Smartcard-Login into Gemstone

An exercise in applied security

by

Alfred Wullschleger,
Swiss National Bank

Disclaimer

-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

This Material is a copy of the presentation at the ESUG 2003, Bleed,
Slovenia.

It has been developed by Alfred Wullschleger,
Swiss National Bank.

Any liability is explicitely excluded.
This publication completely reflects opinions of the author.
The Swiss National Bank shall not be bound by any
information contained herein.

You may send comments
to: wully@bluewin.ch
or to: alfred.wullschleger@snb.ch

-----BEGIN PGP SIGNATURE-----
Version: PGP 8.0

iQA/AwUBP1MlRIBkqNDRsW06EQINYACg63d2yK0GCdIz0f2IQcfWupyx1fwAoO4Q
px5moaJuaAkPTCCgtxq5+iNK
=bMoJ
-----END PGP SIGNATURE-----

Talk and Technical Discussion

- Two Parts
  - first Part (this Talk): Presentation of the ideas
  - second Part: Technical discussion
    - code examples
    - demonstrations
    - details of the implementation
      - as requested by the audience

The Author

- Smalltalker since 1992, never ending enthusiasm
- Project OVID at FIDES Informatik (1992-1999)
- Leading the project OASE at SNB since 1999
  - Financial statistics from Swiss banks and Swiss companies
  - Fully object oriented application using VW3.0 and Gemstone as development base

Motivation

- Gemstone Login uses username and password
  - Both are sent over the network in clear text
  - passwords are typically „trivial“ by User requirement?!
- Smartcards (SC) have many advantages
  - Very high security
    - User must have the card and a password for the card
  - Even relatively trivial passwords are acceptable since the SC is completely disabled after 3 trials

Smartcards at SNB

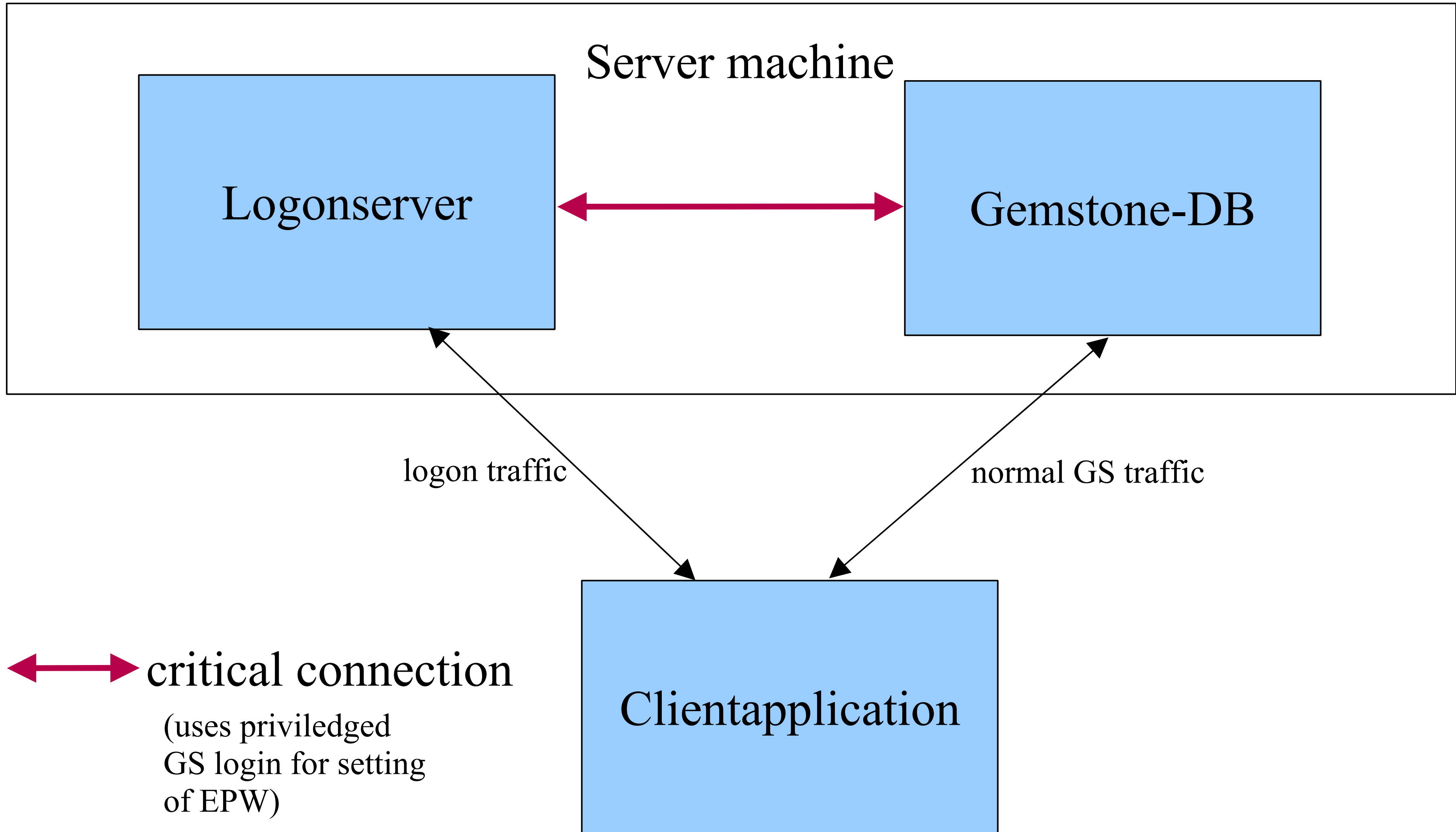
- Smartcards will be introduced in the Bank during 2004
- Goal is Single Signon
  - OASE is one of the first applications which today are ready for Smartcard logon

Smartcard Login

How to implement?

Gemstone Prerequisites

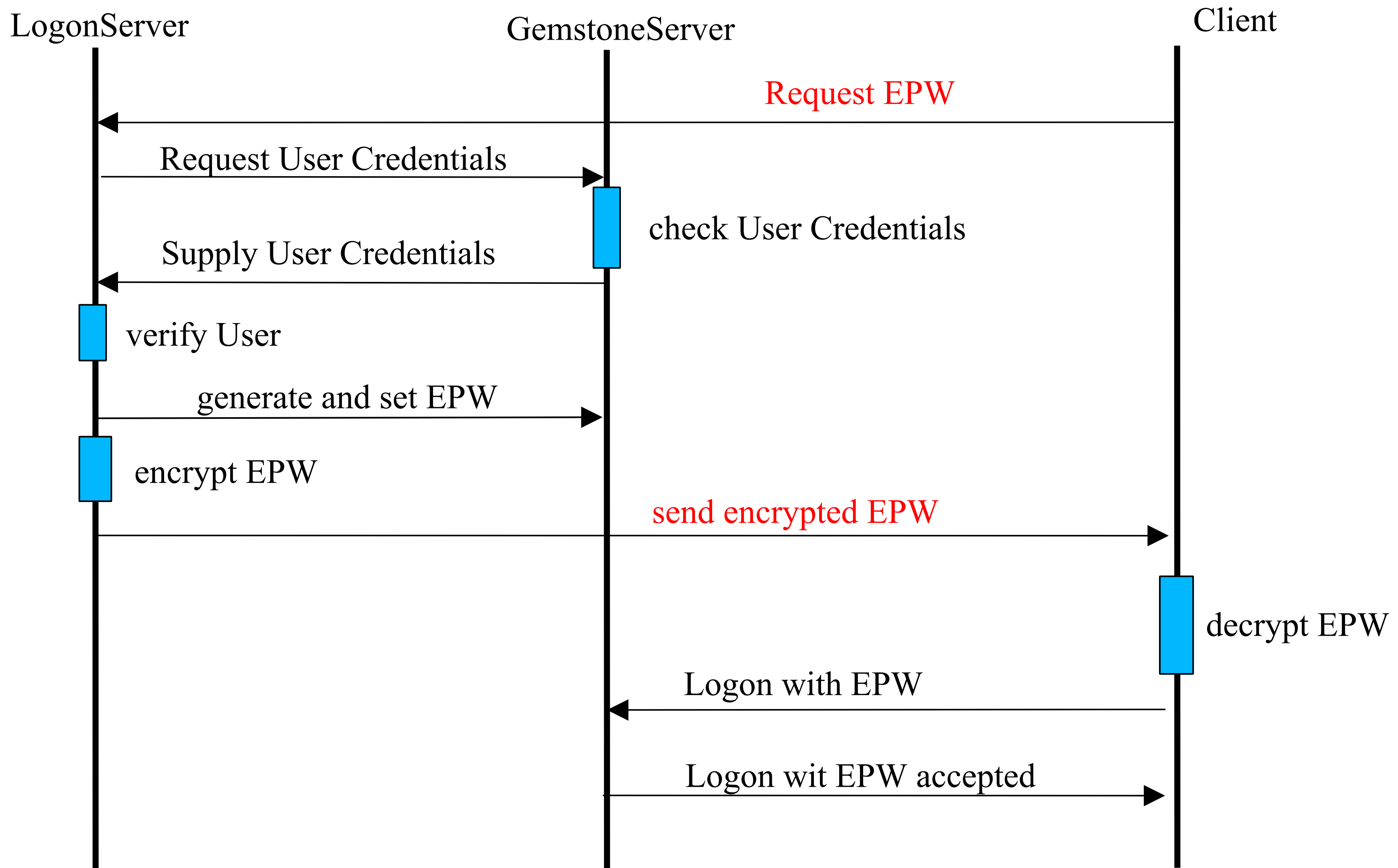
- Logon uses username and password
  - Password size 1024 byte maximum
  - Password expiration can be set to exactely one logon
- So, we can use a „complicated“ One-Time-Password, we call it EPW (from german)
  - format can be choosen appropriate
- Smartcard can be combined with GS:
  - use a Logonserver


Server machine
Logonserver
Gemstone-DB
logon traffic
normal GS traffic
critical connection
(uses priviledged
GS login for setting
of EPW)
Clientapplication

Logonserver (LS)

- LS runs on the same machine as the GS server(s)
  - this guarantees the exchange of critical data with GS without network traffic
  - handles all GS servers on the same machine
  - handles all client logon requests for these servers
  - is implemented as a headless VW3.0 application
    - so it runs wherever we want (AIX, Win2000, Linux, WinXP...)
    - runs as TCP/IP-Server

LS basic operation



LS basic operation

- the client requests his EPW from LS
- LS verifies with GS, that user is ok
  - (i.e. has credentials and a user account)
- LS generates EPW and sets it in GS for one use
- LS encrypts the EPW and sends it to the client
- the client decrypts the EPW and makes normal GS login with his username and the EPW

LS has two concurrent modes

- SC-RSA (Smartcard-RSA-mode)
  - The user uses a Smartcard with RSA-signature keypair
- PW-DH (Password-Diffie-Hellman-mode)
  - The user has no Smartcard
    - a forgotten Smartcard should not prevent the user from logging into GS
    - this mode must be activated in GS for each user individually by administrators, if in SC-RSA-mode before

We need a secure data exchange Client <> LS

- Exchange of secret data through encryption
  - we use Blowfish block cipher
- Each session Client-LS must have a session key
  - can be generated by using Diffie-Hellman (DH) technique
  - how does DH work?

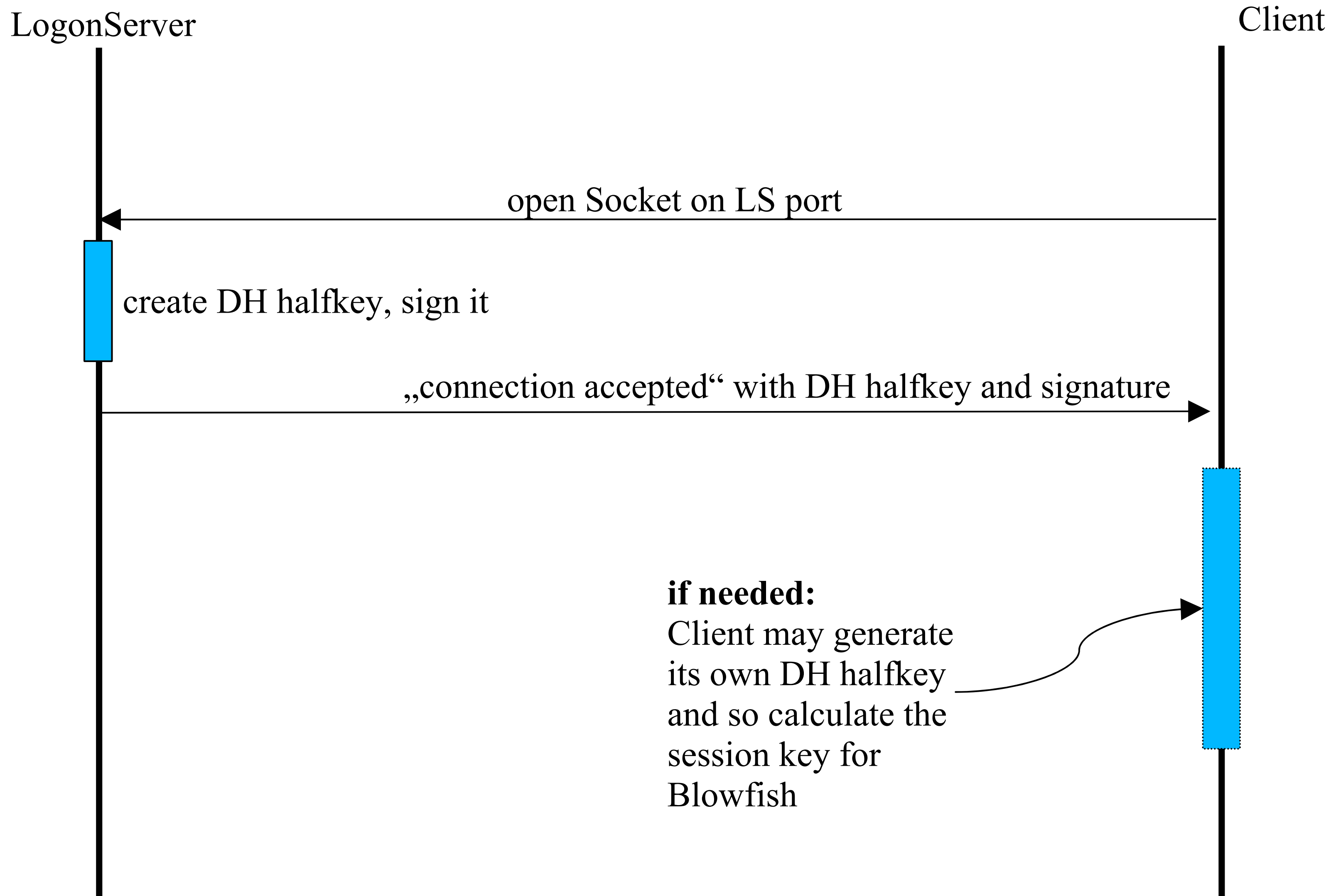
Diffie-Hellman part1

- creation of one secret for two parties
- based on the cyclic group $\boldsymbol{Z_p}$ = {1,2,3...p}, where p is prime.
  - this is equivalent to multiplication of natural numbers modulo p
  - p size large, typically > 256 bits
- use a common base b ∈ $\boldsymbol{Z_p}$, e.g. 65537
- each party selects a secret random number $r_j > 1$, typically large. j=1,2

Diffie-Hellman part2

- each party calculates pub_j = (b ** r_j) mod p, so, $pub_j \in \boldsymbol{Z_p}$
- both parties publicly exchange their pub_j
  - we call pub_j the DH-Halfkey of party j
- each party calculates key = (pub_j)**r_i mod p
  - (j = other, i = self)
- Since (b**r_i)**r_j == (b**r_j)**r_i, both parties get the same key!

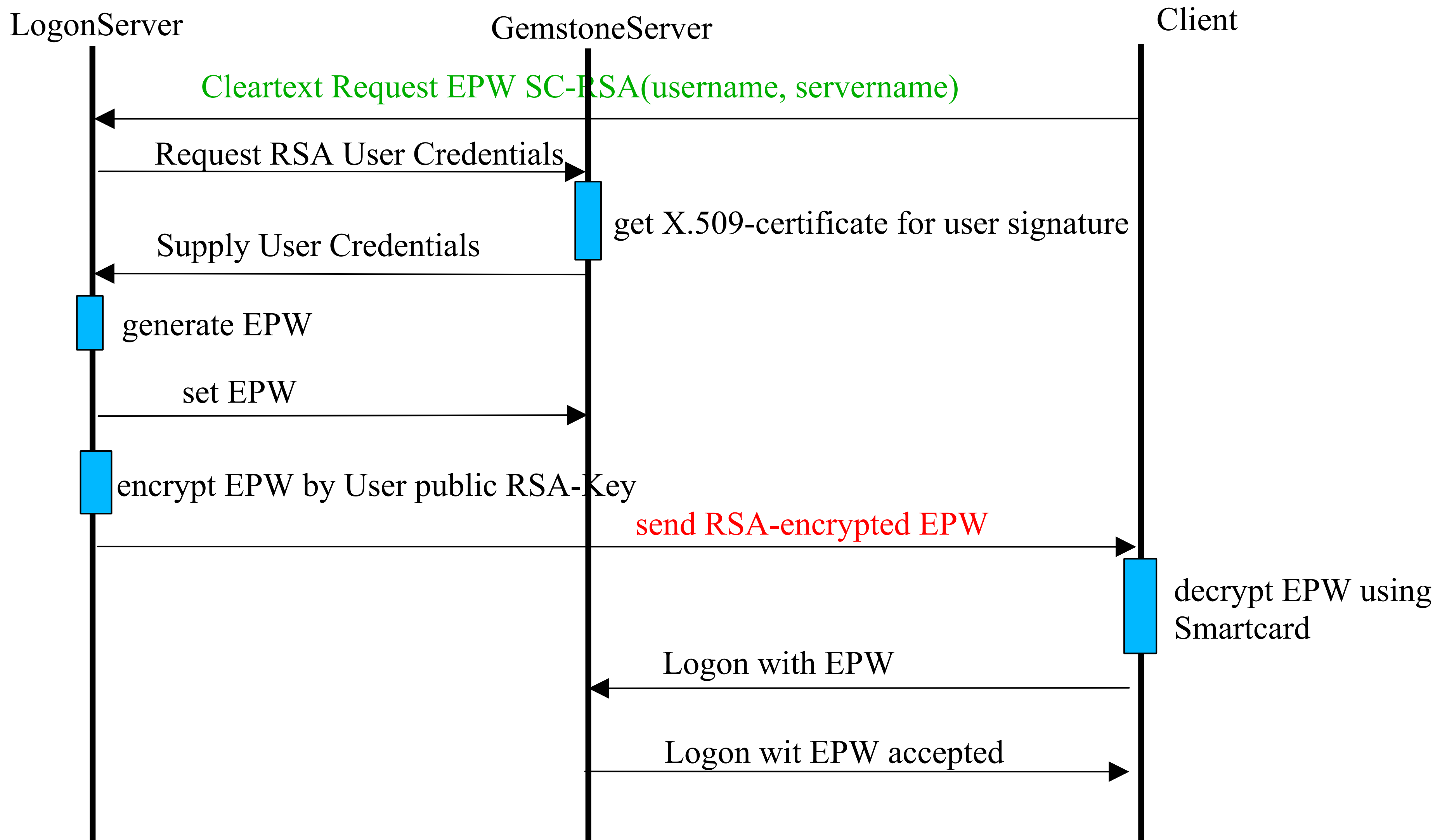
LS connection start



LS connection start (both modes)

- As soon as a connection to LS is established
  - LS creates a Diffie-Hellman halfkey for generation of a session key
  - LS signs this halfkey, so that a user can verify, that he is connected to a legal LS
  - LS sends this information as a „connection accepted“ message to the client
  - this is done irrespective of the mode wished by the client

SC-RSA Overview



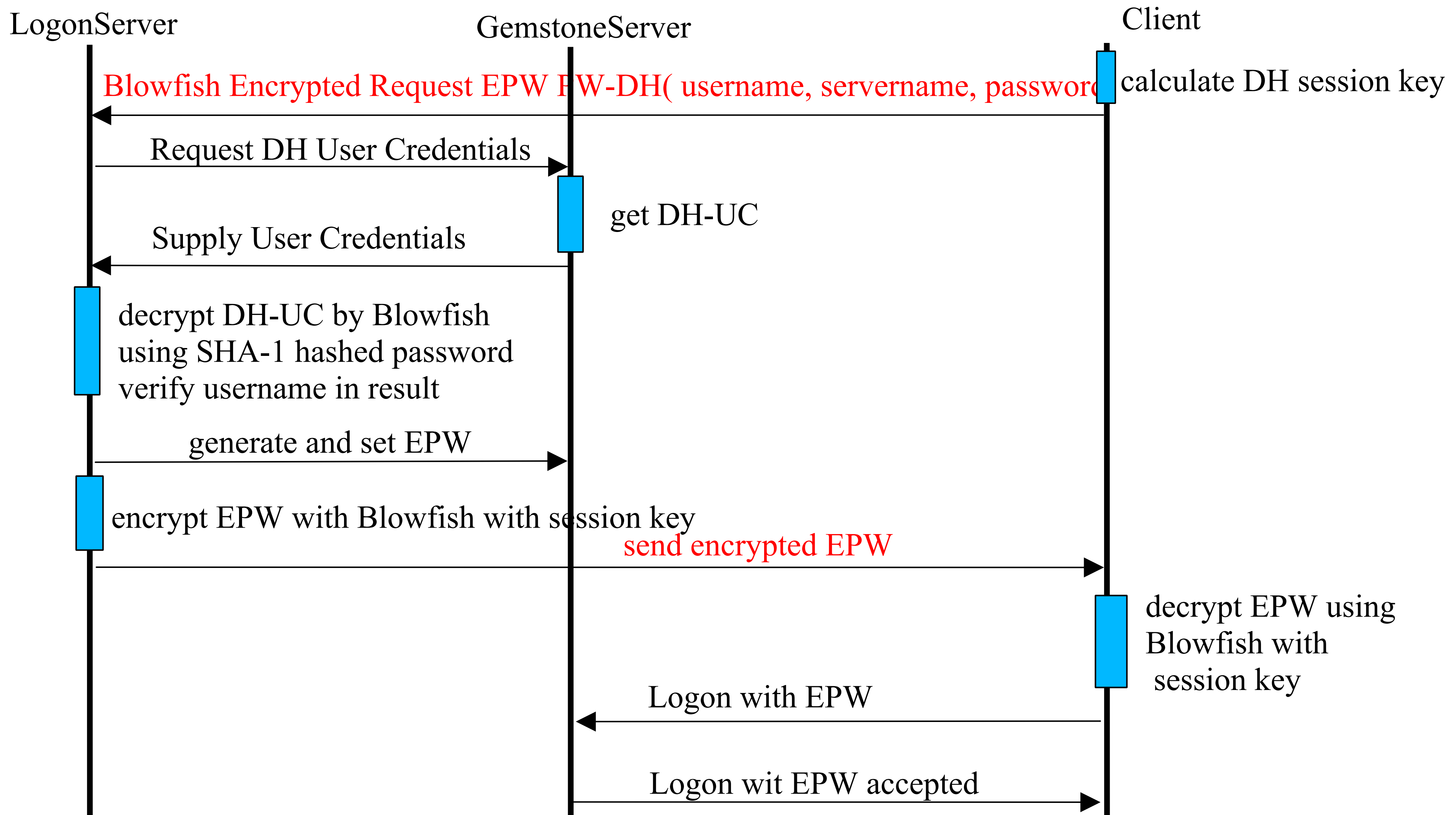
SC-RSA part1

- User requests EWP in SC-RSA-mode
  - parameters: GS servername, GS username
  - sent in clear text to LS
- LS requests X.509-Certificate for the user from GS
  - LS generates EPW and sets it in GS
  - LS RSA-encrypts EPW using signaturekey from the X.509-Certificate

SC-RSA part2

- LS sends encrypted EPW to client
  - if the user has the corresponding Smartcard, he can decrypt the EPW, if not, the information from LS is useless
- user decrypts EPW with his Smartcard
  - makes login in GS, using GS username and EPW

PW-DH Overview



PW-DH part1

- client requests EWP in PW-DH-mode
  - parameters: GS servername, GS username, password
  - client generates halfkey for DH-Keyexchange
  - creates with the halfkey from LS the session key for blowfish encryption
  - sends EPW request encryted to LS together with his halfkey
- LS decrypts EPWrequest
  - by that gets servername, username and password

PW-DH part2

- LS gets encrypted PW-DH-credentials for username from GS
  - GS stores PW-DH-credentials as blowfish-encrypted username plus randomdata
  - password is hashed with SHA-1 to get a blowfish key
    - Watch out: hashing does not help if password is trivial
  - These data are generated by interaction of an administrator with the user
    - when RSA-SC is used, PW-DH-credentials are ereased

PW-DH part3

- LS decrypts PW-DH-credentials using the password, if ok:
  - LS generates EPW and sets it in GS
  - LS blowfish-encrypts EPW using the sessionkey
- client receives and decrypts EPW
  - makes login in GS, using GS username and EPW

Implementation issues

Encryption,
Formats of EPW etc.

RSA basics

- use two secret primes p and q:
  - get n = p*q, a large number (1024 bit)
  - select a number e with gcd(e,(p-1)*(q-1))=1
  - e typically 65537 (or 17 or 3)
    - n and e form the public key for encryption
  - calculate d so that ed == 1 mod(p-1)*(q-1)
    - n and d form the secret decryption key

RSA implementation

- Security is based on factorization being a hard problem
- RSA is fully implemented in VW7.*
  - we have ported it to VW3.0
    - was very easy

Blowfish

- Block cipher defined by Bruce Schneier, 1994
  - encrypts 64-bit-Blocks
- uses keylength between 128 and 448 bit
  - we use 256 bit keys
- we use CBC-Mode
  - each block is XORed with the preceeding encryptedBlock, first block with an InitialVector

Sizes

- RSA-keys on SC and signing key of LS: 1024bit
- Diffie-Hellman parameters
  - p: currently 256 bit
  - base = 65537
- EPW
  - 20-byte random ByteArray converted to LargePositiveInteger
    - giving a passwordstring of 48 to 49 digit characters

XML for all requests/responses

- XML as exchange format between Client and LS
  - To avoid escape sequences like ' etc., we are using integer formats throughout all exchanges which use encryption
  - The ByteArrays are converted using „asLargePositiveInteger"
    - this is compatible with PKCS#1-specification of conversion between bytes and integers

LS Answer on Socket accept:

- <XmlResponse>
  <Validation>true</Validation>
  <DhCode>15883425833416642465360808848836377119677827852925265980146286745390962263576</DhCode>
  <ServerDHSignatur>183709006026842385792416705349556245188475128876202759601481894750869790269995927978776405195325121051883508886524132786861869875368718986668112950424220306601831492161585675580495028687106941544499429842209447096825728927211059446466436584972168382665360133253168788716612548951267550854884951583394172645 34</ServerDHSignatur>
  </XmlResponse>

- User can verify the Signature of the LS

SNB
BNS

SCHWEIZERISCHE NATIONALBANK
BANQUE NATIONALE SUISSE
BANCA NAZIONALE SVIZZERA
BANCA NAZIUNALA SVIZRA

SC-RSA basic format

- SC-RSA-request from client to LS is unencrypted:
  - <RSAEPWRequest>
    <Server>lightning</Server><User>rog</User>
    </RSAEPWRequest>
- SC-RSA-response from LS to client:
  - <EinmalPasswort>25403648156676275885066008022408475387522358683839072136920924589230875478943511911386973276958545070613995304664027444844259941647099339694254712622465008895928176115059189039910207366590660509176631584961534508759536105592032741240560953779756974615610233367097913666219451704733004530461339320939262107515</EinmalPasswort>

SC-RSA improved

- we could improve the protocol by requesting, that the client has to sign the LS halfkey
  - so, we avoid setting EPW for a user, who has no corresponding X.509-certificate on GS for the SC
    - has more computing overhead
  - could avoid denial-of-service-attacks
    - disable user, if repeatedly illegally requesting an EPW
- this is implemented for ticket requests (see later)

PW-DH request

- Internal PW-DH-Request:
  - <XmlEPWRequest>
<Server>lightning</Server>
<User>awu</User><Password>xxxxxx</Password></XmlEPWRequest>
- encrypted PW-DH-Request:
  - <XmlEPWRequest>
<DhCode>281574748271158343297420533227126327934689087781014207105602432679325465102 64</DhCode>
<EinmalPasswortRequest>42170146095638651584330954627501148111195214290485313498389165669484140998639678401900317185968075492053120111139089385114866495946563018655593550775694587428583677231448885711958259445637238672195235016034223474372862471711709551746300760031120608509510132548629714428</EinmalPasswortRequest>
</XmlEPWRequest>

PW-DH response

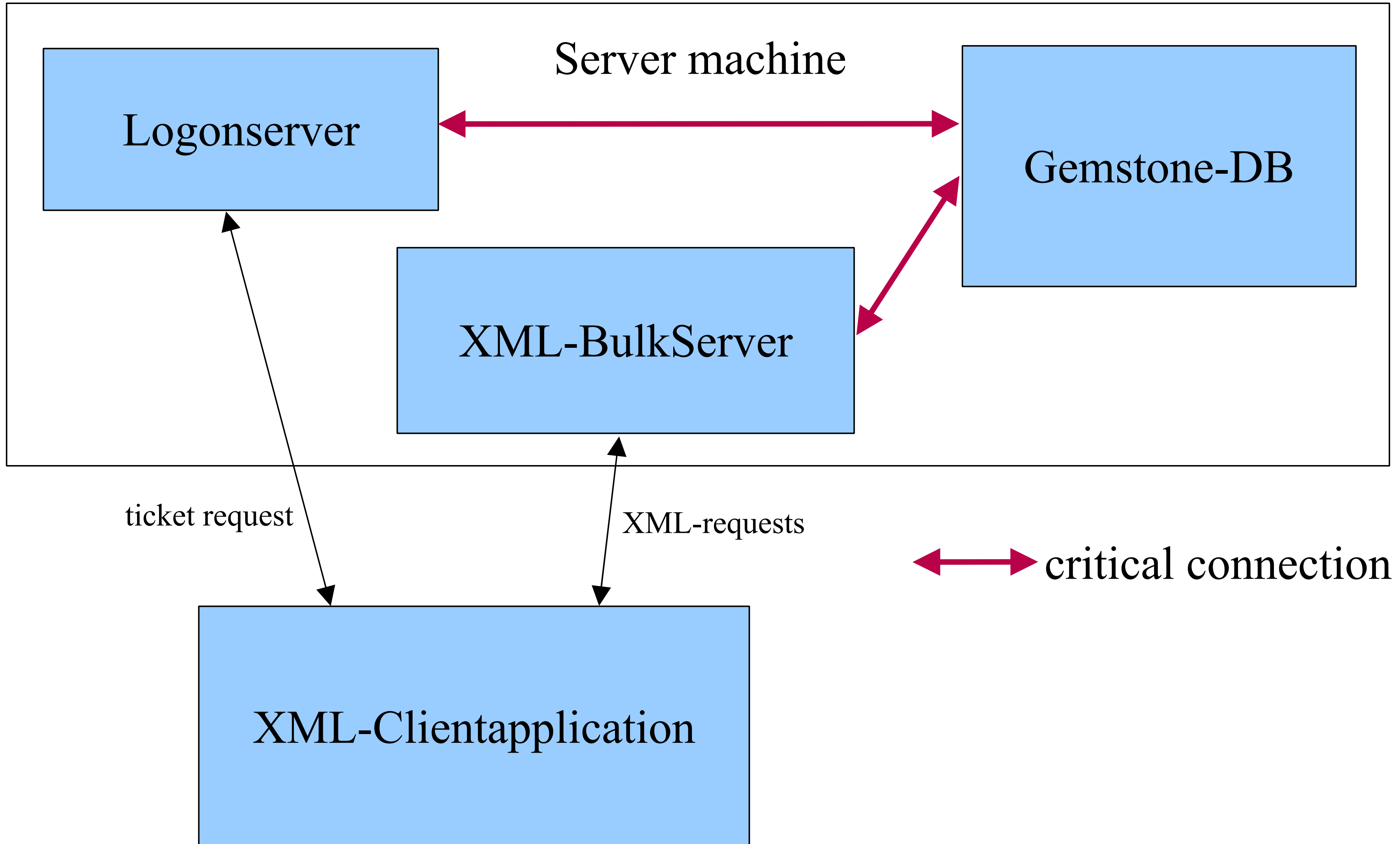
- Encrypted Response from LS:
  - <EinmalPasswort>39101687102161078272434121164568457066871259497097936869261979591784192846211854573079584829211200321873938068169308035706515761190918837390618022514077795
</EinmalPasswort>

Other Functions of LS

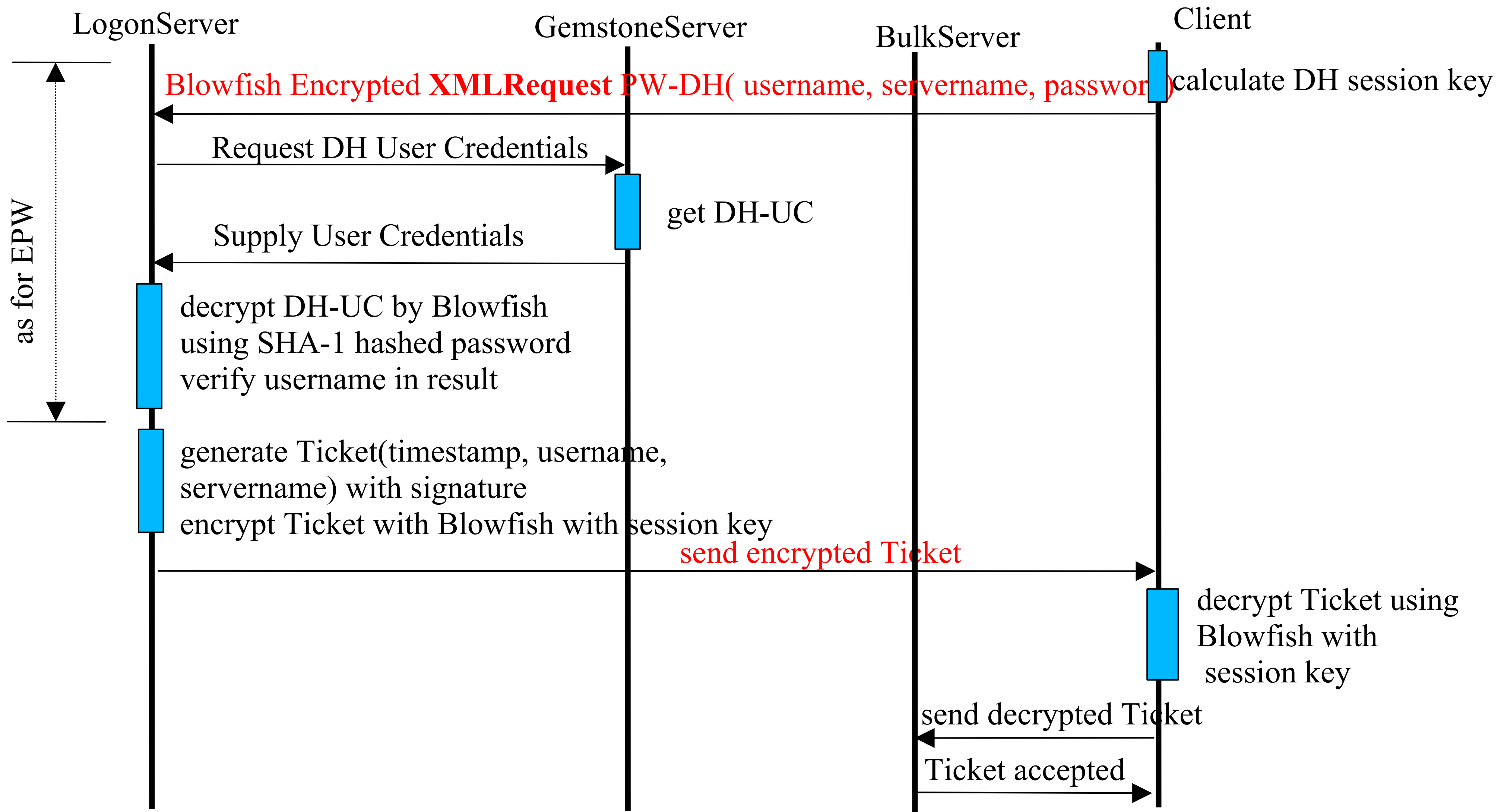
Tickets

XML-Bulkserver

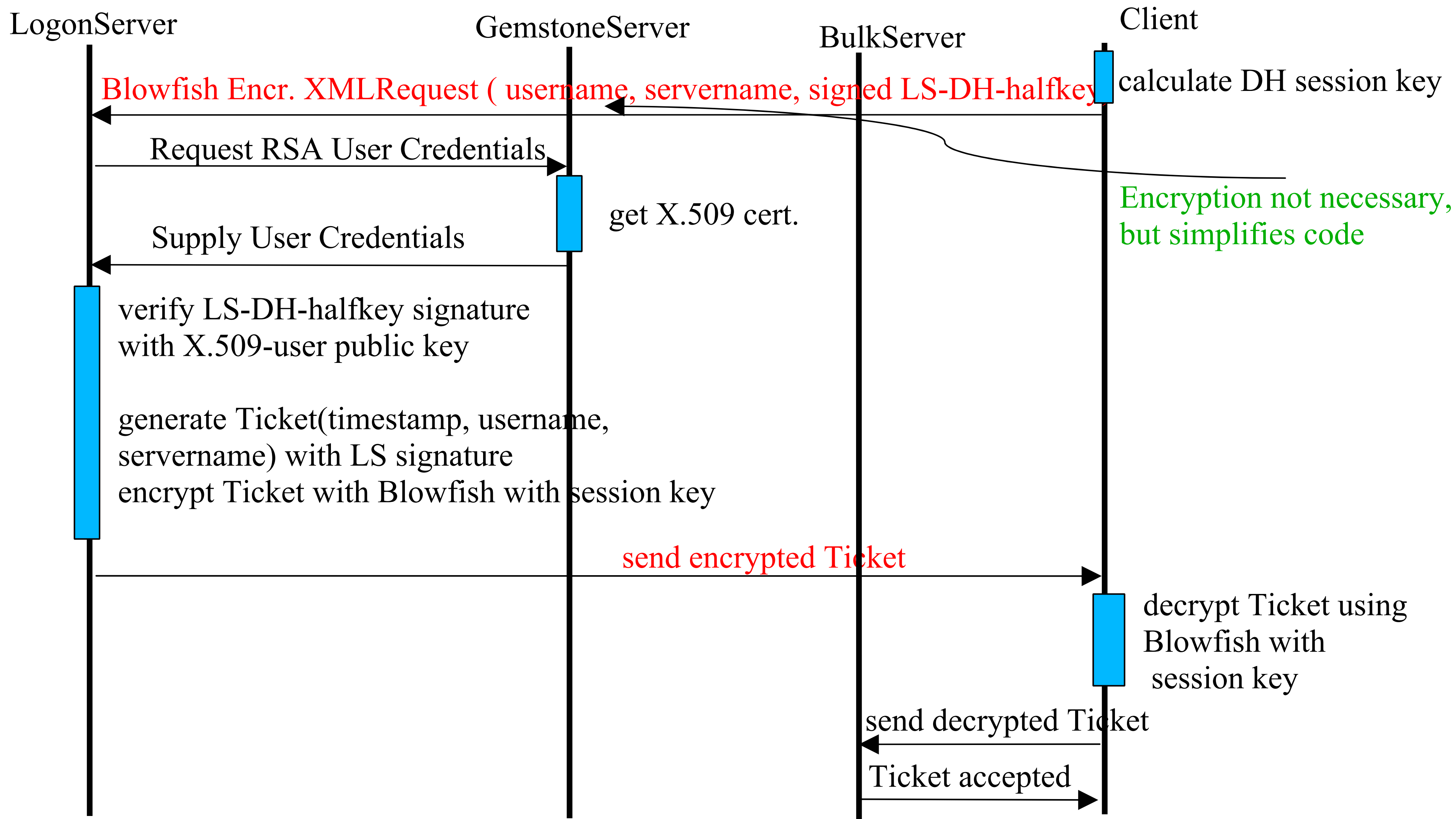
- OASE data can be requested from the so called XML-Bulkserver which is fully XML-based
  - Like LS a headless VW3.0-application
- The user must login into GS to access the data
- How can this be achieved when the user must supply an EPW?
  - by an XML-Ticket which allows access to GS


Server machine
Logonserver
Gemstone-DB
XML-BulkServer
ticket request
XML-requests
critical connection
XML-Clientapplication

XML-Ticket Overview for PW-DH



XML-Ticket Overview for SC-RSA



XML-Ticket basics

- The user requests an XML-Ticket from the LS
- When the user is allowed to access the database, LS creates a ticket
  - contains servername, username and a timestamp for validity checking.
  - the ticket is signed by LS
- The ticket must be presented to the Bulkserver within a short period (currently < 10 seconds).

XML-Ticket checking

- The Bulkserver checks the ticket:
  - verifies the signature of LS
  - checks the timestamp
    - also verifies, that this ticket has not already been used
  - uses the servername to access GS
    - today, we do not restrict access for read for legal users, because of that there is no check necessary for the username

XML-Ticket modes

- PW-DH and SC-RSA use the same technique for sending the ticket to the client:
  - the ticket format has no difference for both modes
  - the ticket is encrypted using blowfish with the DH-sessionkey

XML-Ticket formats

- The TicketRequest differs for both modes
  - parameters: servername and username
- in SC-RSA the user signs the DH-halfkey from the LS with his signing key
- in PW-DH the credentials for servername, username are checked as for EPW using password
- these requests are both sent encrypted by blowfish

TicketResponse

– <XMLTicket>
<Server>dpssM700</Server><User>awu</User>
<XMLTicketTime>3233767397000</XMLTicketTime>
<XMLTicketSignature>150227959672771554223658356609188848131504930433885153358415628484046533885074283485204708129887432046841005815553609797690724046154100779363097261513324905718435929147111194229037132871109267790706342077618299796649268994761120827180240661674334688850191633869392302702031009392187738491257307568961771360 37</XMLTicketSignature>
</XMLTicket>

- identical for both modes

Encrypted TicketRequest Format

- both modes send encrypted with Blowfish:
  - <XMLLogonRequest>
<DhCode>40219239307692214180145053340545080877353522480766790930681240902947674828299</DhCode>
<EncryptedXMLLogonRequest>101377057082959074520545822832481995281355486029905747935665003154347311601168913300557488907343971523088523696699844827287172410408573882043716659853473297753681242918145743635996082099283667341448648205936302301062332051528870082493408610500435018279009052905430115234</EncryptedXMLLogonRequest>
</XMLLogonRequest>

Internal SC-RSA TicketRequest

– <XMLRSARequest>
<Server>dpssM700</Server><User>awu</User>
<Password>10694266580389396165223778257128233571575284804865
488271645277633172859414869405045021075475426683763871005096
512550742078122566663290545340367303214617163684950078288221
884273972865236822265399295831080513274720629788515005292693
213055680355212564287254503022928044905763429004378965569364
748530842326832384</Password>
</XMLRSARequest>

- <Password> contains the signature of the DH halfkey of the LS

Internal PW-DH TicketRequest

- <XmlEPWRequest>
  <Server>dpssM700</Server><User>awu</User>
  <Password>xxxxxxx</Password>
  </XmlEPWRequest>

- Same format as for EPW

Batch-Tickets

- We plan, to introduce also Tickets for Batch-Processing
  - the user requests a ticket for a batch job at some time at some date
  - the LS delivers the ticket
    - the batch job is responsible for the secure deposit of the ticket until used
  - the batch job presents the decrypted ticket at the correct time/date and can be processed

That's it!

Questions?