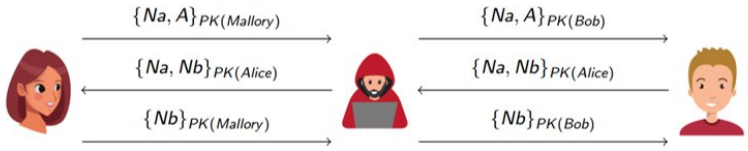
The Needham-Schroeder protocol in B2Scala

Let us now code the **Needham-Schroeder** protocol and exhibit a computation that reflects G. Lowe's attack.

To incorporate an attack scenario:

- We introduce **Mallory** as an intruder.
- Our initial focus is on declaring nonces and public keys for all protocol participants **Alice**, **Bob**, and **Mallory**.



Needham-Schroeder Protocol

The following token declarations will facilitate this process:

```
val na = Token("Alice_nonce")
val nb = Token("Bob_nonce")
val nm = Token("Mallory_nonce")

val pka = Token("Alice_public_key")
val pkb = Token("Bob_public_key")
val pkm = Token("Mallory_public_key")
```

It will be beneficial for future reference to designate the three participants. This can be accomplished through the following token declarations:

```
val alice   = Token("Alice_as_agent")
val bob     = Token("Bob_as_agent")
val mallory = Token("Mallory_as_intruder")
```

To clarify the origin and recipients of messages within the protocol, the encrypted messages introduced in **B2Scala** are expanded into si-terms of the structure **message(Sender, Receiver, Encrypted_Message)}**. Additionally, for emphasis on the messages' roles in the protocol, the encrypted messages are renamed as it **encrypt_n**, with **n** denoting their sequence number. The resulting declarations are as follows:

```
case class encrypt_i  (vNonce: SI_Term, vAg: SI_Term, vKey: SI_Term)    extends SI_Term
case class encrypt_ii (vNonce: SI_Term, wNonce: SI_Term, vKey: SI_Term) extends SI_Term
case class encrypt_iii(vNonce: SI_Term, vKey: SI_Term)                  extends SI_Term

case class message    (agS: SI_Term, agR: SI_Term,  encM: SI_Term)      extends SI_Term
```

Si-terms are introduced to signify the initiation and termination of sessions between *Alice* and *Bob*. These terms are declared as follows:

```
case class a_running(vAg: SI_Term) extends SI_Term
case class b_running(vAg: SI_Term) extends SI_Term
case class a_commit (vAg: SI_Term) extends SI_Term
case class b_commit (vAg: SI_Term) extends SI_Term
```

The coding of **Alice** in B2Scala is as follows:

```
val Alice = Agent {
     GSum( List(bob,mallory), Y =>
           {  tell(a_running(Y)) * tell( message(alice, Y, encrypt_i(na, alice, public_key(Y))) ) *
              GSum( List(a,nb,nm),  WNonce =>
                     { get( message(Y, alice, encrypt_ii(na,WNonce,pka)) ) *
                       tell( message(alice,Y,encrypt_iii(WNonce,public_key(Y))) ) *
                       tell( a_commit(Y) )
                     })
           })
 }
```

We are now ready to code the behavior of **Alice**, **Bob**, and **Mallory**:

Alice's initial action is to start a session, allowing for the possibility that Mallory might intercept it instead of Bob. This is represented by offering a choice between Bob and Mallory using the *GSum([bob, mallory])(…)* construct, denoted as **Y**.

Alice then informs about the session initialization using the *a_running(Y)* si-term and proceeds to send the first encrypted message containing her nonce, her identity, and *Y*'s public key. The sender and receiver are respectively *Alice* and *Y*. Subsequently, Alice awaits the second encrypted message with her nonce and, presumably, Bob's nonce, encrypted with her public key. As the second nonce is unknown, a new choice is introduced using the *WNonce* si-term. Finally, Alice sends the third encrypted message with this nonce, encoded with *Y*'s public key, and concludes the session by notifying the *a_commit(Y)* si-term. Notably, *public_key(Y)* calls a Scala function returning the public key for the given argument *Y*.

The coding of **Bob** in B2Scala is as follows:

```
val Bob = Agent {
   GSum( List(alice,mallory),  Y =>
         {  tell(b_running(Y)) * GSum( List(alice,mallory), VAg =>
              { get( message(Y,bob,encrypt_i(na,VAg,pkb)) )  *  tell( message(bob,Y,encrypt_ii(na,nb,public_key(Y))) ) *
                get( message(Y,bob,encrypt_iii(nb,pkb)) )    *  tell( b_commit(VAg) )
              })
         })
   }
```

In the role of an intruder, Mallory intercepts and relays messages from Alice and Bob, potentially altering certain elements if the messages are encrypted with Mallory's public key. This holds true for all three types of messages exchanged between Alice and Bob. It involves three *GSum* choices corresponding to the three unknown arguments *VNonce*, *VAg*, and *VPK* of the message. In each case, Mallory's approach is to acquire the message and resend it, modifying the public key if he can decrypt the message, i.e., if *VPK* matches his public key.

The coding of **Mallory** in B2Scala is as follows:

```
lazy val Mallory:BSC_Agent = Agent {

  ( GSum( List(na,nb,nm), VNonce =>
      { GSum( List(alice,bob), VAg =>
             {  GSum( List(pka,pkb,pkm), VPK =>
                    {  get( message(alice,mallory,encrypt_i(VNonce,VAg,VPK)) ) *
                        ( if ( VPK == pkm) {  tell( message(mallory,bob,encrypt_i(VNonce,VAg,pkb)) )  }
                                    else
                                           { tell( message(mallory,bob,encrypt_i(VNonce,VAg,VPK))  )  } ) * Mallory
                    })
             })
      })
   )    + ...
```

To wrap up the protocol encoding in **B2Scala**, a **bHM-formula *F*** is defined. This formula is characterized by excluding the initiation of a session between *Bob* and *Alice* othe n one hand, and on the other hand, by mandating the conclusion of the session initiated by Bob with Alice. These requirements are articulated through the **bf-formulae** *inproper_init* and *end_session*, as detailed below:

```
val inproper_init = not( bf(a_running(bob)) or bf(b_running(alice))  or  bf(b_commit(alice)))
val end_session = bf(b_commit(alice))
```

Formula *F* is encoded recursively by mandating *F* after a step that satisfies *inproper_init* and terminating the computation once a step is completed that causes *end_session* to hold. This recursive specification is articulated as follows.

```
val F = bHM { (inproper_init * F) + end_session }
```

Computations are initiated by invoking the following Scala instructions:

```
val Protocol = Agent { Alice || Bob || Mallory }

val bsc_executor = new BSC_Runner
bsc_executor.execute(Protocol,F)
```