László Viktor Jánoky – ZE5LCX

Formal verification of a JWT based authentication and authorization scheme

Introduction – What is JWT?

- JSON Web Tokens (JWT) – RFC 7519
  - Provides means to represent claims between parties
  - The claims are signed and authenticated, often encrypted
- JWT based authentication and authorization schemes
  - Uses JWT as “Bearer tokens”
  - Tokens are encrypted and decrypted using a JWT secret
    - Not readable or modifiable by the user or third parties
  - A token contains information about it’s bearer
    - E.g. username, roles, application data, etc.
  - A token is issued by one service and consumed by others, identifying the user in the process, without having to access a centralized storage

Introduction – JWT Example

1. Alice provides her username/password combination in an authentication request to the **AuthService**

2. The service verifies it, and packs the user information into a token. This token is encrypted using the **JWT Secret** and returned to Alice

3. Alice uses this token to access the **SecuredService**

4. The secured service decrypts the token using the **JWT Secret** and uses the contained data to determine whether Alice is allowed to consume the service or not.

Introduction – Advantages of JWT

- Scalable – the user information travels in the communication, no need for data access to a user database
- Distributed – only the JWT Secret has to be shared among the services
- Transparent – from the client side, there is no difference between JWT and other token based solutions

Introduction – Problem statement

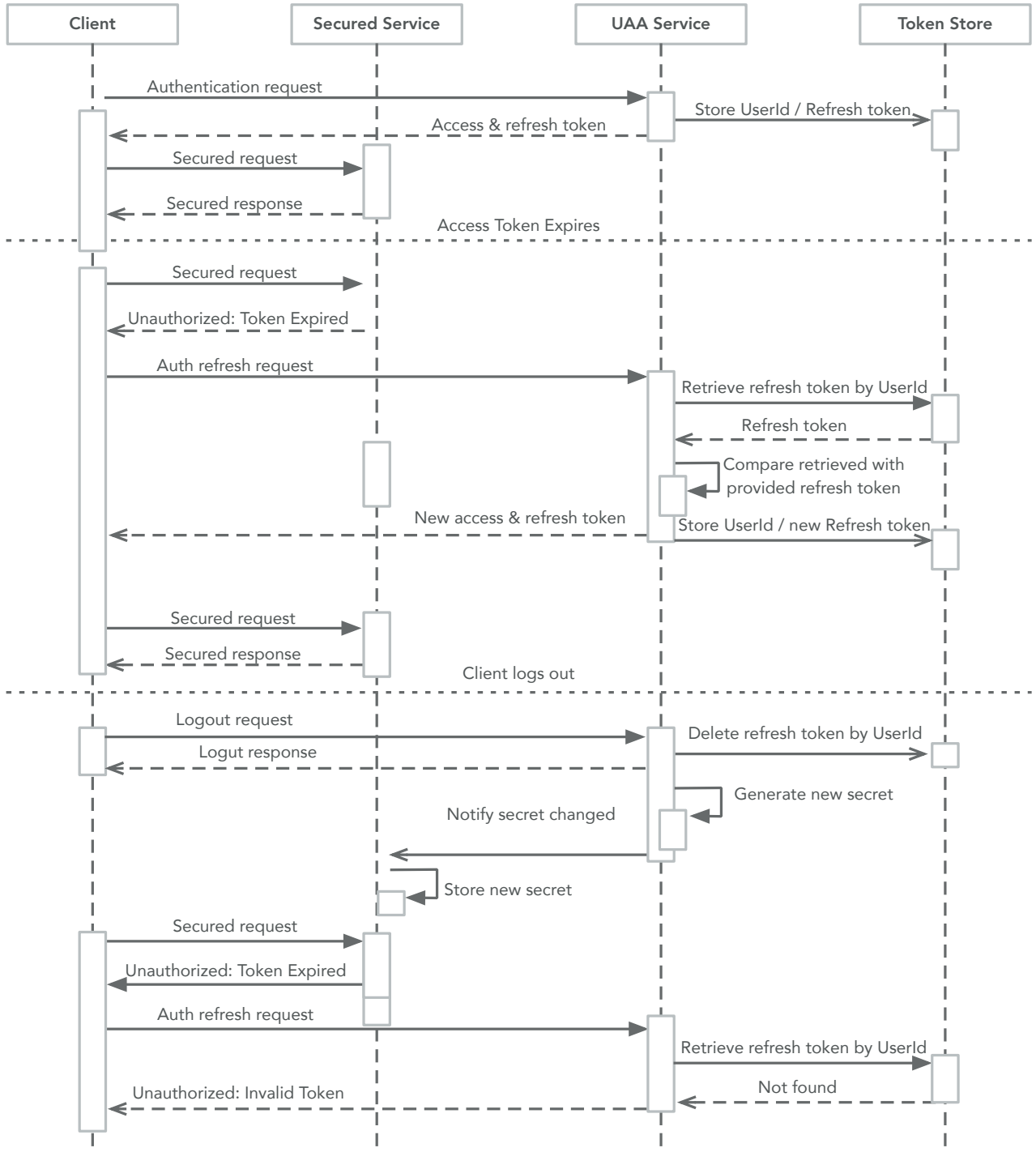
- The main problem with JWT originates from the decentralisation of tokens
- There is no trivial way of token invalidation and revoking
  - Handling client log-out
  - User data changes are hard to propagate
- There are several approaches, each with its unique advantages and drawbacks
  - Short lifetime tokens
  - Blacklist of revoked tokens
  - JWT Secret change on every revocation

Introduction – Current solutions

- Short lifetime tokens
  - Frequent new token acquisitions – server and client load
  - Not instantaneous revocation – possible security issue
- Blacklist of revoked tokens
  - Have to maintain centralized blacklist – defeats the purpose of JWT
- JWT Secret change on every revocation
  - Increased server and client load on frequent revocation events – one revocation means the revocation of all tokens

Possible solution

- Combination of short(er) lifetime tokens and JWT secret change
- Two type of tokens
  - Access token – shorter lifetime JWT token
  - Refresh token – longer lived token, used to acquire new access tokens
- Two classes of revocation
  - Soft – when the client leaves the system without explicit “log-out”, we let the tokens naturally expire (based on their lifetime)
  - Hard – if the client explicitly logs out, we change the JWT secret and revoke the refresh token
- To avoid excessive revocation events, the clients are grouped into fixed size groups, each group with its own JWT secret
  - Group size and allocation tweaked to minimize revocations (based on statistical data about the client population)


Client
Secured Service
UAA Service
Token Store
Authentication request
Store UserId / Refresh token
Access & refresh token
Secured request
Secured response
Access Token Expires
Secured request
Unauthorized: Token Expired
Auth refresh request
Retrieve refresh token by UserId
Refresh token
Compare retrieved with provided refresh token
New access & refresh token
Store UserId / new Refresh token
Secured request
Secured response
Client logs out
Logout request
Delete refresh token by UserId
Logut response
Generate new secret
Notify secret changed
Store new secret
Secured request
Unauthorized: Token Expired
Auth refresh request
Retrieve refresh token by UserId
Not found
Unauthorized: Invalid Token

Goals

- Formally verify the previously described scheme
- Core functionality – must
  - Token assignment process
  - Secured resource access
  - Token revocation process
- “Live characteristics” – if possible
  - Effects on multiple clients – i.e. no starvation because of frequent revocation
  - Edge cases – i.e. revocation during secured resource serving

Tools and steps

01

1. Creating the formal model of the system

- Timed automaton
- Using UPAAL

02

2. Model checking

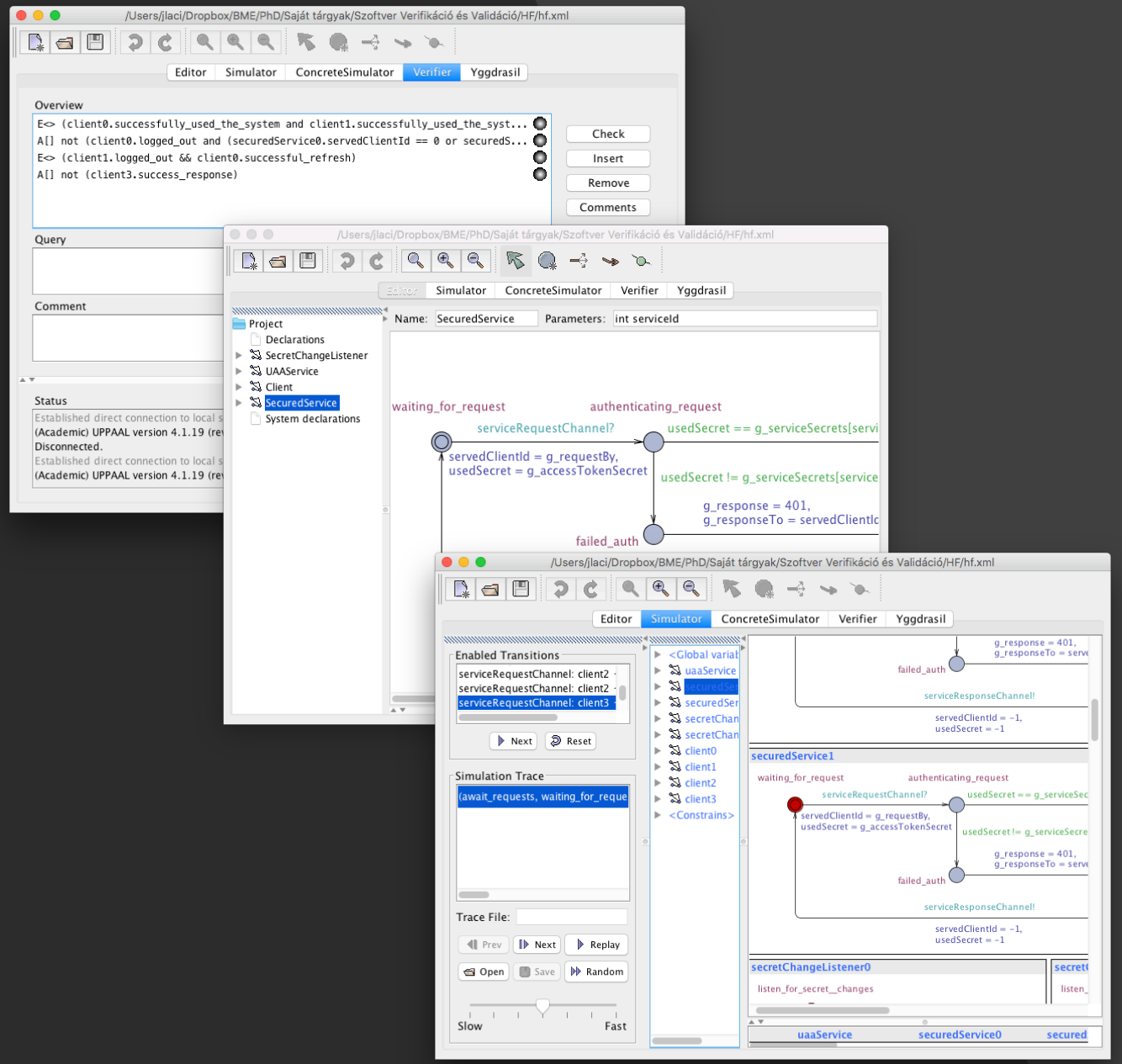
- CTL – Computation Tree Logic
- Using UPPAAL

03

3. Evaluating the results

Tools and steps

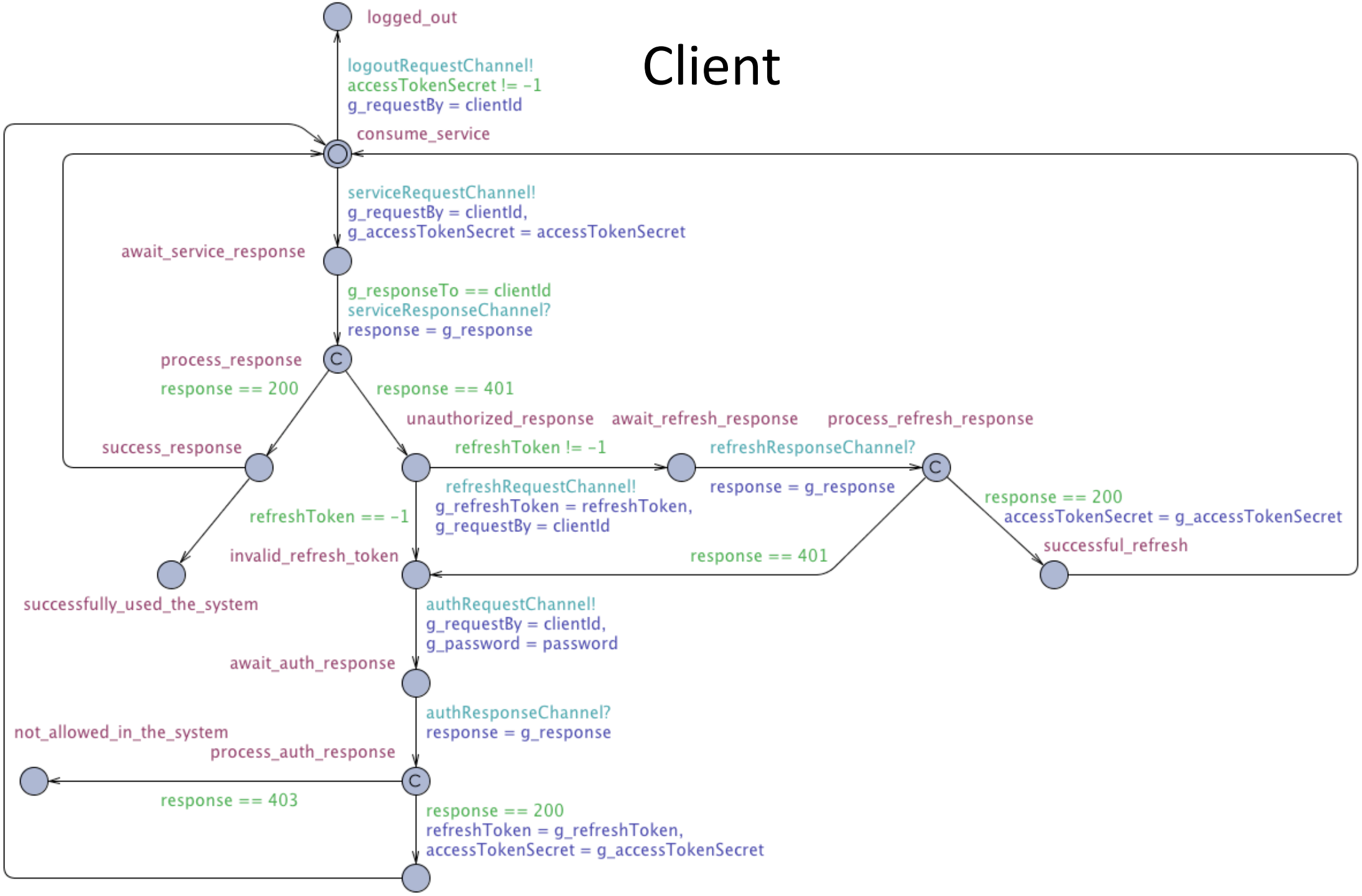
- UPPAAL: Tool for the modeling, validation and verification of real-time systems
- Modeling, using networked timed automatons
- Simulator for trying and visualizing behavior
- Verifier, using a subset of CTL to verify the solution



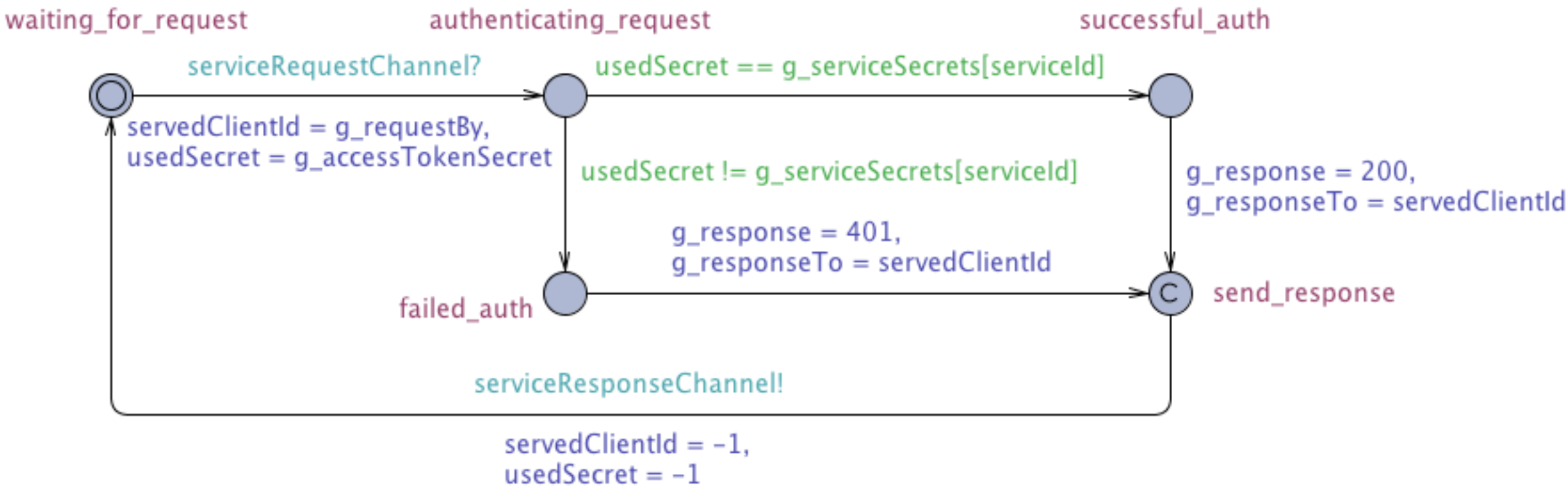
Formal model of the System

- 4 Timed automation templates
- Client
  - modelling a client’s behaviour
- SecuredService
  - the service which we secure using the scheme
- UAAService
  - User Authentication and Authorization service, handling token operations
- SecretChangeListener
  - Utility to model the propogation of secret change events in the system

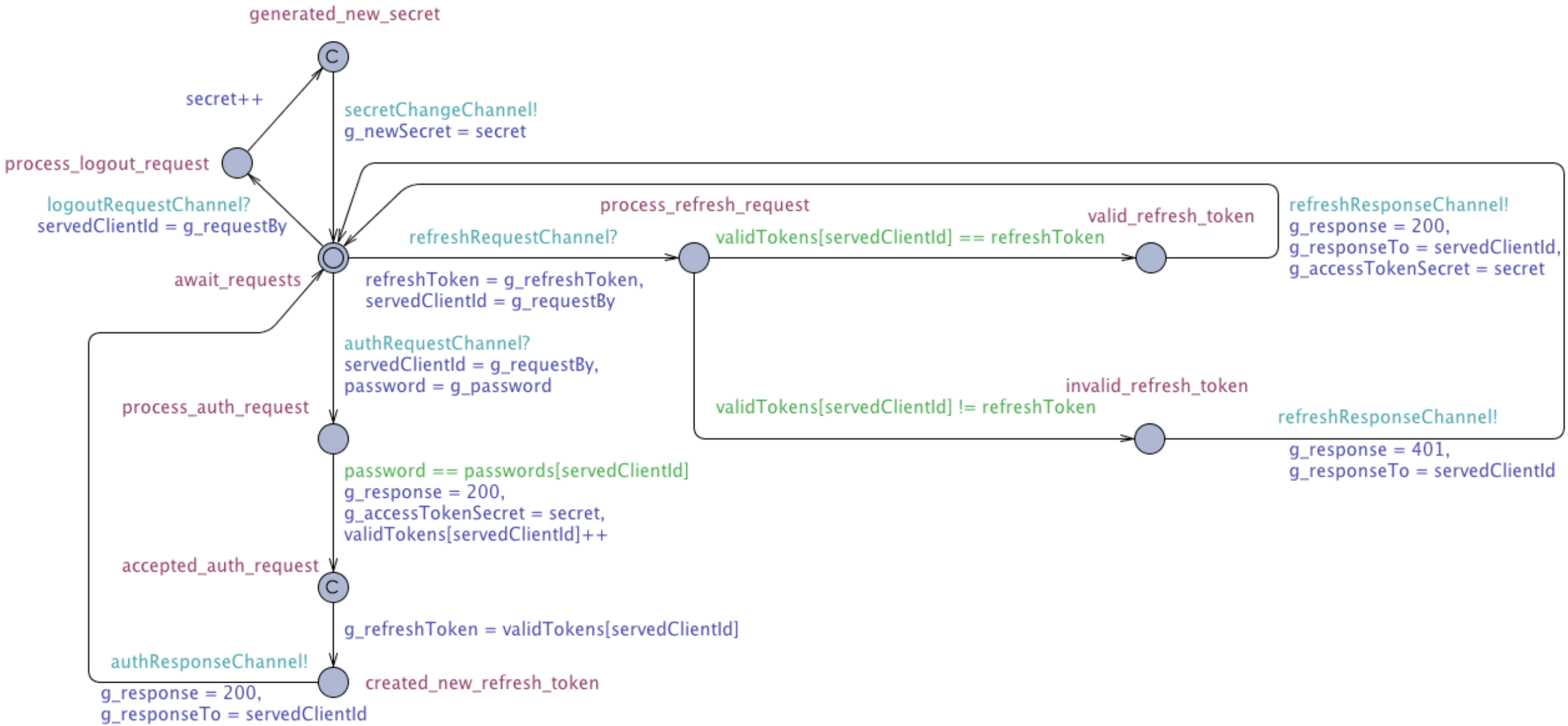
Client


logged_out
logoutRequestChannel!
accessTokenSecret != -1
g_requestBy = clientId
consume_service
serviceRequestChannel!
g_requestBy = clientId,
g_accessTokenSecret = accessTokenSecret
await_service_response
g_responseTo == clientId
serviceResponseChannel?
response = g_response
process_response
C
response == 200
response == 401
unauthorized_response
await_refresh_response
process_refresh_response
success_response
refreshToken != -1
refreshResponseChannel?
refreshRequestChannel!
g_refreshToken = refreshToken,
g_requestBy = clientId
response = g_response
C
response == 200
accessTokenSecret = g_accessTokenSecret
refreshToken == -1
invalid_refresh_token
response == 401
successful_refresh
successfully_used_the_system
authRequestChannel!
g_requestBy = clientId,
g_password = password
await_auth_response
authResponseChannel?
response = g_response
not_allowed_in_the_system
process_auth_response
C
response == 403
response == 200
refreshToken = g_refreshToken,
accessTokenSecret = g_accessTokenSecret

Secured service



UAAService


generated_new_secret
C
secret++
secretChangeChannel!
g_newSecret = secret
process_logout_request
logoutRequestChannel?
servedClientId = g_requestBy
process_refresh_request
valid_refresh_token
refreshResponseChannel!
g_response = 200,
g_responseTo = servedClientId,
g_accessTokenSecret = secret
refreshRequestChannel?
validTokens[servedClientId] == refreshToken
await_requests
refreshToken = g_refreshToken,
servedClientId = g_requestBy
authRequestChannel?
servedClientId = g_requestBy,
password = g_password
invalid_refresh_token
validTokens[servedClientId] != refreshToken
refreshResponseChannel!
g_response = 401,
g_responseTo = servedClientId
process_auth_request
password == passwords[servedClientId]
g_response = 200,
g_accessTokenSecret = secret,
validTokens[servedClientId]++
accepted_auth_request
C
g_refreshToken = validTokens[servedClientId]
authResponseChannel!
created_new_refresh_token
g_response = 200,
g_responseTo = servedClientId

SecretChangeListener

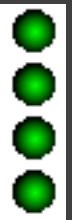
Model checking – Test System

- 4 Clients
  - Client0 – have a valid access and refresh token, knows their password
  - Client1 – have a valid refresh token, knows their password
  - Client2 – don’t have any tokens, but knows their password
  - Client3 – don’t have any token and don’t knows the password
- 2 Secured services
  - With appropriate Secret Change Listeners
- 1 UAAService
- Not optimal, a larger system won’t compute with the available resources (mainly memory) due to the state explosion problem

CTL

- Verified cases
  - Client 1-3 can access the secured service
  - Client 4 can never access the secured service
  - Once a client logged out, and it’s tokens were revoked, it can’t access the service anymore
  - A Client logout causes token revocation on other clients (i.e. Client1 logs out, Client0 must acquire a new access token to access the secured service)

```
E<> (client0.successfully_used_the_system and client1.successfully_used_the_system and client2.successfully_used_the_system)
A[] not (client0.logged_out and (securedService0.servedClientId == 0 or securedService1.servedClientId == 0))
E<> (client1.logged_out && client0.successful_refresh)
A[] not (client3.success_response)
```



Challenges

- Large complexity of the scheme
  - The model had to be simplified
    - Models only one-group of clients with the same JWT Secret
    - No new logins of the same client after logging out, otherwise state explosion
  - Results in long computational times for model checking
    - Usually the memory runs out before the check could be finished
- Passing data between automations is very complicated and error prone
  - Global variables and sync to signal their changes, ensuring no other transitions possible during their transfer
- Some interesting metrics could not be checked using CTL/UPPAAL
  - E.g. the ratio of token acquisitions and token uses in actual requests

Conclusion

- The core features of the solution were verified on the test system
  - Access control
  - Token revocation
- Further directions
  - Investigate how to extend to arbitrary clients and services
  - Other formal methods for examining different metrics during operation

Feel free to ask questions!

Thank you for your attention!