Computational Technologies **Vol 7, No 4, 2002**

FORMAL MODEL OF A GROUP KEY AGREEMENT PROTOCOL

E. E. Enaw
University of Yaounde I
National Advanced School of Engineering, Cameroon
e-mail: `ebotenaw@yahoo.com, eebot@uycdc.uninet.cm`

Представлен вывод абстрактной спецификации по набору неформальных требований для полного группового протокола согласования ключа, основанного на алгоритме Диффи — Хеллмана в модели с изменяющимся числом равноправных участников (Dynamic Peer Group, DPG). В работе использован язык спецификации системы RAISE (RSL). В рамках компонентного подхода к построению спецификации использовались средства раздельной разработки системы RAISE.

1. Introduction

Dynamic Peer Groups which include j video conferencing, distributed simulation, replicated servers e.t.c. are fast becoming common in our society. The open nature of networks around the world today, makes the need to secure communication among members of dynamic peer groups very urgent.

In 1976 Whitfield Diffie and Martin Hellman proposed the Diffie — Hellman (DH) 2-party key exchange protocol [6]. This protocol is however limited to a restrained group comprising only two members and thus is not useful in its basic form in larger groups.

The introduction of computer networking and client server applications and the need to secure communication between computers in such networks motivated the extension of this basic 2-party Diffie — Hellman protocol to groups [1, 2, 4, 5, 10 – 12], so that the extended protocol could be used for key agreement in Dynamic Peer Groups.

The fact that key distribution is the cornerstone of secure group communication further motivated a lot of research in the area of extending the basic 2-party Diffie — Hellman key agreement protocol to groups.

Unfortunately some of these protocols are only of theoretical interest and no formal proofs are available for most properties of these protocols such as: (Contributory Authenticated, Perfect Forward Secrecy, Resistance to Known Key Attacks). Only informal proofs have been done, which rely on the hardness of the Diffie — Hellman decision problem described in [10].

It should be noted that the key exchange protocols that exist today are specified in natural language even well-known protocols like the 2 party Diffie — Hellman protocol [6] hasn't got a rigorous formal definition. Informal specifications in the area of Dynamic Peer Groups may lead to misinterpretations and thus to different implementations that will not be able to work together.

In this paper we contribute to research in the area of key exchange protocols in DPGs by providing a formal specification of a generic Group Diffie — Hellman with Complete Key Authentication Protocol [1 – 3] using the formal specification and verification language "Rigorous Approach to Industrial Software Engineering (RAISE)" [8, 9]. We use the technique of requirements tracing to validate our model by identifying precise locations in the formal specification where specific requirements are met. Confidence conditions are generated for all the modules used in the specification to show their correctness. The property that our system settles in a stable state at the end of the protocol run is specified in the SYSTEM_OK theory, which is subsequently informally verified.

The main advantage of using formal techniques [13, 14] is that a formal specification is a mathematical object which has an unambiquous meaning, therefore mathematical methods could be used to analyse these specifications, such as formal justifications of the correctness of the specification. Other benefits include:

— a formal specification provides a clear understanding of the system;

— the formalisation process can reveal inconsistencies, loose ends and incompletenesss in the informal requirements;

— properties of the system could be separately specified and verification techniques such as theorem proving and model checking could be used to prove that the specification of the model of the system satisfies these properties.

This paper is organised as follows: In section 2 we present key establishment protocols. In section 3 we present some notations used throughout the paper. In section 4 we present some Diffie — Hellman based computations. In section 5 we present some Diffie — Hellman based key generation operations. In section 6 we present requirements and properties of the Generic Complete Group Key Authentication Protocol. In section 7 we outline the method used to specify the system. In section 8 we present and specify the various components of the Generic Complete Group Key Authentication Protocol used in Dynamic Peer Groups, using the RAISE specification Language RSL [8]. In section 9 we provide a formal model of our system based on the components specified in section 8. In section 10, through the formal specification and justification of the "System ok theory", we show that our formal model works properly. In section 11, we conclude and give directions for future work.

2. Key establisment protocols

Several key agreement protocols geared for DPGs have been proposed recently. These protocols were obtained by extending the two-party Diffie — Hellman key agreement to n parties, wich perform initial key agreement (IKA) within the group. When the group is formed and the initial key is agreed upon, group members may leave (or be excluded) and new members may join. Any membership change must cause a corresponding change of the group key in order to maintain key independence (old keys cannot be known to new members and new keys cannot be known to former members).

There are two major categories of key establishment protocols: key agreement protocols [1] and centralised key distribution protocols based on some form of trusted third party (TTP). In this paper we focus on contributory key agreement and briefly note some features of centralised key distribution that make it unsuitable for DPGs.

1. A TTP that generates and distributes keys to all members of the group is a single point of failure and a likely performance bottleneck.

2. A TTP presents a very attractive attack target for adversaries, since all group secrets are generated in one place.

3. Some DPG environments (e. g. ad hoc wireless networks) are highly dynamic and no group member is present all the time. However most key distribution protocols assume fixed centers.

We therefore argue in favor of distributed, contributory key agreement for DPGs, we however recognise the need for a centralised point of control for group membership operations such as adding and deleting members.

2.1. Complete authenticated group Diffie — Hellman key agreement SA-GDH.2 protocol

Let agent_set(a_{nb}, d) = { a_1, ... , a_{nb} } (DPG) be a set of users wishing to share a key S_{nb}. The protocol executes in nb rounds. In the first stage ($nb - 1$ rounds) contributions are collected from individual group members and then in the last stage ($nb - th$ round) the group controller broadcast the group keying material to the other group members (Fig. 1).

Key confirmation is an important feature in key agreement protocols. Its purpose is to convince one or more parties that its peer (or group thereof) is in possession of the key.

Complete key confirmation (in the spirit of complete key authentication) would make it necessary for all group members to compute the key and then confirm to all other members the knowledge of the key. This would entail, at the very least, one round of n simultaneous broadcasts. In this paper we take a more practical approach by concentrating on key confirmation emanating from the the group controller, the first group member to compute the actual key.

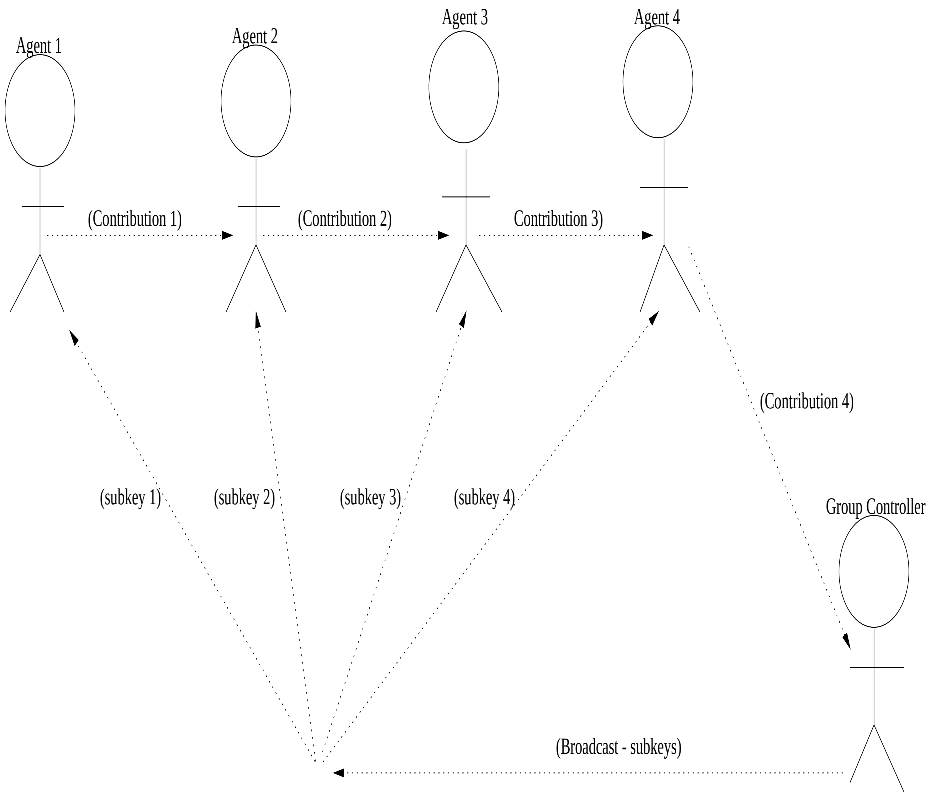

Fig. 1. Key generation operations of a group key agreement protocol.

The last protocol message (the broadcast message from the group controller) contains the group key as computed by the group controller (using the function controller_group_key()).

Upon receipt of the broadcast, each agent computes its key using the function agent_group_key(). Each agent then uses the function match_key(a, agent_group_key(), controller_group_key()) to compare the group key it generates with that generated by the controller. In the event where both keys are equal the agent changes to the AGENT_OK state (indicating a successful end of the protocol), in the event where both keys are not equal the agent changes to the AGENT_ERROR state from where the key generation process is repeated.

Key confirmation coupled with implicit key authentication, provides entity authentication of the group controller to all other group members.

3. Protocol preliminaries

We use the following notation throughout the paper:

a	A group member referred to as an Agent.
nb	Number of protocol parties (group members).
c	The group controller having the maximum index in the DPG.
n	Nonce generated by a group member (random secret generated by a group member).
$created(a, n)$	Identifies an agent that creates and owns a given nonce.
$pub\_key(a)$	The public key of agent "a".
$indx(a)$	Index of a group member "a".
$next(a)$	The agent "a_i" that comes after agent "a" in the DPG, i. e. having index; (indx(a_i) = indx(a) + 1).
$previous(a)$	The agent "a_i" that comes before agent "a" in the DPG, i. e. having index; (indx(a_i) = indx(a) − 1).
S_{nb}	Group key shared among nb members (where nb = max_index).
d	The Dynamic Peer Group (set of all current group members).
k_{ij}	Long term secret shared by agents a_i and a_j, with $(i \neq j)$.
S_{nb}(c)	Controller's view of the group key generated by the function **controller_group_key(c, _, _)**.
$S_{nb}(a_i)$	Agent a_i's view of the group key generated by the function **agent_group_key(a_i, _, _)**.
$is\_ok\_match\_key()$	True if controller's view of the group key equals all group member's (agents) view of the group key. This ends the protocol run.
i, j	Indices of group members.
a_i	i^{th} group member; $i \in [1, n]$.
G	Unique subgroup of Z_p^* of order q with p, q prime.
q	Order of the algebraic group.
α	Exponentiation base; generator in group G.
x_i	Long-term private key of a_i.
n_i	Random (secret) exponent (nonce) $\in Z_p^*$ generated by a_i.

The group controller is represented by letter "**c**" which is equal to a_{nb} i.e. the highest indexed agent in the DPG where nb is equal to the total number of agents in the DPG, "**c**" and a_{nb} are therefore interchangeable.

4. Specification of Diffie — Hellman based computations

The Diffie — Hellman based computations specified below, are used throughout the paper:

The type **Prime** is specified as the subtype of the RSL **built-in** type **Int**. The predicate **is_prime** defines an integer p as a prime number if the only positive integers that divide p are 1 and p itself.

The type **Long_term_secret_key** is defined as a subtype of the type **Prime** which falls within the range {1 .. (q-1)} where q is the order of the algebraic group (a prime number).

The type **Pub_key** is defined as a subtype of the type **Int**, which satisfies the predicate **is_pub_key**.

type
Prime = { | i: Int • is_prime(i) | },
Long_term_secret_key = { | i: Prime • i ∈ {1 .. (q-1)}| },
Pub_key = { | i: Int • is_pub_key(i) | },
Shared_secret_key = { | i : Int • is_shared_secret_key(i) | }

value
α : Prime,
q : Prime,
p : Prime,
long_term_secret_key : Agent → Long_term_secret_key

The **mod** function has two arguments: **Int** and **Int**. The first argument represents some arithmetic operation of result type **Int**. The function **mod** returns an **In**t which is the integer remainder (modulus) resulting from the first integer divided by the second integer.

mod : Int x Int → Int
mod(p1, p2) ≡ p1 \ p2

The **pub_key** function computes the public key of a given agent a_i

pub_key : Agent → Pub_key
pub_key(a_i) ≡ mod($\alpha^{(long\_term\_secret\_key(a_i))}$, p)

The K_{ij} function computes the long term secret K_{ij} shared by agents a_i and a_j, with $(i \neq j)$

K_{ij} : Agent x Agent → Shared_secret_key
$K_{ij}(a_i,a_j)$ ≡
mod($\alpha^{(long\_term\_secret\_key(a_i)*long\_term\_secret\_key(a_j))}$, p),

is_pub_key : Int → **Bool**
is_pub_key(i) ≡
(∃ a : Agent • pub_key(a) = i ∧ is_certificate_in(a)),

is_prime : Int → **Bool**
is_prime(i) ≡
(∀ k : Int • (∃ j : Int • j> 0 ∧ k = i/j • (j = 1 ∨ j = i)))

The **is_shared_secret_key** function simply states that if two agents a_i and a_j, are honest and different $(i \neq j)$, then their long term secret keys are different.

is_shared_secret_key : Long_term_secret_key → **Bool**
is_shared_secret_key(x_i) ≡

($\forall\ a_i, a_j$: Agent •
honest(a_i) $\wedge$ honest(a_j) $\wedge$
($\exists\ x_j$: Long_term_secret_key •
long_term_secret_key(a_i) = x_i $\wedge$
long_term_secret_key(a_j) = x_j $\wedge$ $a_i \neq a_j \Rightarrow$
$x_i \neq x_j$))

We assume that an agent can not guess a nonce used by another. A convenient way to specify this is that a nonce is created by an agent and such an agent is unique.

```
created : Agent x Nonce → Bool
axiom
[created_inj]
        (∀ a1, a2 : Agent, n : Nonce •
           created(a1, n) ∧ created(a2, n) → a1 = a2)
```

5. Key generation operations

Round $i(0 < i < nb)$: An agent a_i updates the message it receives from the previous agent in the DPG by doing the following:

Adding its nonce to the set of nonces received from the previous agent.

```
add_nonce : Nonce x Nonce-set → Nonce-set
add_nonce(n, ns) ≡ {n} ∪ ns,
```

Adding the set of keys it shares with each and every member of the DPG (K_{ij}) to the set of shared keys it received from the previous agent.

```
add_shared_keys :
Shared_secret_key-set x Shared_secret_key-set → Shared_secret_key-set
add_shared_keys(ska, skr) ≡ ska ∪ skr,
```

Adding its identity to the set of agents it received from the previous agent, using the add_agent_set function specified below.

```
add_agent_set : Agent x Agent-set → Agent-set
add_agent_set(a_i, ac) ≡ {a_i} ∪ ac,
```

The agent "a_i" then uses the updated components to generate the **send_message(ns,** a_i**, pub_key(**a_{i+1}**), sks, as)** message which represents its contribution towards the generation of the group key, where:

ns = nonce set - generated above.
pub_key(a_{i+1}**)** = public key of the next agent in the DPG - used to encrypt the send_message().
sks = set of keys it shares with each and every member of the DPG - generated above.
as = set of agents - generated above.

Round nb:

1. The group controller broadcast a set of all subkeys to the group.
2. Upon receipt of the broadcast message, each agent a_i where i is the index of the agent, selects the appropriate subkey V_i where:

V_i = **broadcast_message**(($\{n_1, ..., n_{nb}\} \setminus \{n_i\}$), **c**, **pub_key**($a_i$), $\{k_{1i}, ..., K_{ni}\}$, S_{nb}(**c**))

a_i then proceeds to compute its view of the group key, by adding its nonce to the set of nonces received in the broadcast message:

$S_{nb}(a_i)$ = **nonces**(V_i) $\cup \{n_i\} = \{n_1, ..., n_{nb}\}$

a_i now compares the group key it just generated with that generated by the controller:

match_key(a_i, $S_{nb}(a_i)$, **controller_key**(V_i))

If both keys are equal the agent changes to the **Agent_ok** state indicating the successful completion of the protocol.

Obtain a received nonce set from a received message

```
nonces : Message → Nonce-set
    nonces(m) is
        case m of
            message1(n, a, k) → {n},
            message2($n_a$, $n_b$, a, k) → {$n_a$,$n_b$},
            message3(n, k) → {n},
            receive_message(ns, a, k, sk) → ns,
            send_message(ns, a, k, sk) → ns,
            broadcast_message(ns, a, k, sk, gk) → ns
        end,
```

Obtain the controller's view of the group key from the broadcast message

```
controller_key : Message → Group_key
    controller_key(m) ≡
        case m of
            broadcast_message(ns, a, k, sk, gk) → gk
        end,
match_key : Agent x Group_key x Group_key → Bool
```

5.1. Certificate verification

Before sending a message, an agent gets the certificate of the destination agent from the public key file (set of certificates of all agents in the DPG) which resides with each agent of the DPG, checks to see if the certificate is valid and if it is valid it extracts the public key of the destination agent of the DPG and uses it to encrypt the message before sending.

```
is_certificate_in : Agent → Bool
is_certificate_in(a) ≡
    (∀ cert : Certificate, cs : Certificate-set •
        cert = get_certificate(a, cs) ∧
            is_certificate_valid(cert) ⇒ cert ∈ cs),

is_certificate_valid : Certificate → Bool

get_certificate : Agent x Certificate-set → Certificate
```

6. Requirements and properties

6.1. Requirements

1. All agents must participate in the generation of the group key.

2. Once a group is formed and the initial key is agreed upon, group members may leave (or be excluded) and new members may join.

3. Any membership change must cause a corresponding group key change in order to preserve key independence.

4. The group controller should be able to store the last upflow message it receives.

5. All agents are unique (can‘t have two identical agents).

6. Agent A_i can only make its contribution after receiving and updating agent A_{i-1}‘s contribution, since its contribution is the updated contribution of agent A_{i-1}.

7. The group controller is the only agent that can broadcast, add members to and remove members from the DPG.

8. The highest index agent is called the group controller.

9. The index of the group controller increases by one when members are added and reduces by one when members are removed.

10. The DPG has a maximum size.

6.2. Properties of the model of a generic authenticated group key agreement protocol

— The generic protocol is a contributory authenticated key agreement protocol.

— The generic protocol provides perfect forward secrecy.

— All group members participate in the generation of the group key.

— The group key must be genuine, i.e. a group key is genuine if all members of the protocol end up generating the same key at the end of the protocol.

— The generic protocol provides key integrity.

— The generic protocol provides key confirmation.

— Group members could be honest (won‘t reveal the group key to outsiders) or dishonest.

These properties are necessary to achieve resistance to active attacks mounted by an increasingly powerful adversary, and as always, ironclad security must be achievable with the lowest possible cost.

7. Development method

The development process involved the following aspects:

1. Define a scheme TYPES containing the types and attributes for the non-dynamic entities identified, and make a global object T for this.

2. Define a scheme Buffer containing operations for passing messages from one agent to another.

3. Define the STATION scheme which contains the operations that an agent performs to generate the group key.

4. Define the CONTROLLER scheme which contains the operations that the group controller performs to generate the group key.

5. Define the ADD_MEMBER scheme which contains the operations that the group controller performs to add a new member to the group.

6. Define the REMOVE_MEMBER scheme which contains the operations that the group controller performs to remove an existing member from the group.

7. Define the DPG scheme which specifies a “run” operation on the appropriate modules to execute the key generation process of the DPG. The run terminates when all agents of the DPG settle down in a stable state (ok or error).

8. Define **theorems** which specify properties of the system, identified in the properties section above.

9. We show that the protocol works well, by tracing where requirements are met in the specification and generating confidence conditions for all the modules used in the specification to show their correctness.

10. Consider any efficiency improvements we can make on the final specifications of the various schemes above.

11. Translate to the intended target language.

8. System components

If we want to develop systems of any size, it is a good idea to decompose their description into components and compose the system from the (developed) components. In this section we provide definitions of components of our system and later compose our system from the specified components.

8.1. Agents

We use the short record RSL type definition to specify an Agent. This approach omits the constructor in the definition since there is only one alternative and it would therefore appear odd to include it. Agent is specified as a “sort” since in the requirements, it isn’t stated how agents are to be represented in terms of their names, addresses etc. The short record definition for the “sort” Agent has four destructors namely: state_a, state_r, state and msgs_r, for decomposing or extracting values of the “sort” Agent. The “sort” also has four corresponding reconstructors: change_state_a, change_state_r, change_state, change_msgs, for modifying its values.

The destructor **state** is used in the Agent module **STATION_NEW** specified below to extract values of the sort **Agent_state**. The destructor **state** is used as a **value expression** of a case expression. Depending on the value of **state**, one of six states of the Agent defined below is selected (returned).

The corresponding reconstructor **change_state** is used to change from one Agent state to another (i. e. modify values of the sort **Agent_state**). The destructor is within the value_expression of the corresponding pattern of the case expression. The destructor is not shown here but appears in the full specification of the STATION_NEW module. The explanation above holds for the other constructors and destructors that appear in the short record definition of an Agent above.

```
type
    Agent ::
    state_a : Adding_member_state ↔ change_state_a
    state_r : Removing_member_state ↔ change_state_r
    state : Agent_state ↔ change_state
    msgs_r : Message ↔ change_msgs,
```

We identify two types of agents. The first type of agent is the ordinary member of the DPG, while the second type is the group controller of the DPG. We note that a DPG has one and only one group controller. This requirement is specified in the predicate **is_agent_c** below which states that if two agents have the same index which is equal to **max_index** then both agents in question should be one and the same agent which is the group controller.

The two types of agents identified above, are specified below.

```
type
    Agent_s = { | a : Agent • is_agent_a(a) | },
    Agent_c = { | c : Agent • is_agent_c(c) | },
value
   is_agent_a : Agent → Bool
   is_agent_a(a) ≡ kind(a) = agent ∧ is_indx(a),

   is_agent_c : Agent → Bool
   is_agent_c(c) ≡
      (∀ a : Agent •
         (indx(c) = indx(a) ⇒ c = a) ∧
            kind(c) = controller ∧ indx(c) = max_index),
```

The predicate **is_agent_c** defines the property that the group controller is the highest indexed agent (having index max_index) in the DPG. The predicate **is_agent_a** on the other hand defines a property of an agent, which states that each and every agent of the DPG has a unique index which is within the range (min_index to max_index) with the former belonging to the first agent of the DPG and the later to the group controller.

```
type
   Agent_kind == agent | controller
   Index
value
   kind : Agent → Agent_kind
   index : Agent → Index,
```

```
is_indx : Agent → Bool
is_indx(a) ≡
   min_index ≤ indx(a) ≤ max_index,
```

The agents_indexable **axiom** specified below ensures that indexes of agents are unique.

```
axiom
[agents_indexable]
    (∀ a1, a2 : Agent •
       indx(a2) = indx(a1) ⇒ a2 = a1),
```

As illustrated in Fig. 2 below an agent has six possible states of type **Agent_state** specified below.

```
type
Agent_state ==
    Agent_0 |
    Agent_1 |
    Agent_2 |
    Agent_3 |
    Agent_OK |
    Agent_error,
```

The operations of an agent resulting to the generation of the agent's view of the group key are specified in the **AGENT_NEW** module below. An agent uses the **change_state_a** reconstructor to change from one state to another.

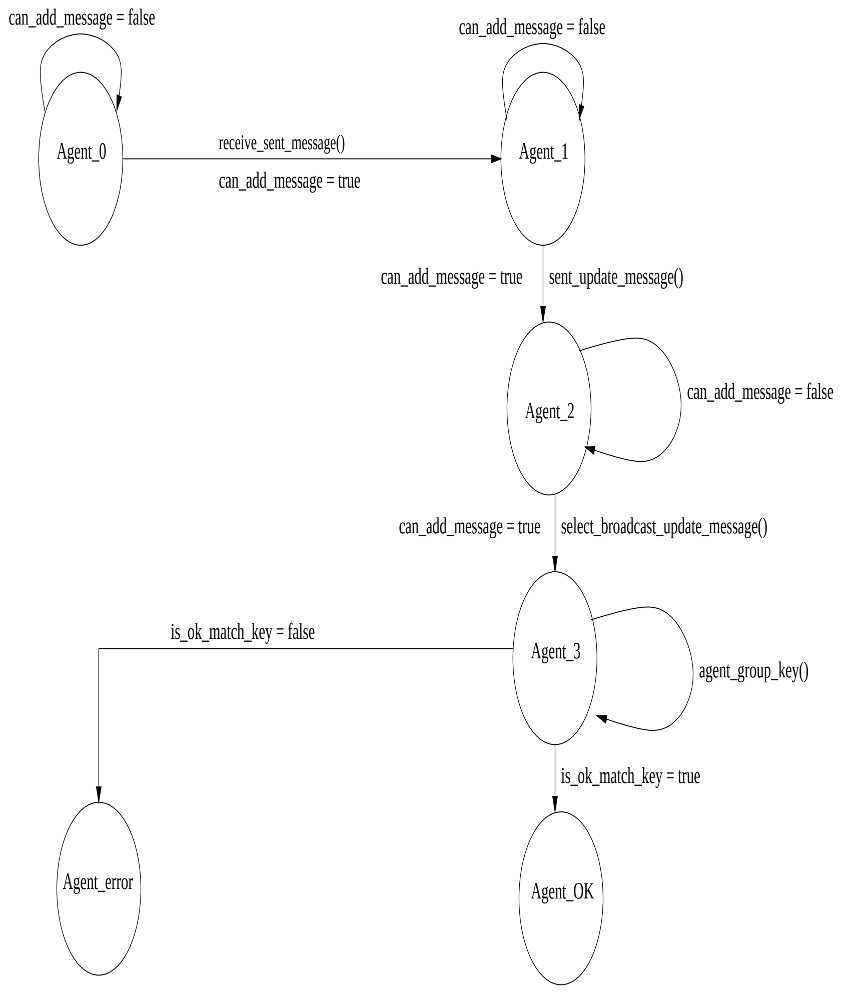

Fig. 2. State transition diagram for a station in the DPG.

```
context: BUFFER, T, TYPES_NEW
scheme STATION_NEW =
    class
        object B : BUFFER(T{Message for Elem})
        value
            s_next : T.Agent_s × T.DPG → T.Agent_s × T.DPG
            s_next(a, d) ≡
                case T.state(a) of
                    T.Agent_0 → receive_sent_message(a, d),
                    T.Agent_1 → send_update_message(a, d),
                    T.Agent_2 → select_broadcast_update_message(a, d),
                    T.Agent_3 → is_ok_match_key(a, d),
                    T.Agent_OK → (a, d),
                    T.Agent_error → (a, d)
                end,
end
```

State_1

Initially an agent is in state “Agent_0”. When it receives the **send_message** (See the Message section below for the set of messages available) from the previous agent in the Dynamic

Peer Group (DPG), using the function **receive_sent_message**, it changes to the next state "Agent 1". It should be noted that the **received** function specified below ensures that an agent can only receive a message that is destined for it since the message must be encrypted with its public key, which can only be decrypted by the agent in question. The **first agent** of the DPG receives an empty set of messages in the first round of the protocol. The **first_station** function below specifies the properties of the first agent.

```
first_station : Agent → Bool
  first_station(a) is
     (∃
        ac : Agent-set, ai : Agent_index •
           indx(a) = min_index ∧ a ∈ ac ∧ a ∈ dom ai
```

State_2

In the second state the first agent of the DPG updates the message it received in the last protocol round and sends the updated message to the next member of the DPG. The first agent therefore carries out the following operations in the second state.

Updates the the message by doing the following:

— Adding its nonce to the empty set of nonces received in the first state.

— Adding the set of keys it shares with each and every member of the DPG to the empty set of shared keys it received in the first state.

— Adding its identity to the empty set of agents it received in the first state, using the **add_agent_set** function specified further ahead.

After carrying out these operations, the first agent now sends the updated message to the second agent of the DPG. Before sending the message the first agent gets the certificate of the second agent from the public key file (set of certificates of all agents in the DPG) which resides with each agent of the DPG, checks to see if the certificate is valid, and if it is valid it extracts the the public key of the second agent of the DPG and uses it to encrypt the message before sending it.

This requirement is specified below in the **send_update_message** algorithm, used by all agents to send an updated message.

```
send_update_message :
   T.Agent_s x T.DPG → T.Agent_s x T.DPG
 send_update_message(a, d) ≡
   let
     buff = T.stat_buffs(d)(a, T.previous(a)),
     (buff', m) = B.get(buff)
   in
     case m of
       T.send_message(ns, T.previous(a), k, sk, ac) →
         if k = T.pub_key(a)
         then
           let
             ad = T.next(a),
             pkf = T.pub_key_file(ac),
             ct = T.get_certificate(ad, pkf)
           in
             if ∼ T.is_certificate_valid(ct)
             then (T.change_state(T.Agent_error, a), d)
             else ...
```

After updating the message, the first agent sends the updated message to the next agent of the DPG using the **put_msg_in** function specified in the Buffer section below. The **put_msg_in** function makes use of the value **stat_buffs** of type **Station_buffs** defined below. Station_buffs = Agent x Agent $\underset{m}{\rightarrow}$ Buffer,

The type **Station_buffs** is a mapping from the product type **(Agent x Agent)** to the type **Buffer** and thus represent's the buffer between two given agents.

The next agent in the DPG (agent with index (i + 1) where i is the index of the first agent) receives the message sent by the first agent only if the public key used to encrypt the sent message belongs to the agent in question. This ensures that only the agent for which the message is intended can read the message. The **received** function below specifies this requirement.

```
received : Agent x Message x State → Bool
    received(a, m, se) ≡
      (∀ st : Status •
        (m, st) ∈ se ∧ key(m) = pub_key(a)),
```

In this state the second agent of the DPG updates the message it received in the last protocol round and sends the updated message to the next member of the DPG.

After updating the received message the second agent now sends the updated message to the next agent. This process continues until the last but one agent sends its updated message to the Group controller of the DPG which is the agent with the highest index. Its role is to broadcast sub-keys used by the corresponding agents of the DPG to compute the group key. It encrypts the sub-key of each member of the DPG with the public key of the agent in question, ensuring that only the intended agent can read this sub key.

State_3
In this state when the agent receives the **broadcast_message** from the group controller generated by the **broadcast_update_message** function specified in the CONTROLLER module, it selects its corresponding sub-key (sub-key which it can decrypt) from the **broadcast_message** and changes to the next state "Agent_3". It should be noted that the controller encrypts sub-keys sent to the rest of the DPG with the public keys of the respective agents, thus an agent selects the sub key it can decrypt, i. e. which the public key used to encrypt it is owned by the agent in question.

State_4
In this state, an agent uses the **agent_group_key** function to compute the group key by adding its nonce to the set of nonces it received through its sub_key (broadcast_message).

It then uses the **match_key** function:

match_key : T.Agent x T.Group_key x T.Group_key → **Bool**

to compare the generated group key to that generated by the controller. If both keys are equal, (indicated by the **is_ok_match_key** function) specified in the **STATION_NEW** module it then changes to the final state **Agent_OK**. If however both keys are not equal, it changes state to the **Agent_error** state from which the key generation process is repeated.

Changing from one state to another is realised through the reconstructor **change_state_a** of short record type DPG' defined in the DPG section above.

8.2. Agents communication

Each agent of the DPG has a pair of buffers used for communication with other agents of the DPG. We present a specification of the BUFFER module below, which buffers values of abstract type Elem.

```
scheme ELEM = class type Elem end
context: E
scheme BUFFER(E : ELEM) =
    class
        type Buffer = E.Elem*
        value
            empty : Buffer = ⟨⟩,
            put : E.Elem × Buffer → Buffer,
            get : Buffer ~→ Buffer × E.Elem,
            is_empty : Buffer → Bool
        axiom
            ∀ e : E.Elem, q : Buffer • is_empty(q) ≡ q = ⟨⟩,
            ∀ e : E.Elem, q : Buffer • put(e, q) ≡ q ^ ⟨e⟩,
            ∀ e : E.Elem, q : Buffer •
                get(q) ≡ (tl q, hd q) pre ∼is_empty(q)
end
```

The BUFFER scheme has four functions:
— empty: specifies an empty buffer;
— put: used for adding values of type Elem to the Buffer;
— get: used to remove values of type Elem from the Buffer;
— is_empty: Given a Buffer, returns a Boolean which indicates whether the buffer is empty or not.

Three corresponding axioms are used to specify the properties of the functions specified above in terms of their signatures (names and types).

The BUFFER scheme is used by agents in all other modules of the system (STATION, CONTROLLER, ADD_MEMBER, REMOVE_MEMBER and DPG) for communication with other agents of the DPG.

An object B which is an instance of the BUFFER scheme is created as shown below, with the **type Message** replacing the **type Elem**. The object B is therefore an instance of the scheme BUFFER, which buffers messages of abstract type Message. The type message is specified in the section that follows

object B : BUFFER(T {Message **for** Elem })

Passing of messages from one agent of the DPG to another is taken care of by the function **put_msg_in** specified below. Agent "c" puts the message in the buffer of the receiving agent "a". The agent "c" on choosing a message for transmission deletes the message from its buffer and puts the message in the buffer of the receiving agent "a".

```
        put_msg_in : Agent x Agent x Message x DPG → DPG
            put_msg_in(c, a, m, d) ≡
                let
                    buff = stat_buffs(d)(c, a),
                    buff' = put(m, buff),
                    c' = change_msgs(m, c)
                in
                    change_stat_buff(
                        stat_buffs(d) † [(c, a) ↦ buff'], d)
                end
            pre (c, a) ∈ dom stat_buffs(d)
```

There is a unique buffer used for communication between Agents in the DPG. This requirement is specified in the unique_station_buff function below.

```
    unique_station_buff : Station_buffs → Bool
      unique_station_buff(sb) ≡
        (
        (∀ a1, a2 : Agent •
          ((a1, a2) ∈ dom sb ⇒ (a1 ≠ a2),
```

8.3. Messages

Message is defined as a variate type comprising eight alternate messages. From the definition below it is apparent that all the messages have parameters. The corresponding data types of these parameters are specified in the TYPES module. It should be noted that in RSL the types and attributes of non-dynamic entities of a system to be specified are specified in the TYPES module which is subsequently instantiated into a global object T used by other modules of the system.

```
    type
      Message ==
        empty_m |
        message1(n1 : Nonce, a : Agent, k1 : Pub_key) |
        message2(
          n2a : Nonce,
          n2b : Nonce,
          a2 : Agent,
          k2 : Pub_key) |
        message3(n3 : Nonce, k3 : Pub_key) |
        receive_message(
          r_m_n : Nonce-set,
          ar : Agent,
          kr : Pub_key,
          r_m_sk : Shared_keys-set,
          Agent-set) |
        send_message(
          sd_m_n : Nonce-set,
          am : Agent,
          ks : Pub_key,
          sd_m_sk : Shared_keys-set,
          Agent-set) |
        select_message(
          s_m_n : Nonce-set,
          ac : Agent,
          kc : Pub_key,
          s_m_sk : Shared_keys-set,
          Agent-set) |
        broadcast_message(
          b_m_n : Nonce-set,
          ab : Agent_c,
          kb : Pub_key,
          b_m_sk : Shared_keys-set,
          ac : Agent-set,
          b_m_sgkk : Group_key),
```

8.4. The group controller

As illustrated in Fig. 3 below the controller has five possible states of type **Controller_state** specified below.

```
type
Controller_state ==
        Controller_0 |
        Controller_1 |
        Controller_2 |
        Controller_OK |
        Controller_error,
```

The operations of the group controller resulting to the generation of the group controller's view of the group key are specified in the CONTROLLER_NEW module below. The controller uses the change_state_c predicate to change from one state to another.

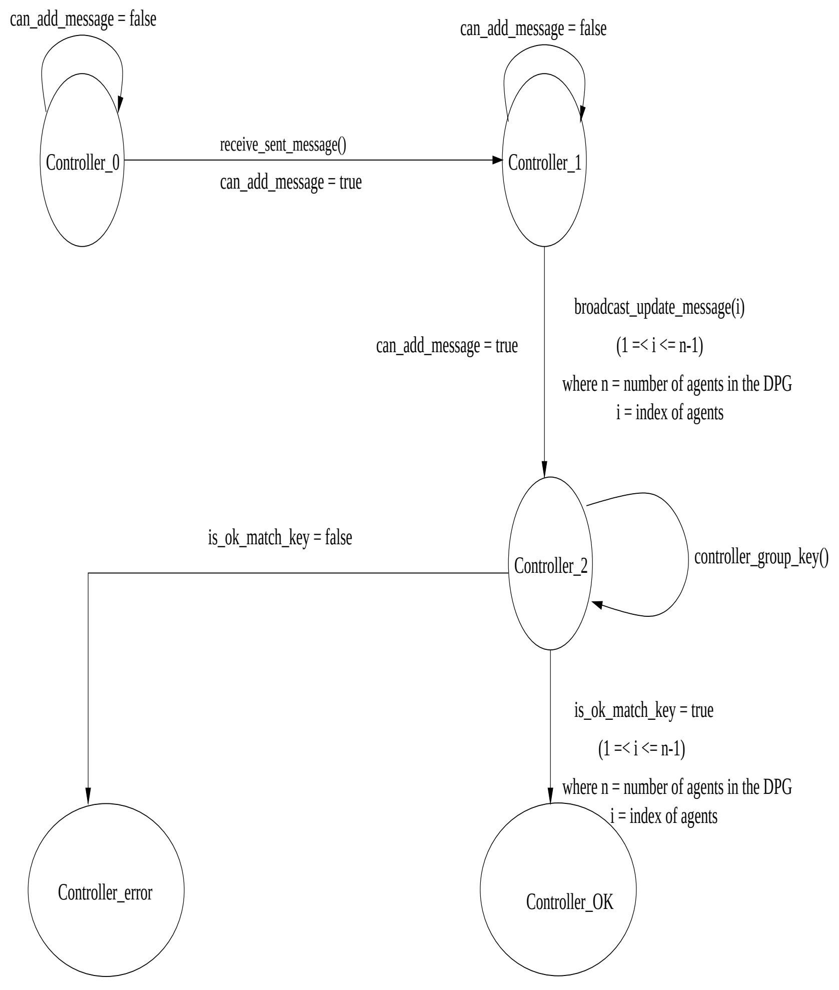

Fig. 3. State transition diagram for the controller in the DPG.

```
context: BUFFER, STATION_NEW, S, T, E, B
scheme CONTROLLER_NEW =
  class
    object S : STATION_NEW
    value
      /*state_c : T.Controller_state → Controller_state
      */
      controller_next :
        T.Agent_c × T.DPG → T.Agent_c × T.DPG
      controller_next(c, d) ≡
        case T.state_c(c) of
          T.Controller_0 → receive_sent_message(c, d),
          T.Controller_1 → broadcast_update_message(c, d),
          T.Controller_2 → is_ok_match_key(c, d),
          T.Controller_OK → (c, d),
          T.Controller_error → (c, d)
        end,
         end
```

Initially the controller is in state "Controller_0". When it receives the **send_message** using the function **receive_sent_message** from the last but one agent in the Dynamic Peer Group (DPG) (agent with index (max_index - 1)), it changes to the next state "Controller_ 1". While in this state, it updates the received message and sends the updated message using the **broadcast_update_message** function, to each and every members of the DPG encrypting each members' sub key with its corresponding public key, after checking to ensure that its certificate is valid. It then changes to the next state "Controller_2". In this state, it computes the group key using the **controller_group_key** function. It then uses the **match_key** function to compare the generated group key to that generated by each and every member of the DPG. If both keys are the same, in all the cases (as indicated by the **is_ok_match_key** function) it then changes to the final state **Controller_OK**. If however both keys are not the same in one or more cases, it changes state to the **Controller_error** state, from which the key generation process is repeated. The controller uses an **if** loop with the index of the agents in the DPG as the control parameter, to loop through all agents in the DPG each time comparing its group key with that generated by the agent that posseses the current index of the loop. The loop starts with index (max_index -1) and ends with min_index (index of the first agent)

In RAISE in other to do justification, the specification should be minimal and functions like broadcast and loops should be avoided. Since we have specified our system so as to eventually do theorem proving (justification), we decided to model this last round of the protocol by getting the group controller to send serial unicast messages to the rest of the DPG, instead of a true broadcast. We however use the word broadcast in the name of the function that sends this serial unicast messages to illustrate the point.

The function **is_controller** specifies the requirement that the highest indexed agent is the group controller.

```
    is_controller : Agent → Bool
    is_controller(c) ≡ indx(c) = max_index,
```

Another role of the Group controller is to add members to the DPG and remove member from the DPG. This requirement is specified in the **can_add_member** and **can_remove_member** functions defined below.

```
Agent_kind == agent | controller
kind : Agent → Agent_kind
controller : Agent_s → Agent_c
can_add_member : Agent → Bool
can_add_member(a) ≡
   kind(a) = controller ∧ indx(a) = max_index,
can_remove_member : Agent → Bool
can_remove_member(a) ≡
   kind(a) = controller ∧ indx(a) = max_index,
```

An additional requirement is that the index of the controller increases by one when it adds a new member into the DPG and the just added member is in the set of agents which constitutes the DPG. The **is_add_member** function below specifies this requirement

```
is_add_member : Agent x Agent-set $\xrightarrow{\sim}$ Bool
is_add_member(a, ac) ≡
   (∀ c : Agent, ac1 : Agent-set, d : DPG •
      is_controller(c) ∧ ac1 = add_agent_set(a, ac) ⇒
         indx(c) = (max_index + 1) ∧
            indx(a) = (indx(c) - 1) ∧ a ∈ ac1 ∧
               is_agent_set(d))
pre a ∉ ac,
```

This function is defined as a partial function because there are situations where it can not be sensibly applied. This situation is specified as a precondition that follows the keyword **pre**. ie to add a new member to a DPG, the member to be added should not already be in the DPG.

Yet another requirement is that the index of the controller decreases by one when it removes an existing member from the DPG and the member to be removed should be in the DPG prior to being removed. The **is_remove_member** function below specifies this requirement

```
is_remove_member : Agent x Agent-set $\xrightarrow{\sim}$ Bool
is_remove_member(a, ac) ≡
   (∀ c : Agent, ac1 : Agent-set, d : DPG •
      is_controller(c) ∧ ac1 = remove_agent_set(a, ac) ⇒
         indx(c) = (max_index - 1) ∧ a ∉ ac1 ∧
            is_agent_set(d))
pre a ∈ ac,
```

8.5. Imperative specification of the broadcast process of the Group Controller

Scheme parameters could be object arrays, this typically occurs in the specification of concurrent systems such as the Authenticated group Diffie—Hellman protocol.

A broadcasting process could be considered as one that inputs values from a single input channel, and then outputs the values to several output channels each ending in some user process. In our system the Group Controller has as input the last upflow message from the n − 1 agent. It updates this message using the **broadcast_update_message** function. The updated message now serves as input to the broadcast process which outputs n − 1 simultaneous messages destined to the remaining n − 1 agents in the Dynamic Peer Group, where n = number of agents in the Dynamic Peer Group DPG.

The broadcast process or rather the scheme defining it is chosen to be parameterized with the kind of data transmitted on the channels.

We therefore need a parameter requirement of the following form,

scheme DATA = **class type** Data **end**.

In addition, the broadcast process is chosen to be parameterized with the channel it inputs from and the channels it outputs to. We therefore need a parameter requirement of the following form,

scheme CHANNEL (D: Data) = **class channel** c : D.Data **end**.

This requirement is parameterized with respect to the type of the channel. A specific requirement is obtained by instantiating the parameterized one with an actual channel type as shown below.

The number of output channels associated with the broadcast process is also made a parameter. This is achieved by parameterizing with an array of output channels, where the array index is an additional parameter satisfying the following requirements.

scheme INDEX =**class type** Index **end**

We now specify the broadcast process as follows:

```
context: T, D, I, INDEX, CHANNEL, ACTUAL_INDEX, ACTUAL_OUT, ACTUAL_DATA,
ACTUAL_IN, CONTROLLER
scheme BROADCAST(
    I : INDEX, D : DATA, IN : CHANNEL(D),
    OUT[i : I.Index] : CHANNEL(D)) =
  class
    value
      broadcast :
        Unit → in IN.c out {OUT[i].c | i : I.Index} Unit
    axiom
      broadcast() ≡
        while true do
          let data = IN.c? in
            ||{ OUT[i].c!data | i : I.Index }
          end
        end
end
```

Here is a description of the parameters of the BROADCAST scheme. The type I.Index contains all the indices of the output-channel array OUT, which corresponds to the indices of the agents in the DPG.

The type D.Data is the type of the data transmitted on all channels. The channel IN.c is the channel that the broadcast process inputs from. Each of the channels OUT[i].c, where i: I.Index is an output channel of the broadcast process.

It is important to note the dependence between the parameters: I and D as referred to in the definition of IN and OUT. This dependence expresses a requirement sharing between the parameters: The I referred to in the definition of OUT is exactly the same I given as the first parameter of the BROADCAST scheme. The same situation is true for D. This dependence ensures the types of input and output data are the same, as there is only one object D and hence only one type D. Data

The definition of the broadcast process can now be described as follows. The process inputs from the IN.c channel and outputs to any of the OUT[i].c channels where i: I.Index. The out access is described by the access description: **out** {OUT[i].c |i: I.Index}.

The axiom means that the process repeatedly input a value from the IN.c channel and then output the value on each of the OUT[i].c channels where i: I.Index.

The output is expressed by the comprehended expression

|| {OUT[i].c!data | i: I.Index}

which represents the parallel composition of all the processes. The outputs are put in parallel and will be carried out at some point in time, leading to broadcasting.

In a DPG comprising four agents including the group controller which is considered here as the broadcast process, the index type are the numbers from 1 to 4.

object ACTUAL_INDEX : **class type** Index = {|i : **Nat** • $1 \leq i \wedge i \leq 4$ |} **end**

The data transmitted on the channel are of type T.Message

context: T
object ACTUAL_DATA : **class type** Data = T.Message **end**

The input channel to the broadcast process is defined as follows:

context: CHANNEL, ACTUAL_DATA
object ACTUAL_IN : CHANNEL(ACTUAL_DATA)

Each user process (agent process) inputs from a channel and outputs to another. The data at the input channel of the user proccess is of type T.Message i. e. the output from the broadcast process and the data at the output channel is also of type T.Message. It is important to note that the operations described so far are those originating from the broadcast process.

context: CHANNEL, DATA
scheme AGENT(D : DATA, IN : CHANNEL(D), OUT: CHANNEL(D))=
class value agent : **Unit** → **in** IN.c **out** OUT.c **Unit end**

The collection of channels output to by the agent process is represented as an array of channels

context: CHANNEL, ACTUAL_DATA, I
object ACTUAL_OUT[i : I.Index] : CHANNEL(ACTUAL_DATA)

9. Model of the system-group key agreement protocol

After providing formal descriptions of the various components of the system, in this section we attempt to compose our system from the (developed) components. We start off by defining the DPG as a set of agents (all group members)with the minimum indexed agent being the first agent of the DPG while the maximum indexed agent is the group controller. This requirement is defined in the predicate **is_agent_set** specified below. We use a DPG comprising four members to illustrate this requirement.

Short record definition is used to specify the type DPG', having as destructors state_dpg, stat_buffs, serv_buffs with corresponding reconstructors change_state_dpg, change_stat_buff, change_serv_buff.

```
type
    DPG' ::
    state_dpg : Dpg_state ↔ change_state_dpg
    stat_buffs : Station_buffs ↔ change_stat_buff
    serv_buffs : Server_buffs ↔ change_serv_buff
    agents_set : Agent-set ↔ change_agent_set,
```

Here the type DPG is defined to have properties defined as part of the type via the subtype predicate **is_agent_set**. The predicate **is_agent_set** specified below defines a DPG as a set of agents comprising four members (agents).

```
type
    DPG = { | d : DPG' • is_agent_set(d) | },

value
    is_agent_set : DPG → Bool
    is_agent_set(d) ≡
       (∀ a1, a2, a3, a4 : Agent, ac : Agent-set •
          indx(a1) = min_index ∧
             indx(a2) = (indx(a1) + 1) ∧
          indx(a3) = (indx(a2) + 1) ∧
       indx(a4) = (indx(a3) + 1) ∧ indx(a4) = max_index ⇒
    first_station(a1) ∧ is_controller(a4) ∧
    agent_set(a4, d) =
      add_agent_set(
        a4,
          add_agent_set(
            a3,
              add_agent_set(
              a2,
                add_agent_set(a1, empty_agent_set))))
    ),
```

The **agent_set** function below defines the set of agents that constitute a DPG in terms of the buffer that the controller uses to communicate with other members of the DPG. It states that all agents that are in the domain of the function **serv_buffs** that maps the function type **Agent_c x Agent_s** to the type **Buffers** and thus represents the buffer between the controller and an agent, make up the current set of agents that constitute the DPG.

```
    agent_set : Agent_c x DPG → Agent-set
    agent_set(c, d) ≡
        {a | a : Agent_s • (c, a) ∈ dom serv_buffs(d)},
```

The following functions indx, next and previous specified below, are used to determine the order of the agents in the DPG.

```
indx: Agent → Index
Defines the next agent in the DPG
next : Agent → Agent
axiom
[next_inj]
        (∀ a1, a2 : Agent •
           indx(a2) = (indx(a1) + 1) ⇒ a2 = next(a1)),
Defines the previous agent in the DPG
previous : Agent → Agent
axiom
[previous_inj]
        (∀ a1, a2 : Agent •
           indx(a2) = (indx(a1) + 1) ⇒ a1 = previous(a2)),
```

We use a finite state machine approach to specify our system. The DPG system module has three states. The **execute_s** function is used to execute the operations in the first state, while the **execute_c** function is used to execute the operations in the third state. In the first state, starting with the first agent of the DPG (agent with index **min_index**) and proceeding in order of occurence as determined by their indexes, all agents of the DPG (except the last but one (agent with index (max_index − 1)) and the controller) make their contributions towards the generation of the group key following the procedure elaborated in the section 8.1 above. In the second state the last but one agent of the DPG makes its contribution, while in the third state, the group controller updates the contribution received from the previous agent of the DPG. The updated contribution is used to generate the controller's view of the group key as well as generate sub keys sent to all agents of the DPG, which are subsequently used by the corresponding agents to generate the group key. The DPG_NEW module below specifies the operation of the system. This module makes use of (the other modules that describe components of the system), to generate the Group key. Each and every agent of the DPG ends up generating the same group key.

```
context: BUFFER, STATION_NEW, CONTROLLER_NEW, TYPES_NEW, E
scheme DPG_NEW =
   class
      object S : STATION_NEW, C : CONTROLLER_NEW
      value
         dpg_next :
            T.Agent_s × T.Agent_s × T.Agent_c × T.DPG →
               T.Agent_s × T.Agent_s × T.Agent_c × T.DPG
         dpg_next(a1, a2, c, d) ≡
            case T.state_dpg(d) of
               T.Dpg_0 → execute_s(a1, a2, c, d),
               T.Dpg_1 →
                  let
                     (a2', d′) = S.s_next(a2, d),
                     (c′, d″) = C.controller_next(c, d′),
                     d″ = T.change_state_dpg(T.Dpg_2, d′)
                  in
                     (a1, a2′, c′, d″)
                  end,
               T.Dpg_2 → execute_c(a1, a2, c, d)
            end,
end
```

The DPG system module uses the **execute_c** and **execute_s** functions to perform a run operation on all agents of the DPG. It uses the index of the agents to control the loop to ensure that all agents in the DPG participate in the key generation process. The specification of the **execute_s** function is given below.

```
context: execute_s :
            T.Agent_s × T.Agent_s × T.Agent_c × T.DPG →
               T.Agent_s × T.Agent_s × T.Agent_c × T.DPG
         execute_s(a1, a2, c, d) ≡
            execute1(a1, a2, c, (T.min_index − 1), d),
         execute1 :
            T.Agent_s × T.Agent_s × T.Agent_c × T.Index ×
            T.DPG →
               T.Agent_s × T.Agent_s × T.Agent_c × T.DPG
         execute1(a1, a2, c, i, d) ≡
            if i ≤ (T.max_index − 1)
            then
               let
                  i = i + 1,
                  a1 = T.owner_i(i),
                  a2 = T.next(a1),
                  (a1′, d′) = S.s_next(a1, d),
                  (a2′, d″) = S.s_next(a2, d′)
               in
                  if i = (T.max_index − 1)
                  then
                     (a1′, a2′, c, T.change_state_dpg(T.Dpg_1, d′))
                  else execute1(a1′, a2′, c, i, d″)
                  end
               end
            else error_message(a1, a2, c, d)
            end,
```

10. System OK theory

We can show that the specified protocol works properly by justifying that: If the agents and controller settle in the ok or error states in the last round of the protocol, then they should both be in a stable state. The SYSTEM_OK_T theory expresses this property.

```
context: T, C, S, DPG_NEW
theory SYSTEM_OK_T :
axiom
   [SYSTEM_OK_T]
      in DPG_NEW |−
         ∀ a1, a2 : T.Agent_s, ns : T.Nonce-set •
            ∃ c : T.Agent_c, d : T.DPG •
               T.state_c(c) = T.Controller_0 ∧
               T.state(a1) = T.Agent₀ ∧ T.state(a2) = T.Agent_0 ⇒
                  let (a1′, a2′, c′, d′) = run(a1, a2, c, d) in
                     T.state(a1′) = T.Agent_OK ∧
                     T.state(a2′) = T.Agent_OK ∧
                     T.state_c(c′) = T.Controller_OK
                  end
end
```

Verification (Informal)

The **run** function and associated functions **completed_ss, completed_sc, completed_s and completed_c** specified and used in the **DPG_NEW** module justify this property.

```
context: run :
        T.Agent_s × T.Agent_s × T.Agent_c × T.DPG →
          T.Agent_s × T.Agent_s × T.Agent_c × T.DPG
      run(a1, a2, c, d) ≡
        let (a1′, a2′, c′, d′) = dpg_next(a1, a2, c, d) in
          if
            completed_ss(a1′, a2′, d′) ∧
            completed_sc(a2′, c′, d′)
          then (a1′, a2′, c′, d′)
          else run(a1′, a2′, c′, d′)
          end
        end,
```

```
context: completed_ss : T.Agent_s × T.Agent_s × T.DPG → Bool
      completed_ss(a1, a2, d) ≡
        completed_s(a1) ∧ completed_s(a2),
      completed_sc : T.Agent_s × T.Agent_c × T.DPG → Bool
      completed_sc(a, c, d) ≡
        completed_s(a) ∧ completed_c(c),
      completed_s : T.Agent_s → Bool
      completed_s(a1) ≡
        T.state(a1) = T.Agent_OK ∨
        T.state(a1) = T.Agent_error,
      completed_c : T.Agent_c → Bool
      completed_c(c) ≡
        T.state_c(c) = T.Controller_OK ∨
        T.state_c(c) = T.Controller_error,
```

The **run** function has four arguments: agent, agent, controller and DPG. The first two arguments represent two agents with index (i and i+1) currently involved in message transmission for ($i \geq min\_index \wedge\ i \leq\ max\_index$) where i is the index of the agents, with min_index being the index of the first station and max_index the index of the group controller. The **run** function calls the **execute_c and execute_s** functions through the **dpg_next** function. These functions together execute all the operations in the other modules of the system, using the index of the agents to cover all the current agents of the DPG, dealing with two neigbouring agents (having index i and i + 1) in each loop.

The **execute_c** function for example, starting with the index of the group controller (max_index), uses the outer "**if**" loop to execute all the functions in the STATION module (which models all the agents of the DPG except the group controller) and CONTROLLER module (which models the group controller). Each loop deals with two neigbouring agents (having index i and i + 1), representing the first two arguments of the **run, execute_s and execute_c** functions. After every loop the index of the agent is decremented by one and the operation repeated for the next agent pair. The inner "**if**" loop is used to check if the index of the first agent of the DPG has been attained. if this condition is **true** the DPG changes to the DPG_OK state, if not the outer "**if**" loop carries on executing each time decrementing the index by one and dealing with two neigbouring agents at a time, until the condition of the inner "**if**" loop is **true**.

The **run** function then uses **complete_ss** and **complete_sc** functions on the results of the **execute_c** function to test if all the Agents of the DPG settled in either an **Agent_OK** or **Agent_error** states. If this condition is **not true**, the run operation executes for all the states of the DPG using the **dpg_next** function each time to change to the next DPG state, until the condition is satisfied.

The **complete_c** function states that the operations of the CONTROLLER module are completed if the controller is either in the **Controller_OK** or **Controller_error** states.

The **complete_s** function states that the operations of the STATION module are completed if all the agents settle in either the **Agent_OK** or **Agent_error** states.

The rigorous arguments presented above justify the **SYSTEM_OK** theory.

11. Conclusion and future work

In this paper, we have presented a formal model of a Generic Group Key Agreement protocol. We justified the safety of the protocol and showed that the protocol works properly by informally verifying the SYSTEM_OK theory.

The specification of the model of the system got srutinized through the generation and verification of confidence conditions evolved from all the modules used in the specification of the system. It should be noted that confidence conditions are generated by RSL automaticaly on the specified modules. This confidence conditions give confidence to our specification by showing that our specification is correct.

RSL allows algebraic, applicative or, imperative styles for specifying systems. In this paper we provided an abstract applicative specification of the Authenticated Group Diffie — Hellman protocol. This applicative style makes it easier to prove properties of the protocol using the RAISE Tools.

Future work will involve coming up with an imperative specifications of the abstract model specified here, and using the RAISE Justification editor to prove the properties informally analysed in this paper. We shall also introduce concurrency in the last round of the protocol where the group controller broadcast corresponding sub_keys (subsequently used to generate the group key) to each and every member of the DPG. In this paper we got the group controller to send serial sub_keys in the last protocol round, though we provide a separate imperative specification of the broadcast process.

Our long-term goal is to come up with a fully specified and verified generic Authenticated Group Key Agreement protocol for Dynamic Peer Groups (DPG).

Such a fully specified and verified system could find application in a wide variety of fields where information security is a critical issue; examples include electronic commerce, electronic bank transfers, video conferencing etc.

Our model shall be extended to include issues like periodic re-keying after using the key for a given duration, or to encrypt or sign a given volume of data, which ever is more frequent.

References

[1] Amir Y., Ateniese G., Hasse D. et al. Secure group communication in asynchronous networks with failures // Intergration and Experimentation. 1999. Aug. 22.

[2] Ateniese G., Steiner M., Tsudik G. Authenticated group key agreement and friends // Proc. 5th ACM Conf. on Computer and Communication Security, Nov. 2–5, 1998, San Francisco, CA.

[3] Ateniese G., Chevassut O., Hassee D. et al. The design of a group key agreement API. 1999. Aug. 25.

[4] Becker C., Willie U. Communication complexity of group key distribution // Proc. ACM Conf. on Computer and Communication Security, Nov. 1998.

[5] Boneh D. The Decision Diffie — Hellman problem // Proc. Third Algorithmic Number Theory Symp., Lecture Notes in Comp. Sci. Vol. 1423. B.: Springer-Verlag, 1998. P. 48–63.

[6] Diffie W., Hellman M. New direction in cryptography // IEEE Trans. on Information Theory. IT-22(6): 644–6. Nov. 1979.

[7] George C. Proving Safety of Authentication Protocols: a Minimal Approach. UNU/IIST Report No. 154 Feb., 1999.

[8] The RAISE Language Group. The RAISE Specification Language. BCS Practitioner Ser. Prentice Hall, 1992.

[9] The RAISE Method Group. The RAISE Development Method. BCS Practitioner Ser. Prentice Hall, 1995.

[10] Harney H., Muckenhim C., Rivers T. Group Key Management Protocol (gkmp) Architecture. INTERNET DRAFT, Sept. 1994.

[11] Ingemarsson I., Tang D., Wong C. A conference key distribution system // IEEE Transactions on Information Theory. 1982. Vol. 28, No. 5. P. 714–720.

[12] Lim C. H. Authenticated Key Distribution for Security Services in Open Networks. Information and Communications Research Center, Future Systems, Inc. May 19, 1997.

[13] Mauw S., Veltink G. J. Algebraic Specification of Communication protocols. Cambridge Tracks in Theoretical Computer Science 1993. No. 36.

[14] Menezes A., Oorschot P. Van, Vanstone S. Handbook of Applied Cryptography. CRC Press Series on Discrete Mathematics and its Application. CRC Press, 1996.

[15] Mohanty H., George C. Specifying a Communication Protocol And Composing Transaction Schedules For a Mobile Environment. UNU/IIST Report No. 142, Aug. 3, 1998.

[16] Steiner M., Tsudik G., Waider M. Diffie — Hellman key distribution extended to groups // ACM Conf. on Computer and Communication Security, pages 31–37, March 1996.

[17] Steiner M., Tsudik G., Waider M. CLIQUES. A new approach to group key agreement // IEEE Intern. Conf. on Distributed Computing System, May 1998.

[18] Tanaka T., George C. Proving Properties of a Security Protocol Specified in RSL. UNU/IIST Report No. 143, July, 1998.

Received for publication January 14, 2002