Verteilte Systeme im Sommersemester 2021

Steffen Herweg, Matr. Nr. 873475
Luca Fabio Kock, Matr. Nr. 879534

Osnabrück, 18.05.2021

Aufgabenblatt 5

Tests:

Erfolgreiche Anmeldung:

```
lucakock@id.hsos.de@si0024-015-lin:~/Verteilte_Systeme/Blatt5$ ./pub_sub_client
Pub / sub server is: 0.0.0.0:40040
Client usage:
     'quit' to exit;
     'set_topic' to set new topic;
     'subscribe' subscribe to server & register / start receiver;
     'unsubscribe' from this server & terminate receiver.
> login
username:lucakock
password:passwort
lucakock 9921bd10074d32c3c419cb8570f32755ff5aae416dbd1229ca48979276ad6d50
validate() -> OK
```

Fehlerhaftes Passwort bei Anmeldung:

```
> login
username:lucakock
password:falschesPasswort
lucakock 0a7f42724c4ece7cf52da158053cc967c0b1d33a628649a05dcc4a827c670091
validate() -> Wrong Hash for Session
```

Kommunikationsbeispiel:

```
username:lucakock
password:passwort
lucakock 9921bd10074d32c3c419cb8570f32755ff5aae416dbd1229ca48979276ad6d50
validate() -> OK
> subscribe
subscribe() -> OK
> Test
publish() -> OK
> Test
publish() -> OK
> set_topic
enter topic> 0815
enter passcode> 0815
set_topic() -> OK
> Test2
publish() -> OK
>
```

```
Receiver@si0024-015-lin.res.hsos.de

Server gestartet auf 0.0.0.0:40041
18-05-2021 16:38:55 lucakock: <no topic set>: Test
18-05-2021 16:39:14 lucakock: 0815: Test2
```

Pub_sub_Client.cc:

```
std::string hash(std::string first, std::string second){
    std::string str = first + ";" + second;
    std::stringstream hash_val;
    unsigned char result[SHA256_DIGEST_LENGTH];
    SHA256((unsigned char*) str.c_str(), str.length(), result);
    for (int i = 0; i < SHA256_DIGEST_LENGTH; i++) {
        char tmp[3];
        sprintf(tmp, "%02x", result[i]);
        hash_val << tmp;
    }
    return hash_val.str();
}
```

```
                // TODO: Topic fuer Server vorbereiten ...
                request.mutable_opttopic()->set_passcode(passcode.c_str());
                request.mutable_opttopic()->set_topic(topic.c_str());
                request.mutable_sid()->set_id(sid);
                request.set_hash_string(hash(std::to_string(sid),digest));

                // TODO: Hashwert erzeugen

                // TODO: RPC abschicken ...
                Status status = stub_->set_topic(&context, request, &reply);


                // Status / Reply behandeln
                this->handle_status("set_topic()", status, reply);
            }
            else if (cmd.compare("subscribe") == 0)
            {
                /* Ueberpruefen, ob Binary des Receivers existiert */
                if (access(receiverExecFile, X_OK) != -1)
                {
                    /* Receiver starten */
                    if ((rec_pid = fork()) < 0)
                    {
                        std::cerr << "Cannot create process for receiver!\n";
                    }
                    else if (rec_pid == 0)
                    {
                        /* Der Shell-Aufruf */
                        /* xterm -fa 'Monospace' -fs 12 -T Receiver -
e ...pub_sub_deliv */
                        /* kann nicht 1:1 uebertragen werden. Bei Aufruf via e
xec() */
                        /* verhaelt sich das Terminal anders. */
```

```
                        /* Alternative: Aufruf von xterm ueber ein Shell-
Skript. */
                        /* Allerdings haette man dann 2 Kind-Prozesse. */
                        execl("/usr/bin/xterm", "Receiver", "-
fs", "14", receiverExecFile, (char *)NULL);
                        /* -fs 14 wird leider ignoriert! */
                        exit(0); /* Kind beenden */
                    }

                    /* TODO: Hier den Request verschicken und Ergebnis auswert
en! */
                    /* Platzhalter wie oben lokal erstellen ... */
                    ClientContext clientContext;
                    PubSubParam request;
                    ReturnCode response;
                    // TODO: Receiver Adresse setzen ...
                    request.mutable_optaddress()-
>set_ip_address(get_receiver_ip());
                    request.mutable_optaddress()->set_port(40041);
                    request.mutable_sid()->set_id(sid);
                    request.set_hash_string(hash(std::to_string(sid),digest));

                    // TODO: RPC abschicken ...
                    Status status = stub_-
>subscribe(&clientContext, request, &response);
                    // TODO: Status / Reply behandeln ...
                    this->handle_status("subscribe()", status, response);
                    if(!(response.value() == pubsub::ReturnCode_Values_OK) &&
rec_pid > 0){
                        if (kill(rec_pid, SIGTERM) != 0)
                            std::cerr << "Cannot terminate message receiver!\n
";
                        else
                            rec_pid = -1;
                    }

                }
                else
                {
                    std::cerr << "Cannot find message receiver executable!\n";
                    std::cerr << "Press <return> to continue";
                    char c = getc(stdin);
                    continue;
                }
            }
            else if ((cmd.compare("quit") == 0) ||
                     (cmd.compare("unsubscribe") == 0))
            {
                /* Receiver console beenden */
```

```
                if (rec_pid > 0)
                {
                    if (kill(rec_pid, SIGTERM) != 0)
                        std::cerr << "Cannot terminate message receiver!\n";
                    else
                        rec_pid = -1;
                }
                /* Bei quit muss ebenfalls ein unsubscribe() gemacht werden. *
/


                /* TODO: Hier den Request verschicken und Ergebnis auswerten!
*/
                /* Platzhalter wie oben lokal erstellen ... */

                ClientContext clientContext;
                PubSubParam request;
                ReturnCode response;
                // TODO: Receiver Adresse setzen ...
                request.mutable_optaddress()-
>set_ip_address(get_receiver_ip());
                request.mutable_optaddress()->set_port(40041);
                request.mutable_sid()->set_id(sid);
                request.set_hash_string(hash(std::to_string(sid),digest));

                // TODO: RPC abschicken ...
                Status status = stub_-
>unsubscribe(&clientContext, request, &response);
                // TODO: Status / Reply behandeln ...
                this->handle_status("unsubscribe()", status, response);
                /* Shell beenden nur bei quit */
                if (cmd.compare("quit") == 0)
                    break; /* Shell beenden */
            }
            else if(cmd.compare("login") == 0){
                std::cout << "username:";
                std::string username;
                getline(std::cin,username);

                std::cout << "password:";
                std::string password;
                getline(std::cin, password);
                {
                    ClientContext clientContext;
                    UserName request;
                    SessionId response;

                    request.set_name(username.c_str());
```

```
                    Status status = stub_-
>get_session(&clientContext, request, &response);
                    sid = response.id();
                    digest = hash(username,password);
                    std::cout << username << " " << digest << std::endl;
                }
                {
                    ClientContext clientContext;
                    PubSubParam request;
                    ReturnCode response;

                    request.mutable_void_();

                    request.mutable_sid()->set_id(sid);
                    request.set_hash_string(hash(std::to_string(sid),digest));

                    Status status = stub_-
>validate(&clientContext, request, &response);
                    this->handle_status("validate()",status,response);
                }
            }
            else  /* kein Kommando -> publish() aufrufen */
            {
                /* TODO: Hier den Request verschicken und Ergebnis auswerten!
*/
                /* Platzhalter wie oben lokal erstellen ... */

                ClientContext clientContext;
                PubSubParam request;
                ReturnCode response;
                // TODO: Message setzen ...
                request.mutable_optmessage()->set_message(cmd.c_str());
                request.mutable_sid()->set_id(sid);
                request.set_hash_string(hash(std::to_string(sid),digest));

                // TODO: RPC abschicken ...
                Status status = stub_-
>publish(&clientContext, request, &response);
                // TODO: Status / Reply behandeln ...

                this->handle_status("publish()", status, response);
```

Pub_Sub_server.cc:

```
std::string hash(std::string digest, int sessionId){
    std::string str = std::to_string(sessionId) +";"+ digest;
    std::stringstream hash_val;
    unsigned char result[SHA256_DIGEST_LENGTH];
    SHA256((unsigned char*) str.c_str(), str.length(), result);
    for (int i = 0; i < SHA256_DIGEST_LENGTH; i++) {
        char tmp[3];
```

```
        sprintf(tmp, "%02x", result[i]);
        hash_val << tmp;
    }
    return hash_val.str();
}
```

```
  Status subscribe(ServerContext *context, const PubSubParam *request,
                   ReturnCode *reply) override
  {
    int sid = request->sid().id();
    try{
      std::string username = validSessions.at(sid);
      std::string digest = hashes.at(username);

      std::string expectedHash = hash(digest,sid);
      if(expectedHash == request->hash_string()){
        std::string receiver = stringify(request->optaddress());
        bool created = subscribers.emplace(receiver, PubSubDelivService::NewSt
ub(grpc::CreateChannel(receiver, grpc::InsecureChannelCredentials()))).second;
        std::cout << "Created subscriber now " << subscribers.size();
        if(created){
          reply->set_value(pubsub::ReturnCode_Values_OK);
        }else{
          reply->set_value(pubsub::ReturnCode_Values_CANNOT_REGISTER);
        }
      }else{
        reply->set_value(pubsub::ReturnCode_Values_WRONG_HASH_FOR_SESSION);
      }
    }
    catch(std::out_of_range ex){
      reply->set_value(pubsub::ReturnCode_Values_SESSION_INVALID);
    }
    return Status::OK;
  }

  Status unsubscribe(ServerContext *context, const PubSubParam *request,
                     ReturnCode *reply) override
  {
    int sid = request->sid().id();
    try{
      std::string username = validSessions.at(sid);
      std::string digest = hashes.at(username);

      std::string expectedHash = hash(digest,sid);
      if(expectedHash == request->hash_string()){
        std::string receiver = stringify(request->optaddress());
        int removed = subscribers.erase(receiver);
        if(removed > 0){
          reply->set_value(pubsub::ReturnCode_Values_OK);
```

```
        }else{
          reply->set_value(pubsub::ReturnCode_Values_CANNOT_UNREGISTER);
        }
      }else{
        reply->set_value(pubsub::ReturnCode_Values_WRONG_HASH_FOR_SESSION);
      }
    }
    catch(std::out_of_range ex){
      reply->set_value(pubsub::ReturnCode_Values_SESSION_INVALID);
    }
    return Status::OK;
  }
```

```
  Status publish(ServerContext *context, const PubSubParam *request,
                 ReturnCode *reply) override
  {
    int sid = request->sid().id();
    try{
      std::string username = validSessions.at(sid);
      std::string digest = hashes.at(username);

      std::string expectedHash = hash(digest,sid);
      if(expectedHash == request->hash_string()){
        std::cout << "DELIVERING to " << subscribers.size() << std::endl;
        // TODO: Nachricht an alle Subscriber verteilen
        ClientContext clientContext;
        EmptyMessage empty;
        Message requestOut;
        requestOut.set_message((username + ": " + topic + ": " + request-
>optmessage().message()));
        for (auto& subscriberPair : subscribers) {
          Status status = subscriberPair.second-
>deliver(&clientContext, requestOut, &empty);
          handle_status("deliver()",status);
        }
        reply->set_value(pubsub::ReturnCode_Values_OK);
      }else{
        reply->set_value(pubsub::ReturnCode_Values_WRONG_HASH_FOR_SESSION);
      }
    }
    catch(std::out_of_range ex){
      reply->set_value(pubsub::ReturnCode_Values_SESSION_INVALID);
    }
    return Status::OK;
  }

  Status set_topic(ServerContext *context, const PubSubParam *request,
                   ReturnCode *reply) override
  {
```

```
    int sid = request->sid().id();
    try{
      std::string username = validSessions.at(sid);
      std::string digest = hashes.at(username);

      std::string expectedHash = hash(digest,sid);
      if(expectedHash == request->hash_string()){
        if(request->opttopic().passcode().compare(PASSCODE) == 0){
          // TODO: Topic setzen und Info ausgeben
          topic = request->opttopic().topic();
          reply->set_value(pubsub::ReturnCode_Values_OK);
        }
        else{
          reply->set_value(pubsub::ReturnCode_Values_CANNOT_SET_TOPIC);
        }
      }else{
        reply->set_value(pubsub::ReturnCode_Values_WRONG_HASH_FOR_SESSION);
      }
    }
    catch(std::out_of_range ex){
      reply->set_value(pubsub::ReturnCode_Values_SESSION_INVALID);
    }
    return Status::OK;
  }

  Status get_session(ServerContext *context, const UserName* request, SessionI
d *reply){
    int sessionId = clock();
    pendingSessions.emplace(sessionId, request->name());
    reply->set_id(sessionId);
    return Status::OK;
  }

  Status validate(ServerContext *context, const PubSubParam * request, ReturnC
ode *reply){
    int sid = request->sid().id();
    try{
      std::string username = pendingSessions.at(sid);
      if(validSessions.find(sid)==validSessions.end()){
        try{
          std::string digest = hashes.at(username);
          std::string expectedHash = hash(digest,sid);
          if(expectedHash == request->hash_string()){
            pendingSessions.erase(sid);
            validSessions.emplace(sid,username);
            reply->set_value(pubsub::ReturnCode_Values_OK);
          }else{
            reply-
>set_value(pubsub::ReturnCode_Values_WRONG_HASH_FOR_SESSION);
```

```
          }
        } catch (std::out_of_range ex) {
           reply->set_value(pubsub::ReturnCode_Values_NO_HASH_FOR_SESSION);
        }
      }else{
        reply->set_value(pubsub::ReturnCode_Values_USER_ALREADY_LOGGED_IN);
      }
    } catch (std::out_of_range ex){
      reply->set_value(pubsub::ReturnCode_Values_SESSION_INVALID);
    }
    return Status::OK;
  }

  Status invalidate(ServerContext *context, const PubSubParam * request, Retur
nCode *reply){
    int sid = request->sid().id();
    try{
      std::string username = validSessions.at(sid);
      std::string digest = hashes.at(username);

      std::string expectedHash = hash(digest,sid);
      if(expectedHash == request->hash_string()){
        validSessions.erase(sid);
        reply->set_value(pubsub::ReturnCode_Values_OK);
      }else{
        reply->set_value(pubsub::ReturnCode_Values_WRONG_HASH_FOR_SESSION);
      }
    }
    catch(std::out_of_range ex){
      reply->set_value(pubsub::ReturnCode_Values_SESSION_INVALID);
    }
    return Status::OK;
  }

  void importHashes(){
    std::ifstream hashFile("./hashes.txt");
    if(!hashFile.is_open()){
      std::cerr << "hashes.txt could not be opened!\n";
    }
    std::string username, digest;
    while(hashFile >> username>>digest){
      hashes.insert(std::pair<std::string,std::string>(username,digest));
    }
  }
```