuer and consumers [RFC7797]. Figure 15 illustrates JWT structure that consists of three different parts: the header that includes a signing algorithm and tokens type, the payload that includes the data and the signature that can be used to validate JWT's integrity and its signee, with either a shared secret or by its public key, depending on what algorithm has been used in token creation.

| | Encoded | Decoded |
|---|---|---|
| **Header** | eyJhbGciOiJIUzI1NiI<br>sInR5cCI6IkpXVCJ9 | {<br>"alg": "HS256",<br>"typ": "JWT"<br>} |
| | . | |
| **Payload** | eyJzdWIiOiIxMjM0NTY<br>3ODkwIiwidXNlcm5hb<br>WUiOiJ1c2VyMSIsImlhd<br>CI6MTUxNjIzOTAyMn0 | {<br>"sub": "1234567890",<br>"username": "user1",<br>"iat": 1516239022<br>} |
| | . | |
| **Signature** | SflKxwRJSMeKKF2QT4fwp<br>MeJf36POk6yJV_adQssw5c | |

*Figure 15: A JSON Web Token in encoded and decoded form.*

As a single mechanism for both authentication and authorisation, plain JWTs do not necessarily make for a secure mechanism, since authorisation might be passed with the signed token, making it impossible to change privileges of an already-issued token. For just authentication, with each service doing authorisation JWTs are easy to use and offer a good solution for a secure authentication layer.

For centralised authorisation, it is also possible to use authorisation-included JWTs only within the system, with an API gateway fetching a new token from the authorisation service for each request. This model also adds an extra check before any request reaches the service, making it harder for an attacker to take advantage of signed but yet revoked tokens. Unfortunately, increased security also adds extra complexity to the system, making the authorisation service a single point of failure even when the user has already

logged in. Because of this, the refresh token model presented in section 5.3 with OAuth 2.0 and OpenID Connect might be a better alternative.

### 6.2.2. OAuth 2.0

*OAuth* is a standard for delegating access to application to use user information stored in other service. A good use case for OAuth would be Gmail's contact list that is required by Facebook to show which of your contacts are already using the service. Access to the contact list is gained through service APIs. After authorisation, the application can use specific parts of the service on users' behalf once or until the access is revoked.

OAuth standard was originally born in 2007, and published as informational RFC 5849 in 2010. First implementations were created by Twitter, with Google and others following soon. However, the standard was not deemed adequate for non-browser-applications and it was too complex for simple use. *OAuth 2.0 framework*, published as standard in 2012 [RFC6749], responds to these problems by simplifying many aspects of the original OAuth, a now obsolete RFC 5849.

OAuth 2.0 has different grant types for specific authorisation use cases, such as single-page apps that cannot have application secrets, or even password-specific grant type for authorisation services own applications. Most common usage is for a normal service that can store application secrets in its own database and communicate with the authorisation endpoint without visibility to the user. This grant type is known as *Authorization Code*, that can also be used without secrets in single-page apps or mobile applications.

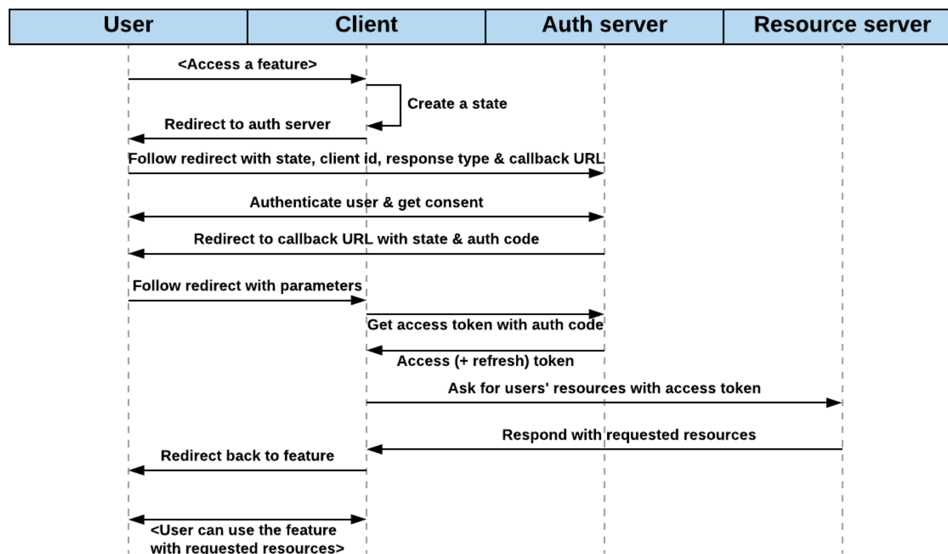


*Figure 16: OAuth 2.0 Authorization Code Grant*

Authorization Code Grant, shown in Figure 16, starts with the user being redirected from service to the authorising service's auth endpoint with specified response type and unique client id of the service or application. Also included are redirect URL informing where the user should be returned when done, scope indicating the parts of user's information service wants to access and state, which is a random string that is expected to be returned when the authorisation is done. Authorising service (after logging in, if no session exists) then presents the user an authorisation prompt with scopes needed by the service. After accepting the prompt, user is redirected back to the original service along with the same state string that was passed to the service, and an auth code, a one-time code that the service will use to get the access token in order to use the resources of the authorising service. Depending on the flow, token exchange can include a client secret that is known only to the service requesting the token, not to the user or browser. In single-page or mobile apps, the secret can be left out, but this needs to be defined when the application is initially registered with the authorisation service.

*Implicit Grant* is a simplified version of the Authorisation Code Grant that was previously recommended for single-page JavaScript applications and other software that do not necessarily have a backing service running and keeping secrets from the front end application itself. In Implicit Grant flow, access token is received from the authorisation service without separate token exchange using the one-time key given in redirect. While

this was saving an additional HTTP request, it has problematic security implications that can allow an attacker to steal the access token more easily. As the token is transferred as part of the redirect URL, browsers cache it into their history and it becomes easier for the attacker to steal the token. Other problematic feature of the Implicit Grant is the lack of 'state' variable found in Authorisation Code Grant that needs to be the same throughout the authorisation flow, making brute force attacks harder. Because of these, current recommendation for single-page applications and mobile applications is to use the Authorisation Code Grant without application secret, where state is preserved through the authorisation flow and no tokens are passed in redirect URLs.

Different tokens between client and server are also an issue. Newly upgraded Best Current Practice for OAuth 2.0 suggests using the standard approach to access tokens by separating them to a specific refresh token with even years of lifetime and to a more short-lived access token, that is used to access the service. This approach is shown in Figure 17 below. Refresh token is used by the application to get the access token from the service's token endpoint, and then accessing the provided resources [RFC8252].

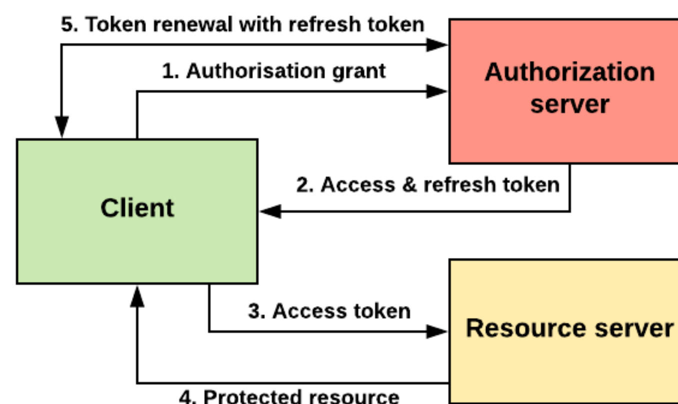


*Figure 17: Refresh & access token usage with authorisation server.*

OAuth 2.0 has become *de facto* standard for API access delegation and is currently widely used for this purpose. While OAuth 2.0 is an authorisation protocol, it has also been used for authentication. This is done by authorizing the user with only user's information in authorisation scope, and then fetching the user identity from the authorisation provider. This is known as pseudo-authentication and OAuth 2.0 should not be mis-

taken to be an authentication protocol, as identity might not be as tightly coupled with the API access to certain services given with OAuth 2.0 authorisation. For example, if the checked scope is user information, this might also be available to an administrator of the authorisation service and it could be used to gain access to another service. Other possible attack routes have also been presented and fixed [SuB12].

OAuth 2.0 can also be combined with JSON Web Tokens. In a 2015 abstract extension, JWTs for authorisation grants and client credentials were described in a standard [RFC7523]. As client credentials can be described in JWT format, browser could authenticate and authorise against an OAuth 2.0 endpoint without user needing to do anything if the JWT is signed by a trusted party. When combined with the Web Authentication API currently developed as an experimental feature in browsers, future end users could sign in and authorise services with their browser-stored JWTs without typing passwords. JWTs as OAuth 2.0 tokens also help making the services more secure, as the application can be sure of the token-issuers identity with its public key as tokens have been signed. In the new 5G standard, currently in development, JWT's will be used as access tokens with the *Client Credentials Grant* [GPP18].

### 6.2.3. OpenID Connect

*OpenID Connect* (OIDC) [SBJ14] is an authentication layer built on top of the OAuth 2.0 specification. OIDC uses JSON Web Tokens (JWT) to represent identities and authentication happens using similar authentication flows as OAuth 2.0 has: authorisation code, implicit flow and hybrid flow. As OAuth 2.0 is already in wide-spread use, creating OIDC-compatible authentication flows is relatively easy as developers already have some knowledge of OAuth 2.0. OIDCs *Authorization Code Flow* is similar to OAuth 2.0's one presented in Figure 16. Different OIDC flows and their properties are illustrated below in Figure 18.

Similarly to OAuth 2.0, OIDC also supports refresh tokens, but access tokens are known as id tokens, since in OIDC specification access tokens are for authorisation decisions made by resource servers. With both refresh and id tokens being signed JWTs, authentication to separate service endpoints happens easily as each endpoint can independently assess the validity of identity tokens representing the user. OIDC JWTs can also contain information of users' roles, making role-based access control (RBAC) an easy approach

even in distributed contexts.

| Property | Authorization Code Flow | Implicit Flow | Hybrid Flow |
|---|---|---|---|
| All tokens from authorization endpoint | no | yes | no |
| All tokens from token endpoint | yes | no | no |
| Tokens not revealed to user agent | yes | no | no |
| Client can be authenticated | yes | no | yes |
| Refresh token possible | yes | no | yes |
| Communication in one round trip | no | yes | no |
| Most communication server-to-server | yes | no | varies |

*Figure 18: OpenID Connect flow comparison [SBJ14], coloring added for the thesis.*

Authorisation Code Flow is almost an exact copy of the OAuth 2.0 one. When authenticating, user is given a one-time code that is passed back to the application (client), which then uses the one-time code, along with its client secret to fetch id token to get the logged in user's identity. Additionally, a refresh token is given to the application to renew the shorter-lived id token when it expires.

*Implicit Flow* is meant for applications that do not necessarily have a back end server to communicate with, but need to authenticate in order to communicate with one or numerous API endpoints. Similar to OAuth 2.0's implicit grant, no client secret is used and ID token is stored within an application (desktop or mobile) or in the browser. Implicit Flow is much lighter than Authorisation Code Flow, but carries security risks similar to its OAuth 2.0 paragon that are described in the above chapter.

*Hybrid flow* is more rarely used, but it still exists as part of OIDC. In Hybrid flow, both the front end and back end of the application can receive their own ID tokens, with front end receiving it from the *identity provider* (IdP) when logging in, and back end by using the passed one-time code and client secret similar to Authorisation Code flow. One use case for this is when we want to pass roles and other information to the back end application in the ID token that we do not want to be visible to the user. Other possibility for this is to use encrypted JSON Web Token, but as it isn't covered in the OpenID Connect specification, we do not explore this further.

### 6.2.4. Macaroons

Similar to JWTs, Macaroons constitute a proof-carrying authorisation scheme where an access token can be modified and restricted further after issuance, include validation re-

quirements from 3rd party service and be passed along to other services using the issuing service on behalf of the user [BPE14]. Macaroons include contextual caveats that can be used to limit the given scope to what the token can be used for. This is similar to vallet keys found in more expensive cars that allow them to be driven for a limited amount and with reduced speed for parking the car, but cannot be used to fully drive the car when compared to what can be done with an actual key.

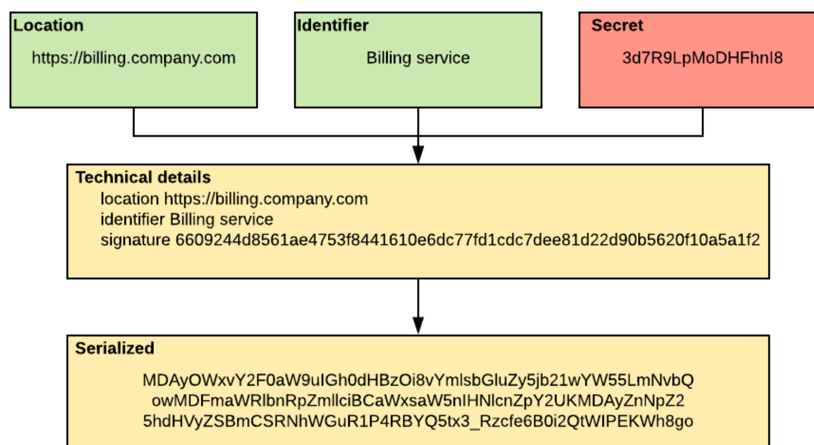


*Figure 19: Example Macaroon for a billing service.*

As shown in Figure 19, Macaroon consists of a secret string known only by the creating service, location indicating where the Macaroon should be used and an identifier that can describe Macaroon's scope, for example "Billing service". Macaroons are based on a hashing function, but can also be signed with private keys and validated with public ones, similarly to client certificates. Most interesting properties of Macaroons are first and third party caveats that can be layered on top of the original Macaroon. For example, initial Macaroon is hashed with the secret variable known only by the issuer. Each layer on top of the initial Macaroon, containing caveats such as username or validity, is hashed with itself, and issuer can easily verify each addition as well as the original token when the initialisation vector is known. With key signing, others can also validate the Macaroon if public keys are known.

Third party caveats specify that Macaroon cannot be trusted unless it is satisfactory for a third party, for example, an identity service such as Google or Facebook. To satisfy the requirement, user must log in to the service and have it sign the token. In a web browser environment, this can be done automatically as user is often already signed in to the ser-

vice within the same browser session.

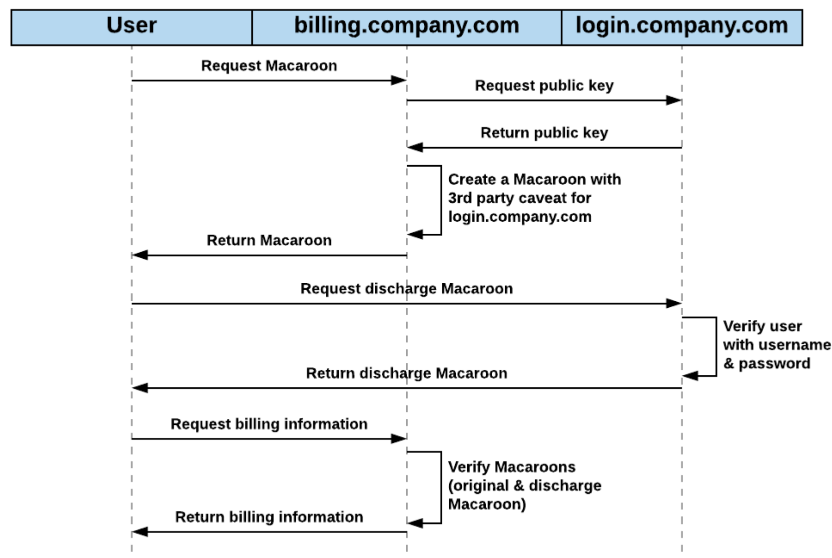


*Figure 20: Macaroon usage with third party caveats.*

Macaroons are not an authentication or authorisation framework such as OAuth 2.0 or OpenID Connect, but can be used to harden them by substituting the created access tokens with Macaroons [BPE14]. This allows an authorisation service to delegate authentication to a third party service more easily, as is shown in Figure 20, and removes the worry of revocation as identity endpoint can issue Macaroons for 10 seconds and almost every new request would require client to ask for a newly signed Macaroon. As Macaroons default hashing is based on the HMAC function, this operation does not require much computing power and verification can thus be performed without much computational overhead.

# 7. Communication Mechanisms Between Microservices and Distributed Systems

As distributed systems and microservices perform most of their communication between remote parts, efficiency of that communication is essential. Latency can easily become a bottleneck if any call to a system generates more calls to other systems, with each sub request adding up to a time it takes to resolve the original request. While HTTP-based methods have not disappeared from intra-service communication landscape, they have gotten company from a whole family of other communication standards and paradigms. In this chapter, we attempt to present a variety of both synchronous and asynchronous methods for achieving efficient and trusted communication between separated system components.

When speaking of asynchronous and synchronous communication, it is important to understand that we are not discussing programming language semantics, where a language allows an operation to be handled asynchronously while doing other tasks. With asynchronous we mean, in this context, events that are not necessarily fully completed when the remote request is finished, such as pushing a message to a queue, where messages are picked up and processed later. Whether the later processing happens within a second or within a few hours makes no difference.

## *7.1. Synchronous Mechanisms*

Synchronous mechanisms offer an immediate response on the performed remote action. Traditionally, all communication between systems and system components has happened in a synchronous manner.

Many events in systems require synchronous processing. For example, a bank transfer from an account to another requires that events on both accounts, withdrawal from one and deposit to other, are completed. Otherwise money will either disappear or come into existence from nothing. With synchronous communication, requesting party can be certain that the event was processed before proceeding.

### 7.1.1. Remote Procedure Calls

*Remote Procedure Call* (RPC) was the first widely accepted idea of remote communication between processes [TaV07]. First designed to allow programs to run on different computers without sharing memory, it evolved into *XML-RPC* in 1998 for the purpose of functioning on top of HTTP. More recent application of the RPC paradigm is *JSON-RPC*, originally from 2005.

The RPC model, as well as its XML and JSON variants, consists of application calling its own conversion stub that packages its request to a specified format and sends it to a remote server for processing. Once remote server has performed the task, its stub passes it back to the calling client's stub and execution continues. An example of this is shown in Figure 21.

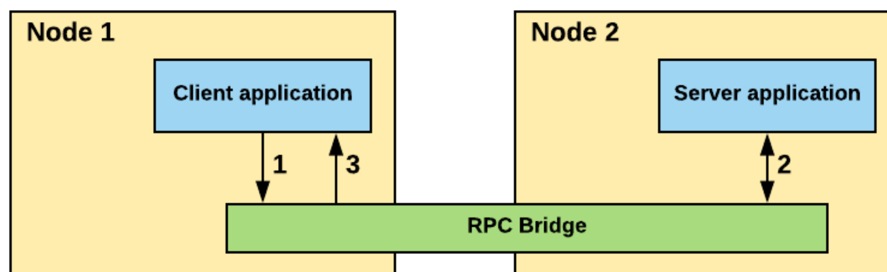


*Figure 21: Example of RPC call over a network. Execution continues when steps 1, 2 and 3 are completed.*

During the current decade, synchronous RPCs were mostly forgotten and thought to be a dying breed that would eventually be replaced completely by *RESTful APIs*, but recent development has proved otherwise. In 2015, Google published its internal RPC framework known as Stubby, under a new name, *gRPC Remote Procedure Call* (gRPC) [GM-D15] for both asynchronous and synchronous message passing. The gRPC framework has since become part of the *Cloud-Native Computing Foundation* (CNCF) and is considered a key framework in internal communication of distributed systems.

```
message Person {
  string name = 1;
  int32 id = 2;
  bool is_admin = 3;
}

service Messaging {
  // Sends messages between people
  rpc SendMsg (StandardMessage) returns (StandardMessage) {}
}

message StandardMessage {
  string message = 1;
}
```

*Figure 22: gRPC protocol buffer describing a person and communication between people*

gRPC as an implementation framework allows clients written in different languages to communicate with each other using a Google-oriented way to serialize data structures known as *protocol buffers* that are presented in Figure 22. In this regard, gRPC is similar to XML-RPC, but with smaller and more efficient packaging and native language support, making its use more tempting.

### 7.1.2. HTTP-Based Methods

*Representational State Transfer* (REST) architecture was defined in 2000 as part of the PhD dissertation by Roy Fielding [Fie00]. REST uses HTTP properties to define architectural constraints that require APIs to represent their functionality by using base URLs, such as api.company.com/users to represent user service, api.company.com/billing to represent billing etc. Elements are represented under collections, for example api.company.com/users/user1 that covers a specific user. In addition, APIs conforming to the RESTful specification require the use of standard HTTP methods to create, read, update and delete objects in collections. Possible actions for a single item are shown in Figure 23.

| Action | Path | Method |
|---|---|---|
| Create | https://api.company.com/item | POST |
| Read | https://api.company.com/item/3 | GET |
| Update | https://api.company.com/item/3 | PUT |
| Destroy | https://api.company.com/item/3 | DELETE |

*Figure 23: HTTP API endpoints for manipulating items*

RESTful APIs have become a standard way of representing API resources to outside users, but their usage is also increasing within distributed systems and especially within microservices. As services can allow several different functions to other services within the system, it is easy to represent them with RESTful semantics, making it easy for developer of another service to use the needed implementation available.

| GraphQL | REST |
|---|---|
| **GET https://api.company.com/graphql**<br>{<br>user(id: 12345) {<br>name<br>email<br>friends {<br>id<br>name<br>}<br>}<br>} | **GET https://api.company.com/user/12345**<br><br>**GET https://api.company.com/user/12345/friends** |

*Figure 24: REST queries are smaller, but GraphQL allows requesting specifically what is needed*

Recently, RESTful APIs have seen a contestant as *GraphQL* was published by Facebook in 2015, with the latest working draft towards a GraphQL standard from October 2016 [Gra16]. GraphQL is a query language and type system that offers a developer-friendly way to search for data provided by the API and to easily mock queries using the provided web user interface to receive only the needed parts from the API, as is shown in Figure 24. GraphQL does not provide similar semantics with URIs as RESTful APIs do, but it offers same functionality through its own query language. Currently it is most often used as a read-only API for data, but this might change in the future as its usage grows and GraphQL implementations and its use cases become more tested and hardened in terms of security. As internal usage does not have as strict security requirements as something that is open to the world, GraphQL can more easily become trusted as an internal component within a system.

## 7.2. Asynchronous mechanisms

Asynchronous mechanisms offer systems a possibility to perform non-time-critical actions when resources allow. They also allow the system to scale more easily when more

requests are coming in as events are not failing due to more stringent timeout expectations often found in synchronous communication.

In message queue -based communication, described in 6.2.1. and 6.2.2., message sender (publisher) will publish the message into a broker-maintained message queue. Message recipients (consumers) are meanwhile subscribed into the queue with the broker, and receive the message when it is published with the broker. Message queues support point-to-point delivery where message is strictly from a single party to another, as well as the publish/subscribe model where a single message queue can be shared with one or more publishers and consumers [EFG03]. Consumers in publish/subscribe model are often separated into listeners that only receive messages from a queue when they are connected to it, and into consumers that also receive the queued messages that the broker received when they were not connected. The broker-centric model, also known as loose coupling, allows system components to operate without the knowledge of each other and their addresses, as all communication is done through a message broker system [TaV07]. This is illustrated in Figure 25.

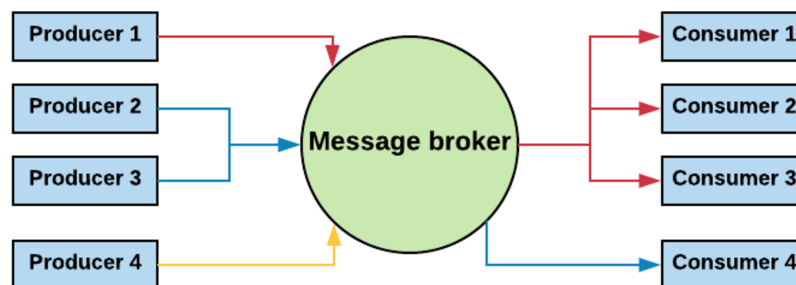


*Figure 25: Publish-subscribe with broker. Topic produced by producer 4 currently has no consumers.*

*Publish/subscribe* model has enabled a generation of new fault-tolerant microservice systems, where parts of the system can be taken down for maintenance or replacement without other parts being affected. The amount of data that is produced and consumed has exploded in the recent years, and as the trend grows, data streams have become a critical part of infrastructure. Progress in the development of event-based architecture is leading to the rise of the streaming platform, where message brokers are at the heart of each system [LZF17].

### 7.2.1. Advanced Message Queuing Protocol

*Advanced Message Queuing Protocol* (AMQP) is a binary protocol that was the first step in an effort to create standards in the field of proprietary messaging formats in order to allow different systems to communicate with each other [Vin06]. Adopted as an international standard in 2014 [ISO14], AMQP is a good option for communicating between systems or organisations, as it is well defined and easily scalable. Especially business systems have grown fond of it. AMQP message format is shown in Figure 26 and defines the bare message that includes the original data from publisher as an unalterable part of the message. Bare message includes optional list of standard properties, as well as application-specific properties, including the message body itself.

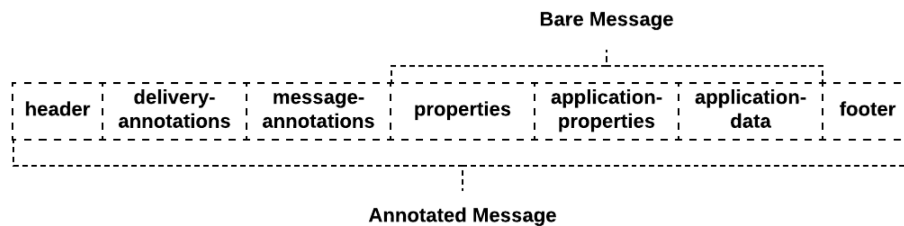


*Figure 26: AMQP message format specification [ISO14].*

AMQP is usable both with a message broker or between single system endpoints, and is quite ambiguous as a protocol in order to support multiple different scenarios. Adoption of the standardised 1.0 version has been relatively slow, but over time brokers supporting AMQP will either add support or migrate entirely to it, as it does not contain breaking changes to its pre-standardised versions.

### 7.2.2. Message Queue Telemetry Transport

*Message Queue Telemetry Transport* (MQTT) is another standardised messaging protocol, built specifically to handle working with the publish/subscribe -based messaging architectures [ReD17]. Originally created by IBM, MQTT was submitted to OASIS standardisation body and became a standard with version 3.1.1 in 2014 [OAS14]. The MQTT protocol with its compact message sizes is a perfect candidate for small sensors and other Internet of Things devices where resources and network bandwidth are often scarce. When compared to AMQP, MQTT protocol is more suitable for pushing mes-

sages to a processing network, but MQTT has a clear scope of being a client-server protocol. Therefore, AMQP is better suited for point-to-point communication without external brokers.

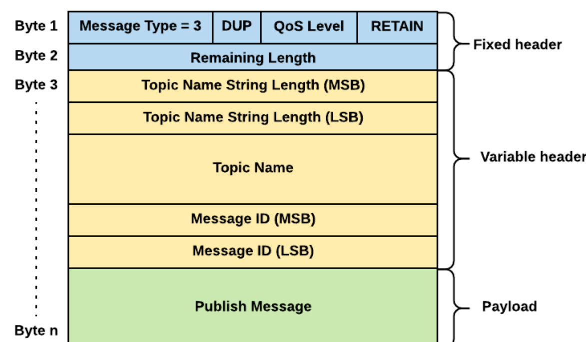


*Figure 27: MQTT control packet message format for type 3 (PUBLISH) messages [OAS14]*

The MQTT protocol consists of control packets, shown in Figure 27. Message type specifies the attempted operation, with 3 being a type for publishing a message. Other message types include opening a connection, as well as acknowledging a received message. MSB and LSB in Figure 27 stand for Most Significant Byte and Least Significant Byte.

In contrast to AMQP, the MQTT protocol does not scale so easily within single message queues, but passing messages between entities close to each other and onwards to processing platforms is a well supported use case. When we consider a modern data analysis platform where endpoints are not just publishing data, but instead they are also subscribers to different data streams, both MQTT and AMQP might be needed in order for the platform to work efficiently.

### 7.2.3. Message Broker Specific Protocols

As the publish/subscribe model gains popularity, the number of different message brokers available has also increased. Most brokers adhere to existing standards and often support either MQTT, AMQP or even both of them. Some message brokers have defined their own protocols that have become relevant as the popularity of the broker has grown. Two of these are Apache Foundation's Kafka and Pulsar.

Kafka was originally a software created within LinkedIn, and was open sourced in 2012. Currently developed by the Apache Foundation, Kafka has grown to be the most used

message broker due to its almost endless scalability as it can handle almost a million events with just a single consumer. Kafka has its own binary protocol that works over TCP. Kafka's protocol supports messages of different sizes, arrays as well as strings. Though proprietary, the protocol is well documented and has well-supported client libraries available in almost every programming language.

Newer contestant in the broker field is another Apache Foundation project, Pulsar. Currently in incubator stage, it originates from Yahoo and has seen years of production usage inside the company, and is rising quickly in usage. Pulsar has two different protocols available: its own binary protocol using *protocol buffers* (Protobuf) and a separate protocol using standard W3C websockets [RFC6455]. While Pulsar is a much younger project than Kafka, its websocket protocol helps in adoption as the standard can be used on any existing programming language with websocket support available. Some clients already exist for Pulsar's own protocol as well, with many more in the roadmap, and with growing adoption, Pulsar's Protobufs, presented in Chapter 7.1.1 in Figure 22, might very well be the next big proprietary messaging protocol.

# 8. Authentication, Session Management and Identity Propagation for Microservices

The API gateway is a great tool in controlling access to different microservices. Ambassador [Amb17] is one such gateway that can be used to protect services running in the Kubernetes container orchestration platform [Kub14]. Ambassador works together with other components using SPIFFE, providing trusted communication environment for Ambassador and other resources running in the platform. Besides acting as a load balancer for back end services, Ambassador offers support for external authentication modules that can be used in incoming request validation. To demonstrate the power of scalable session management and support for user accounts stored elsewhere, we created, as part of the thesis work, an external authentication service for Ambassador using OpenID Connect and JSON Web Tokens. Code for the authentication service can be found from its repository [Myy18], with functional code also included as attachment in Chapter 11.

The authentication service, named simply *Ambassador-Auth-OIDC* (AuthService) checks requests for cookies that contain JWTs and if such cookie exists, also checks the embedded JWT for validity. AuthService is written in the Go programming language for fast operations and JWTs are signed using the HMAC-SHA-512 algorithm [RFC7519], making it possible to validate them if the secret is known. AuthService also supports login through OIDC with Authorisation Code Flow and logout functionality.

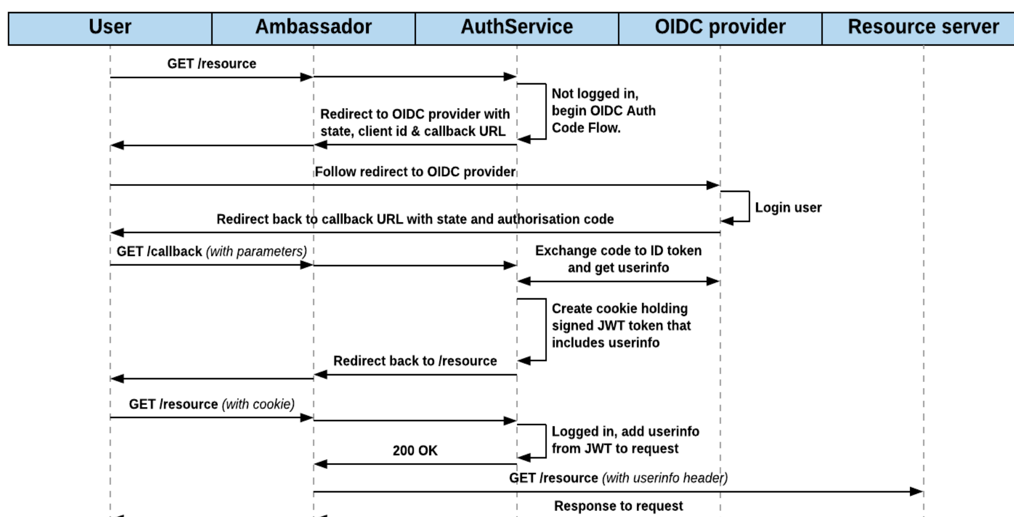


*Figure 28: Request flow from an unauthenticated user through Ambassador, AuthService and OIDC provider*

Figure 28 represents an unauthenticated request towards a protected resource. As the user is not logged in, OIDC Authorisation Code Flow is used to login the user and her userinfo is stored into a signed JWT that is included in user's HTTP cookie. User is then redirected back to the original resource and cookie stored in browser is attached to the request. AuthService then validates the JWT within the cookie and gives Ambassador permission to proceed with the request that is sent to the resource server.

AuthService uses an external key-value storage to store OIDC mid-login states and revoked JWTs, so any running instance can finish the authentication flow. The storage is however not used on request validation and AuthService only syncs its revocation blacklist with it each minute, fetching JWTs that other AuthService instances have revoked. Separating key-value storage from AuthService's operation allows more than one AuthService to be in use at the same time and allows multiple API gateways with fast stateless request validation to be geographically distributed. This design choice also helps with the service scaling to thousands or even millions of simultaneous requests as the key-value store would easily become a bottleneck. It also removes additional latency as JWTs can be validated without network requests.

A common pitfall with JWTs is revocation. As JWTs themselves are stateless, often suggested solution is to forget revocation by issuing tokens with short lifetime and checking session status each time a token is renewed. This requires more work in the browser-side and if device is sleeping in between, JWT has often expired and login needs to be done again. AuthService mitigates this by using longer lifetimes for JWTs and instead controls a revocation blacklist that contains blacklisted tokens until their validity expires. Each request is checked against the local blacklist and the local blacklist is synchronised from the central key-value store once every minute, so revocations done by other AuthService instances are synced to every instance. Reason for not storing the blacklist only in the key-value store and syncing it locally is the same as the reason for using JWT's in the first place: local validation for all incoming requests without additional latency caused by querying the key-value store.

Each AuthService endpoint can easily process almost a thousand requests each second. As they can be horizontally scaled with additional copies, AuthService will not become a bottleneck even with a high number of simultaneous users.

# 9. Conclusions

Many technologies exist for implementing authentication, authorisation and accounting for distributed systems and microservices. This thesis has presented several current and also emerging technologies in this field and also given a concrete example of their use for the Ambassador microservice API gateway. JSON Web Tokens and OpenID Connect's Authorisation Code Flow were used in the authentication component to implement a stateless session handling and a secure way to authenticate users.

As systems, technologies and networks are evolving in ways that we often cannot predict, we need a versatile approach for AAA in order to securely support a wide variety of different scenarios. Public-key cryptography is likely to be a key part of any future AAA scheme, both in protecting communication as well as in message verification. The use of JSON Web Tokens and OAuth 2.0 in the new 5G mobile network standard [GP-P18] shows that these technologies will be around for a while, even if new technologies are certainly developed and current ones upgraded.

Another key aspect in development of new AAA technologies is their ability to scale. As a link posted to Twitter can go around the world in minutes, the amount of users a system serves can rise fairly quickly from a few to hundreds of thousands. The AAA must not become a bottleneck in any environment and it must scale along other system components.

The streaming platform paradigm, presented in Chapter 7.2, enables future growth with support for millions of simultaneous messages. Loose coupling in microservices gives services freedom to be down without affecting the whole environment, also making it possible to do rolling upgrades without any visible changes to users.

Newer technologies for modern AAA, presented in Chapter 6 and shown to be feasible in Chapter 8, are already key parts of world's most used services and the adoption of such new technologies will likely increase in the future. They also provide good building blocks for new authentication and authorisation frameworks as well as for communication models for internal and external communication of distributed systems.

The work around distributed AAA is nowhere finished and as systems and computers evolve, so must both architectural patterns and algorithms used. One, perhaps a bit more

distant issue will be quantum computing. As computational power grows exponentially, algorithms used to safeguard communication and authenticate users must be switched to quantum-safe ones, preventing anyone from calculating the secrets used, for example, in Diffie-Hellman, TLS connections or JSON Web Tokens.

In a more nearer future more work will be needed in service meshes. As we're moving from securing internal networks by securing their outside connections, the zero trust networking paradigm [GiB17] is suggesting that even such networks should have their communications completely secured. The SPIFFE framework, presented in Chapter 5.2.2, is a step in the right direction, but more work is needed, for example on initialisation of trust when a new device or user joins the network.

# References

[AKB01] Andersen, H., Kaashoek, D., Balakrishnan, M., Morris, R., Resilient overlay networks, *Proceedings of 18th ACM Symposium on Operating Systems*, October 2001, pp 131-145.

[Amb17] The Ambassador API gateway, Datawire, https://www.getambassador.io/ (visited 23.8.2018).

[BPE14] Birgisson, A., Politz, J. G., Erlingsson, Ú., Taly, A., Vrable, M., Lentczner, M., Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud, *Proceedings of Network and Distributed Systems Symposium '14*, February 2014.

[DVW92] Diffie, W., Van Oorschot, P., Wiener, M., Authentication and authenticated key exchanges, *Designs, Codes and Cryptography*, issue 2, June 1992, pp. 107-125.

[EFG03] Eugster P., Felber P., Guerraoui R., Kermarrec A-M., The Many Faces of Publish/Subscribe, *ACM Computing Surveys*, volume 35, June 2003, pp. 114-131.

[Ell99] Ellis, J. H., The History of Non-Secret Encryption, *Cryptologia*, volume 23, issue 3, Taylor & Francis, January 1999, pp. 267-273.

[Fie00] Fielding, Roy Thomas, *Architectural Styles and the Design of Network-based Software Architectures*, PhD dissertation, University of California, Irvine, 2000, https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (visited 20.5.2018).

[GiB17] Gilman, E., Barth, D., *Zero Trust Networks*, O'Reilly Media, June 2017, ISBN 978-1-491-96219-0.

[GMD15] gRPC Motivation and Design Principles, Google Inc, September 2015, https://grpc.io/blog/principles (visited 20.5.2018).

[GPP18] *Security architecture and procedures for 5G System*, technical specification 33.501, release 15, version 15.1.0, 3rd Generation Partnership Project (3GPP), June 2018.

[Gra16 ] Draft RFC Specification for GraphQL, Facebook Inc, October 2016, http://facebook.github.io/graphql/October2016/ (visited 26.5.2018).

[HEL05] Hughes, J., Maler, E., Lockhart, H., Wisniewski, T., Mishra, P., Ragouzis, N., Security Assertion Markup Language (SAML) V2.0 Technical Overview, Working Draft 08, *OASIS Open*, September 2005.

[ISO14] ISO/IEC 19464:2014, Advanced Message Queuing Protocol (AMQP) v1.0, May 2014, https://www.iso.org/standard/64955.html (visited 20.9.2018).

[Jon11] Jones, Michael B., The Emerging JSON-based identity protocol suite, *Proceedings of W3C workshop on identity in the browser*, May 2011, paper 24.

[KHD10] Karp, A., Haury, H., Davis, M., From ABAC to ZBAC: The Evolution of Access Control Models, *Proceedings of International Conference of Information Warfare and Security*, April 2010, pp. 202-211.

[KNT94] Kohl, J., Neuman, B.C., Ts'o, T., The evolution of the Kerberos authentication service, Distributed Open Systems, *IEEE Computer Society Press*, 1994, pp. 78-94.

[Kub14] Google Inc, Kubernetes orchestration platform, https://kubernetes.io/ (visited 23.8.2018)

[Lin06] Lindqvist, Håkan, *Mandatory Access Control*, Master's Thesis in Computing Science, Umeå University, SE-901 87, 2006.

[LFZ17] Liao, J., Zhuang, X., Fan, R., Peng, X., Towards a General Distributed Messaging Framework for Online Transaction Processing Applications, *IEEE Access*, volume 5, June 2017, pp. 18166-18178.

[MET99] Metz, Christopher, AAA Protocols: Authentication, Authorization and Accounting for the Internet, *IEEE Internet Computing*, issue 6, November 1999, pp. 75-79.

[Myy18] Myyrä, Antti, OpenID Connect authentication component for Ambassador API gateway using JSON Web Tokens, October 2018, https://github.com/ajmyyra/ambassador-auth-oidc .

[OAS14] OASIS Open, MQTT Version 3.1.1, Standards Track, October 2014, http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf (visited 27.5.2018).

[OSM00] Osborn, S., Sandhu, R., Munawer, Q., Configuring role-based access control to enforce mandatory and discretionary access control, *ACM Transactions on Information and System Security*, volume 3, issue 2, May 2000, pp 85-106.

[ReD17] Reddy, P., Deepthi, J., Message Queuing Telemetry Transport, *International Journal & Magazine of Engineering, Technology, Management and Research*, volume 4, issue 3, March 2017, pp. 380-384.

[RFC3647] Chokhani et al., Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practives Framework, Informational Track, November 2003, https://tools.ietf.org/html/rfc3647 .

[RFC4422] Melnikov A., Zeilenga K., Simple Authentication and Security Layer (SASL), RFC 4422, Standards Track, June 2006, https://tools.ietf.org/html/rfc4422 .

[RFC5280] Cooper, D., et al., Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, RFC 5280, Standards Track, May 2008, https://tools.ietf.org/html/rfc5280 .

[RFC6455] Fette, I., Melnikov, A., The WebSocket Protocol, RFC 6455, Standards Track, December 2011, https://tools.ietf.org/html/rfc6455 .

[RFC6749] Hardt, D., The OAuth 2.0 Authorization Framework, RFC 6749, Standards Track, October 2012, https://tools.ietf.org/html/rfc6749 .

[RFC6960] Santesson, S, et al., X.509 Internet Public Key Infrastructure Online Certificate Status Protocol – OCSP, RFC 6960, Standards Track, June 2013, https://tools.ietf.org/html/rfc6960 .

[RFC7519] Jones, M., Bradley, J., Sakimura, N., JSON Web Token (JWT), RFC 7519, Standards Track, May 2015, https://tools.ietf.org/html/rfc7519 .

[RFC7523] Jones, M., Campbell, B., Mortimore, C., JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants, RFC 7523, Standards Track, May 2015, https://tools.ietf.org/html/rfc7523 .

[RFC8252] Denniss, W., Bradley, J., OAuth 2.0 for Native Apps, RFC 8252, Best Current Practice, October 2017, https://tools.ietf.org/html/rfc8252 .

[RFC8446] Rescorla, E., The Transport Layer Security (TLS) Protocol Version 1.3, RFC 8446, Standards Track, August 2018, https://tools.ietf.org/html/rfc8446 .

[RZL13] Roman, R., Zhou, J., Lopez, J., On the features and challenges of security and privacy in distributed internet of things, *Computer Networks*, issue 57, June 2013, pp. 2266-2279.

[SaS75] Saltzer, J., Schroeder, M., The Protection of Information in Computer Systems, *Proceedings of the IEEE*, volume 63, issue 9, September 1975, pp. 1278-1308.

[SBJ14] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., Mortimore, C., OpenID Connect Core 1.0 incorporating errata set 1, The OpenID Foundation, November 2014, http://openid.net/specs/openid-connect-core-1_0.html (visited 6.5.2018).

[SCF96] Sandhu, R., Coyne, E., Feinstein, H., Youman, C., Role-Based Access Control Models, *Computer*, Volume 29, Issue 2, February 1996, pp. 38-47.

[Sir14] Siriwardena, Prabath, *Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE*, Apress, August 2014, ISBN 978-1430268185.

[SPI17] Secure Production Identity Framework for Everyone, Google Inc / Cloud Native Computing Foundation, January 2017, https://github.com/spiffe/spiffe/blob/master/standards/SPIFFE.md (visited 2.9.2018)

[SuB12] Sun, S., Beznozov, K., The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems, *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, October 2012, pp. 378-390.

[TaV07] Tanenbaum, A., Van Steen, M., *Distributed Systems: principles and paradigms*, 2nd edition, Prentice-Hall, 2007, ISBN 0-13-239227-5.

[Tur14] Turner, Sean, Transport Layer Security, *IEEE Internet Computing*, volume 18, issue 6, October 2014, pp. 60-63.

[Vin06] Vinoski, Steve, Advanced Message Queuing Protocol, *IEEE Internet Computing*, volume 10, issue 6, November 2006, pp. 87-89.

[YuT05] Yuan, E., Tong, J., Attribute based access control (ABAC) for Web services, *IEEE International Conference on Web Services (ICWS'05)*, July 2005, pp. 561-569.

# Attachment: Ambassador AuthService

Functional components of the Ambassador OpenID Connect authentication service are included in the subchapters. Tests, listed specific dependencies and documentation that are vital parts of any software project, are not included as they are not needed for running the code, but they can be found in the actual code repository [Myy18].

Chapter A.1 includes the main code of the program with routes for different functions and a system health test. Chapter A.2 has the login functionality related to the OpenID Connect Authorization Code Flow and JSON Web Token creation. Chapter A.3 includes the authorisation functionality used for validating incoming requests and revoking JWTs of logged out sessions.

AuthService's version 1.0 was released in the beginning of October 2018. Within its first month, its containerised version passed 10 000 downloads on Docker hub. While some downloads are most likely from the same users, it is still safe to say that it has been a needed component in the area of distributed systems and microservices.

## *A.1. Main*

```
package main

import (
    "log"
    "net/http"
    "net/url"
    "os"
    "time"

    "github.com/gorilla/handlers"
    "github.com/gorilla/mux"
)

var port string

func init() {
    port = os.Getenv("PORT")
    if len(port) == 0 {
        log.Println("No port specified, using 8080 as default.")
        port = "8080"
    }
}

func parseEnvURL(URLEnv string) *url.URL {
    envContent := os.Getenv(URLEnv)
    parsedURL, err := url.ParseRequestURI(envContent)
```

```
    if err != nil {
        log.Fatal("Not a valid URL for env variable ", URLEnv, ": ",
envContent, "\n")
    }

    return parsedURL
}

func parseEnvVar(envVar string) string {
    envContent := os.Getenv(envVar)

    if len(envContent) == 0 {
        log.Fatal("Env variable ", envVar, " missing, exiting.")
    }

    return envContent
}

func scheduleBlacklistUpdater(seconds int) {
    for {
        time.Sleep(time.Duration(seconds) * time.Second)
        go updateBlacklist()
    }
}

// HealthHandler responds to /healthz endpoint for application
monitoring
func HealthHandler(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    w.Write([]byte("OK"))
}

func main() {
    wh := newWildcardHandler()

    router := mux.NewRouter()
    router.HandleFunc("/healthz",
HealthHandler).Methods(http.MethodGet)
    router.HandleFunc("/login/oidc",
OIDCHandler).Methods(http.MethodGet)
    router.HandleFunc("/login", LoginHandler).Methods(http.MethodGet)
    router.HandleFunc("/logout",
LogoutHandler).Methods(http.MethodGet)
    router.PathPrefix("/").Handler(wh)

    updateBlacklist()
    go scheduleBlacklistUpdater(60)

    var listenPort = ":" + port
    log.Println("Starting web server at", listenPort)
    log.Fatal(http.ListenAndServe(listenPort, handlers.CORS()
(router)))
}
```

## A.2. Login

```
package main

import (
    "context"
    "encoding/json"
    "errors"
    "fmt"
    "log"
    "math/rand"
    "net/http"
    "os"
    "strings"
    "time"

    oidc "github.com/coreos/go-oidc"
    jwt "github.com/dgrijalva/jwt-go"
    "github.com/google/uuid"
    "golang.org/x/oauth2"
)

var ctx context.Context
var oauth2Config oauth2.Config
var oidcProvider *oidc.Provider
var oidcConfig *oidc.Config

var hmacSecret []byte
var nonceChars =
[]rune("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
")

func init() {
    // Because Host still has a port if it was in URL
    hostname = strings.Split(parseEnvURL("SELF_URL").Host, ":")[0]


    clientID := parseEnvVar("CLIENT_ID")
    clientSecret := parseEnvVar("CLIENT_SECRET")

    ctx = context.Background()

    provider, err := oidc.NewProvider(ctx,
parseEnvURL("OIDC_PROVIDER").String())
    if err != nil {
        log.Fatal("OIDC provider setup failed: ", err)
    }

    oidcConfig = &oidc.Config{
        ClientID: clientID,
    }

    var oidcScopes []string

    // "openid" (oidc.ScopeOpenID) is a required scope for OpenID
Connect flows.
    oidcScopes = append(oidcScopes, oidc.ScopeOpenID)
    for _, elem := range strings.Split(parseEnvVar("OIDC_SCOPES"), "
") {
```

```
        oidcScopes = append(oidcScopes, elem)
    }

    var redirURL = parseEnvURL("SELF_URL").String()
    if string(redirURL[len(redirURL)-1]) == "/" {
        redirURL = string(redirURL[:len(redirURL)-1])
    }
    redirURL = redirURL + "/login/oidc"

    oauth2Config = oauth2.Config{
        ClientID:     clientID,
        ClientSecret: clientSecret,
        RedirectURL:  redirURL,

        // Discovery returns the OAuth2 endpoints.
        Endpoint: provider.Endpoint(),

        Scopes: oidcScopes,
    }

    oidcProvider = provider

    rand.Seed(time.Now().UnixNano())

    // 64 char(512 bit) key is needed for HS512
    hmacSecret = initialiseHMACSecretFromEnv("JWT_HMAC_SECRET", 64)
}

// OIDCHandler processes authn responses from OpenID Provider,
exchanges token to userinfo and establishes user session with cookie
containing JWT token
func OIDCHandler(w http.ResponseWriter, r *http.Request) {
    var authCode = r.FormValue("code")
    if len(authCode) == 0 {
        log.Println(getUserIP(r), "Missing url parameter: code")
        returnStatus(w, http.StatusBadRequest, "Missing url parameter:
code")
        return
    }

    var state = r.FormValue("state")
    if len(state) == 0 {
        log.Println(getUserIP(r), "Missing url parameter: state")
        returnStatus(w, http.StatusBadRequest, "Missing url parameter:
state")
        return
    }

    // Getting original destination from DB with state
    destination, err := redisdb.Get("state-" + state).Result()
    if err != nil {
        if err.Error() == "redis: nil" { // State didn't exist,
redirecting to new login
            log.Print(getUserIP(r), " No state found with ", state, ",
starting new auth session.\n")
            beginOIDCLogin(w, r, "/")
            return
        }
```

```
        returnStatus(w, http.StatusInternalServerError, "Error
fetching state from DB.")
        panic(err)
    }

    oauth2Token, err := oauth2Config.Exchange(ctx, authCode)
    if err != nil {
        log.Println("Failed to exchange token:", err.Error())
        returnStatus(w, http.StatusInternalServerError, "Failed to
exchange token.")
        return
    }

    rawIDToken, ok := oauth2Token.Extra("id_token").(string)
    if !ok {
        log.Println("No id_token field available.")
        returnStatus(w, http.StatusInternalServerError, "No id_token
field in OAuth 2.0 token.")
        return
    }

    // Verifying received ID token
    verifier := oidcProvider.Verifier(oidcConfig)
    idToken, err := verifier.Verify(ctx, rawIDToken)
    if err != nil {
        log.Println("Not able to verify ID token:", err.Error())
        returnStatus(w, http.StatusInternalServerError, "Unable to
verify ID token.")
        return
    }

    userInfo, err := oidcProvider.UserInfo(ctx,
oauth2.StaticTokenSource(oauth2Token))
    if err != nil {
        log.Println("Problem fetching userinfo:", err.Error())
        returnStatus(w, http.StatusInternalServerError, "Not able to
fetch userinfo.")
        return
    }

    claims := json.RawMessage{}
    if err = userInfo.Claims(&claims); err != nil {
        log.Println("Problem getting userinfo claims:", err.Error())
        returnStatus(w, http.StatusInternalServerError, "Not able to
fetch userinfo claims.")
        return
    }

    cookie := createCookie(claims, idToken.Expiry, hostname)

    // Removing OIDC flow state from DB
    err = redisdb.Del("state-" + state).Err()
    if err != nil {
        log.Println("WARNING: Unable to remove state from DB,",
err.Error())
    }

    log.Println(getUserIP(r), "Login validated with ID token,
redirecting with cookie.")
    http.SetCookie(w, cookie)
```

```
    http.Redirect(w, r, destination, http.StatusFound)
}

// beginOIDCLogin starts the login sequence by creating state and
forwarding user to OIDC provider for verification
func beginOIDCLogin(w http.ResponseWriter, r *http.Request, origURL
string) {
    var state = createNonce(8)
    err := redisdb.Set("state-"+state, origURL, time.Hour).Err()
    if err != nil {
        panic(err)
    }

    http.Redirect(w, r, oauth2Config.AuthCodeURL(state),
http.StatusFound)
}

func createCookie(userinfo []byte, expiration time.Time, domain
string) *http.Cookie {

    token := jwt.NewWithClaims(jwt.SigningMethodHS512, jwt.MapClaims{
        "jti": uuid.New().String(),
        "iss": hostname,
        "iat": time.Now().Unix(),
        "exp": expiration.Unix(),
        "uif": base64encode(userinfo), // Userinfo will be readable to
user
    })

    tokenString, err := token.SignedString(hmacSecret)
    if err != nil {
        panic(err)
    }

    cookie := &http.Cookie{
        Name:    "auth",
        Value:   tokenString,
        Path:    "/",
        Domain:  domain,
        Expires: expiration,
    }

    return cookie
}

func createNonce(length int) string {
    var nonce = make([]rune, length)
    for i := range nonce {
        nonce[i] = nonceChars[rand.Intn(len(nonceChars))]
    }

    return string(nonce)
}

func parseJWT(tokenstr string) (*jwt.Token, error) {
    token, err := jwt.Parse(tokenstr, func(token *jwt.Token)
(interface{}, error) {
    if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
        return nil, fmt.Errorf("Unexpected signing method: %v",
token.Header["alg"])
```

```
        }

        return hmacSecret, nil
    })

    if err != nil {
        return nil, err
    }

    if token.Valid {
        return token, nil
    }

    return nil, errors.New("Token not valid")
}

func initialiseHMACSecretFromEnv(secEnv string, reqLen int) []byte {
    envContent := os.Getenv(secEnv)

    if len(envContent) < reqLen {
        log.Println("WARNING: HMAC secret not provided or secret too
short. Generating a random one from nonce characters.")
        return []byte(createNonce(reqLen))
    }

    return []byte(envContent)
}
```

## *A.3. Auth*

```
package main

import (
    "crypto/md5"
    "encoding/base64"
    "encoding/hex"
    "encoding/json"
    "log"
    "net/http"
    "os"
    "reflect"
    "strings"
    "time"

    jwt "github.com/dgrijalva/jwt-go"
    "github.com/go-redis/redis"
)

var hostname string
var redisdb *redis.Client

var logoutCookie = false

var blacklist []string

type blacklistItem struct {
    Key         string    `json:"key"`
```

```
    JWTHash    string    `json:"hash"`
    Expiration time.Time `json:"exp"`
}

func init() {
    redisAddr := parseEnvVar("REDIS_ADDRESS")
    redisPwd := parseEnvVar("REDIS_PASSWORD")
    redisdb = redis.NewClient(&redis.Options{
        Addr:     redisAddr,
        Password: redisPwd,
        DB:       0,
    })

    _, err := redisdb.Ping().Result()
    if err != nil {
        log.Fatal("Problem connecting to Redis: ", err.Error())
    }

    envContent := os.Getenv("LOGOUT_COOKIE")
    if envContent == "true" {
        logoutCookie = true
    }
}

// LoginHandler processes login requests
func LoginHandler(w http.ResponseWriter, r *http.Request) {
    beginOIDCLogin(w, r, "/")
}

// Wildcardhandler to provide ServeHTTP method required for Go's
handlers
type wildcardHandler struct {
}

func (wh *wildcardHandler) ServeHTTP(w http.ResponseWriter, r
*http.Request) {
    AuthReqHandler(w, r)
}

func newWildcardHandler() *wildcardHandler {
    return &wildcardHandler{}
}

// AuthReqHandler processes all incoming requests by default, unless
specific endpoint is mentioned
func AuthReqHandler(w http.ResponseWriter, r *http.Request) {
    cookie, err := r.Cookie("auth")
    if err != nil {
        log.Println(getUserIP(r), r.URL.String(), "Cookie not set,
redirecting to login.")
        beginOIDCLogin(w, r, r.URL.Path)
        return
    }

    if len(cookie.Value) == 0 { // No auth header set
        log.Println(getUserIP(r), r.URL.String(), "Empty authorization
header.")
        returnStatus(w, http.StatusBadRequest, "Cookie empty or
malformed.")
    } else {
```

```
        token, err := parseJWT(cookie.Value)
        if err != nil {
            if err.Error() == "Token is expired" {
                w.Header().Set("X-Unauthorized-Reason", "Token
Expired")
                log.Println(getUserIP(r), r.URL.String(), "JWT token
expired.")
            } else {
                log.Println(getUserIP(r), r.URL.String(), "Problem
validating JWT:", err.Error())
            }

            returnStatus(w, http.StatusUnauthorized, "Malformed or
expired token in cookie.")
            return
        }

        if checkBlacklist(hashString(token.Raw)) {
            log.Println(getUserIP(r), r.URL.String(), "Token in
blacklist.")
            returnStatus(w, http.StatusUnauthorized, "Not logged in")
            return
        }

        uifClaim, err := base64decode(token.Claims.(jwt.MapClaims)
["uif"].(string))
        if err != nil {
            log.Println(getUserIP(r), r.URL.String(), "Not able to
decode base64 content:", err.Error())
            returnStatus(w, http.StatusBadRequest, "Malformed
cookie.")
            return
        }

        log.Println(getUserIP(r), r.URL.String(), "Accepted.")
        w.Header().Set("X-Auth-Userinfo", string(uifClaim[:]))
        returnStatus(w, http.StatusOK, "OK")
    }
}

// LogoutHandler blacklists user token
func LogoutHandler(w http.ResponseWriter, r *http.Request) {
    cookie, err := r.Cookie("auth")
    if err != nil {
        log.Println(getUserIP(r), r.URL.String(), "Cookie not set, not
able to logout.")
        returnStatus(w, http.StatusBadRequest, "Cookie not set.")
        return
    }

    token, err := parseJWT(cookie.Value)
    if err != nil {
        log.Println(getUserIP(r), r.URL.String(), "Not able to use
JWT:", err.Error())
        returnStatus(w, http.StatusBadRequest, "Malformed JWT in
cookie.")
        return
    }

    tokenHash := hashString(token.Raw)
```

```
    if checkBlacklist(tokenHash) {
        log.Println(getUserIP(r), r.URL.String(), "Token already
blacklisted, cannot to logout again.")
        returnStatus(w, http.StatusForbidden, "Not logged in.")
        return
    }

    jwtExp := int64(token.Claims.(jwt.MapClaims)["exp"].(float64))

    _, err = addToBlacklist(tokenHash, time.Unix(jwtExp, 0))
    if err != nil {
        log.Println(getUserIP(r), "Problem setting JWT to Redis
blacklist:", err.Error())
        returnStatus(w, http.StatusInternalServerError, "Problem
logging out.")
        return
    }

    log.Println(getUserIP(r), r.URL.String(), "Logged out, token added
to blacklist.")

    if logoutCookie { // Sends empty expired cookie to remove the
logged out one.
        var emptyClaims []byte
        newCookie := createCookie(emptyClaims, time.Now().AddDate(0,
0, -2), hostname)
        http.SetCookie(w, newCookie)
    }

    returnStatus(w, http.StatusOK, "Succesfully logged out.")
}

func returnStatus(w http.ResponseWriter, statusCode int, errorMsg
string) {
    w.WriteHeader(statusCode)
    w.Write([]byte(errorMsg))
}

func getUserIP(r *http.Request) string {
    headerIP := r.Header.Get("X-Forwarded-For")
    if headerIP != "" {
        return headerIP
    }

    return strings.Split(r.RemoteAddr, ":")[0]
}

func hashString(str string) string {
    hasher := md5.New()
    hasher.Write([]byte(str))
    return hex.EncodeToString(hasher.Sum(nil))
}

func base64encode(data []byte) string {
    str := base64.StdEncoding.EncodeToString(data)
    return str
}

func base64decode(str string) ([]byte, error) {
    arr, err := base64.StdEncoding.DecodeString(str)
```

```
    if err != nil {
        return nil, err
    }

    return arr, nil
}

func addToBlacklist(tokenHash string, exp time.Time) (bool, error) {
    blKey := createNonce(8)
    blItem := &blacklistItem{Key: blKey, JWTHash: tokenHash,
Expiration: exp}
    blJSON, err := json.Marshal(blItem)
    if err != nil {
        panic(err)
    }

    err = redisdb.HSet("blacklist", blKey, string(blJSON)).Err()
    if err != nil {
        return false, err
    }

    blacklist = append(blacklist, tokenHash)
    return true, nil
}

func updateBlacklist() {
    res, err := redisdb.HVals("blacklist").Result()
    if err != nil {
        panic(err)
    }

    var newBlacklist []string

    for _, i := range res {
        var blItem blacklistItem

        err = json.Unmarshal([]byte(i), &blItem)
        if err != nil {
            panic(err)
        }

        if blItem.Expiration.Before(time.Now()) {
            log.Println("Removing expired token", blItem.Key, "from
blacklist.")
            err = redisdb.HDel("blacklist", blItem.Key).Err()
            if err != nil {
                panic(err)
            }
            continue
        }

        newBlacklist = append(newBlacklist, blItem.JWTHash)
    }

    if !reflect.DeepEqual(blacklist, newBlacklist) {
        blacklist = newBlacklist
        log.Println("Blacklist changes in Redis, local blacklist
recreated.")
    }
}
```

```
func checkBlacklist(jwtHash string) bool {
	for _, e := range blacklist {
		if jwtHash == e {
			return true
		}
	}

	return false
}
```