Problem 1:

The first problem is a simplified version of the Shared Key Authentication (SKA) protocol. This simplified version is also similar to the one described in the lecture "4.1 Authentication" and shares the same vulnerability. This makes the protocol not secure and A and/or B can reach the wrong conclusion if an attacker is present.

```
A --> B:  {N_A}K        where N_A is a fresh nonce

B --> A:  {N_B}K, N_A   where N_B is a fresh nonce

A --> B:  N_B
```

This shows the logic of the protocol, and the issue arises if the attacker preforms a "Reflection Attack."

```
A --> B:  {N_A1}K        – First session

                         where N_A1 is a fresh nonce

B --> A:  {N_B1}K, N_A1  – First session

                         where N_B1 is a fresh nonce

A --> B:  {N_B1}K        – Second session

                         where N_B1 is the nonce from the first session

B --> A:  {N_B2}K, N_B1  – Second session

                         where N_B2 is a fresh nonce

A --> B:  N_B1           - First session
```

Problem 2:

The second problem is another version of the Needham-Schroeder Protocol, which uses a trusted Key Distribution Center to distribute a session key between the two principals. The protocol seems safe, and I am unable to find any vulnerabilities. This is what the protocol looks like:

```
1.  A --> KDC: A,B,N
2.  KDC --> A: {N,A,B}K_A, {N,K_AB}K_A
3.  KDC --> B: {N,A,B}K_B, {N,K_AB}K_B
4.  A --> B:  {N+1}K_AB  compare value N+1 with contents of message 3
5.  B --> A:  {N-1}K_AB  compare value N-1 with contents of message 2
```

To prove that the protocol is safe it is best to go through each step. First A initiate's connection with the server that it wants to speak with B, it also includes a nonce N that will later be used to recognize the freshness of the messages. The server then sends the session key K_AB to both A and B encrypted with the public key of the respective clients. By encrypting the session key, it ensures that it is not possible for any attackers to view the session key. A and B encrypts an incremented and decremented version of the nonce respectively and sends them to the other. The nonce is then decrypted by the receiving parties and compared to the nonce they got from the server, confirming that both have the correct session key and that the message is fresh.

If there should be a problem with this protocol, I think it would originate from the usage of the same nonce throughout the whole exchange. Since the nonce is not encrypted in the first interaction between A and the server, anyone can view it and if you then manage to get the session key K_AB you could easily pretend to be either of the principals. Therefore, an improvement could be that the server generates a new nonce when distributing the session key to the principals. This is possible since the first nonce is only important for A to recognize the freshness of the exchange between the server and A. Such an improvement would look like this:

```
1.  A --> KDC: A,B,N1
2.  KDC --> A: {N1,A,B}K_A, {N2,K_AB}K_A
3.  KDC --> B: {N1,A,B}K_B, {N2,K_AB}K_B
4.  A --> B: {N2+1}K_AB  compare value N+1 with contents of message 3
5.  B --> A: {N2-1}K_AB  compare value N-1 with contents of message 2
```

Problem 3:

The third problem shows a protocol that allows A to suggest a session key K_AB to communicate with B. It uses the server S to securely share the session key using the keys shared with the server K_AS and K_BS. The respective principals A, B and S ignore any message that is older than 5 seconds.

```
A --> S:  A, {T_A, B, K_AB}K_AS   where T_A is current time

S --> B:  {T_S, A, K_AB}K_BS      where T_S is current time
```

This protocol is flawed, and an intruder T can force B to adopt an instance of K_AB that is arbitrarily old. There are two possible ways this protocol is flawed. Firstly lets assume the protocol works the other way around;

```
B --> S:  B, {T_B, A, K_AB}K_BS   where T_B is current time

S --> A:  {T_S, B, K_AB}K_AS      where T_S is current time
```

If this is the case, the attack of the intruder would look something like this.

A --> S: A, {T_A, B, K_AB}K_AS	
S --> T(B): {T_S, A, K_AB}K_BS	T intercepts the message meant for B
T(B) --> S: B, {T_B, A, K_AB}K_BS	T pretends to be B and sends a message to A
S --> T(A): {T_S, B, K_AB}K_AS	T intercepts the message meant for A
T(A) --> S: A, {T_A, B, K_AB}K_AS	T pretends to be A and sends a message to B
S --> B: {T_S, A, K_AB}K_BS	

The intruder can then loop through steps 2-5 as long as it needs to and eventually force B to adopt an arbitrarily old instance of K_AB.

If the protocol does not work the other way around, the attack would look something like this. Only downside of this approach is that it can only send K_AB keys that are at a maximum of 5 seconds old.

A --> S: A, {T_A, B, K_AB}K_AS	where T_A is current time
T(A) --> S: A, {T_A, B, K_AB}K_AS	T keeps a copy of A --> S, and must send it within 5 seconds
S --> B: {T_S, A, K_AB}K_BS	where T_S is current time