IBM Software Group Enterprise Networking Solutions
z/OS® V1R11 Communications Server

z/OS V1R11 Communications Server
Scalability, performance, constraint relief, and accelerator

z/OS Communications Server Development, Raleigh, North Carolina

This presentation describes the enhancements in z/OS V1R11 Communications Server for scalability, performance, constraint relief, and acceleration. This theme is a major area of enhancements in z/OS V1R11 Communications Server.

Scalability, performance, constraint relief, and accelerators

- ✓accept_and_receive API enhancements
- ✓TCP/IP path length improvements
- ✓Virtual storage constraint relief
- ✓TCP throughput improvements for high-latency networks
- ✓NSS private key and certificate services for XML appliances
- ✓TCP/IP support for system z10 hardware instrumentation

The new asynchronous version of the accept_and_receive sockets API is targeting high-volume servers on z/OS, such as WebSphere Application Server. Such servers repetitively issue three sockets calls for every new connection. This enhancement collapses those three sockets API crossings into one thereby saving processor resources and improving response time.

One of the sockets-related control blocks has moved into 64-bit common storage from ECSA. This change provides virtual storage constraint relief (VSCR) on high workload systems.

In addition to the general drive to reduce path-length, this release has specific enhancements to improve performance when securing Enterprise Extender traffic with IPSec.

All items on this list are described in more detail in the next slides, except the last bullet, TCP/IP support for system z10 hardware instrumentation. z/OS V1R11 Communications Server now uses the z/OS MVS CSVDYLPA service to load its IP load modules. Using CSVDYLPA gives the z/OS MVS Contents Supervisor awareness of the location and attributes of z/OS Communications Server load modules and entry points. Vendor utility functions intended to map z/OS Communications Server code can now use z/OS MVS services CSVQUERY or CSVINFO to obtain the location of z/OS Communications Server load modules and entry points. The primary reason for such mapping is to perform detailed performance analysis.

IBM Software Group Enterprise Networking Solutions

General TCP/IP path length improvement objectives

- ITR = ETR/CPU busy, where ETR is transaction rate (throughput)
  - Internal Throughput Rate is how much work the system can do at 100% busy
  - ETR is difficult to impact so focus is reducing processor consumption
- Goal is to provide release-to-release ITR improvement
  - Continual process to improve overall System z price/performance
  - Communications Server ITR goal in z/OS V1R11:
    - **Reduce Communications Server's processor consumption for request/response workloads**
    - **While not elongating network latency, and while also providing 31-bit common storage (ECSA) constraint relief**
- The challenge:
  - z/OS V1R11 ECSA constraint relief item (using 64-bit CSA for socket control block)
    - Introduces new, cycle-intensive addressing mode-switching coupled with save/restore of high-order GPR halves
    - Had to overcome path-length growth due to 64-bit memory access
  - z/OS V1R10 path-length is good; much effort went into providing significant ITR gains

Page 3

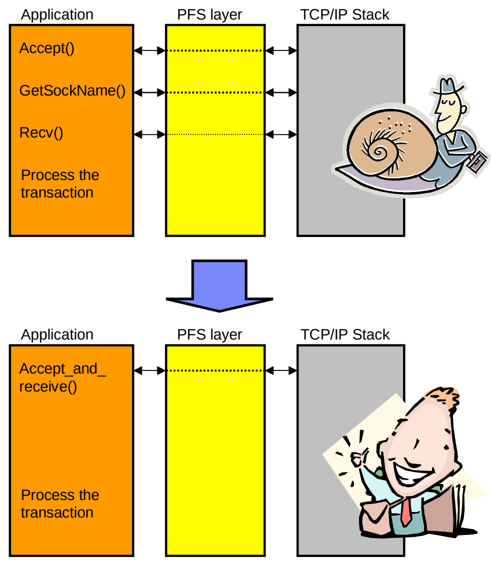
The metric commonly used to relate system throughput and processor consumption is Internal Throughput Rate (or **ITR**). z/OS is committed to release-to-release improvements in the ITR metric. One common approach for improving ITR is to reduce the software pathlength (or machine cycle consumption) for various workloads. z/OS V1R11 Communications Server has a performance goal to improve ITR, while not increasing network latency, and while also providing substantial 31-bit common storage constraint relief.

The major challenge in achieving a pathlength improvement in z/OS V1R11 is that we'll first need to overcome pathlength GROWTH that comes into play during 64-bit memory accesses. This is related to the ECSA-constraint relief item, where the socket control block moved to 64-bit common storage.

Also, z/OS V1R10 performed quite well in the key benchmarks.

Asynchronous accept_and_receive sockets call

- The accept_and_receive call has existed for a few releases
  - BPX1ANR
- It combines three sockets API crossings into a single API crossing
  - Reduces latency and processor time for server applications that receive connections
- z/OS V1R11 adds these capabilities to the accept_and_receive call:
  - 64-bit support
    - BPX4ANR
  - Asynchronous support
    - BPX1AIO
    - BPX4AIO (64-bit)
- Is available to be exploited by all server implementations on z/OS



The accept_and_recv call BPX1ANR was introduced in OS/390 R7 as a means of optimizing the performance of TCP server applications that immediately issue a recv call. The main idea here was that accept() and recv() can be combined into a single API call, reducing the overhead of traversing both the USS and the TCP/IP layer two times for each API. It also provided the ability to retrieve the local IP address associated with the new connection on the same call, eliminating the need for a getsockname() to be issued. The number of API calls was reduced from three to one, boosting performance.

In z/OS V1R11, support has been added in BPX4ANR so that 64-bit applications can exploit accept_and_recv.

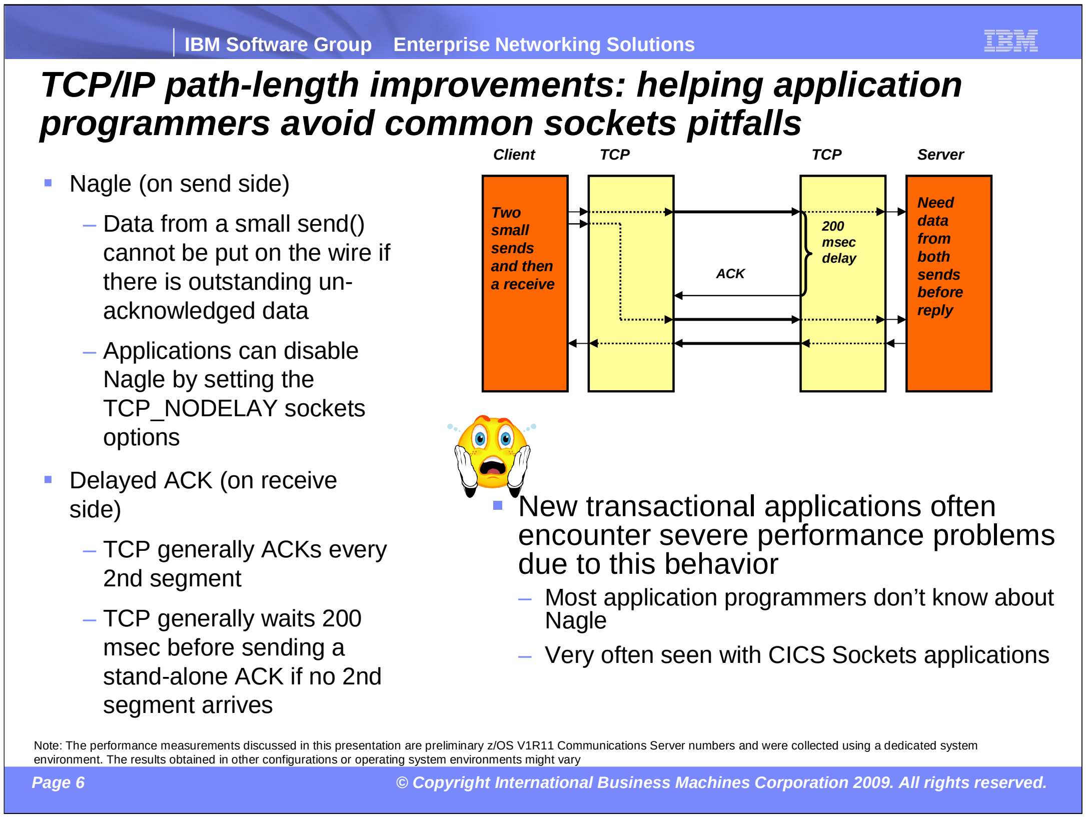
Support has also been added so that accept_and_recv processing can be done using asynchronous I/O by way of BPX1AIO or BPX4AIO.

Finally, enhancements have been made at the TCP layer. TCP listeners that are performing accept_and_recv calls now prioritize established connections which have data available to complete the recv() part of the accept_and_recv() call over connections which have not yet received data. This processing is more efficient in that it mitigates delays in receiving data by favoring well behaved connections.

General TCP/IP path length improvements in z/OS V1R11

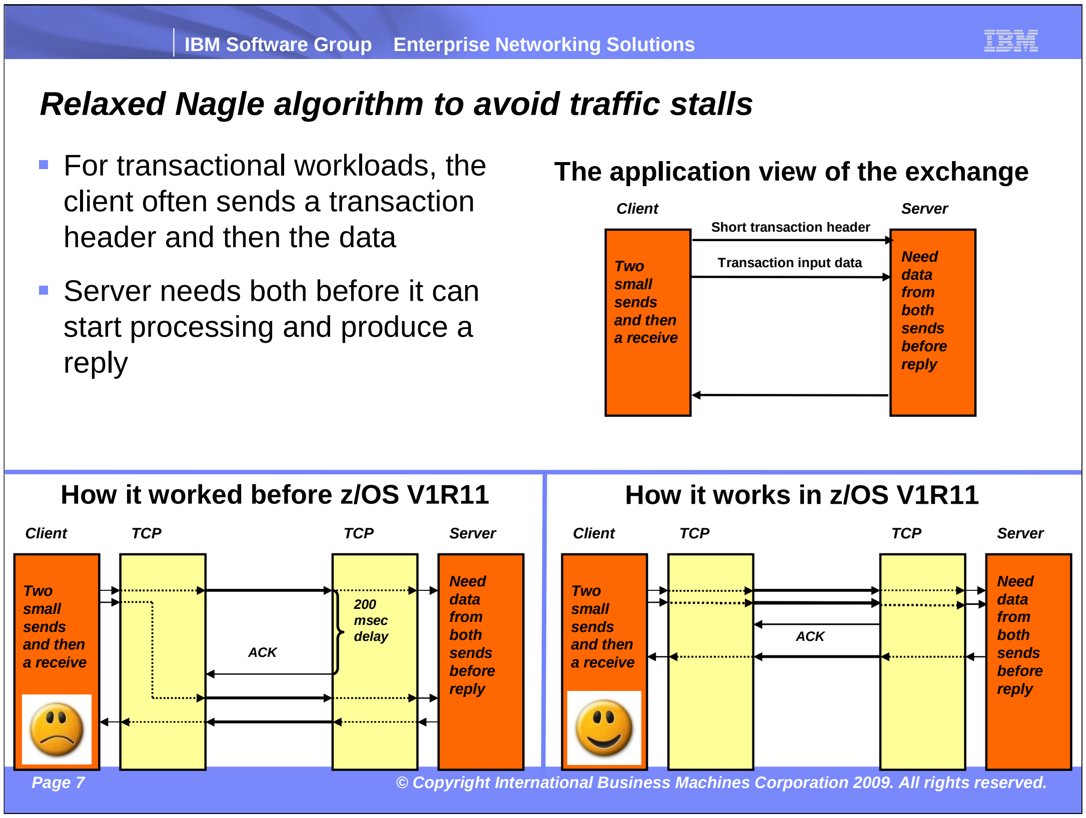
- Use z/Architecture 64-bit arithmetic instructions to maintain double-word SNMP and diagnostic counters
  - As opposed to using earlier (System 390) 31-bit instructions which need to load up and store a pair of registers
- Exploit asynchronous cache pre-fetching
- Minimize TCP/IP data-path references to the new 64-bit Socket Control Block, to avoid address-mode switching
- Code scrubbing certain critical paths, new fast-paths for normal cases

The strategy for Communications Server path length reduction is to reduce memory access latency. Solutions include using newer z/Architecture instructions for maintaining double-word SNMP counters, exploiting a cache-line pre-fetch facility in System z hardware, and minimizing the number of 64-bit socket control block references made along the TCP/IP data path. Some code scrubbing was also done, although V1R10 had already removed much of the low-hanging fruit from the mainline paths.



The Nagle Algorithm has been a standard component of TCP send-side flow control since the mid-1990s. Its intent is to preserve bandwidth, by limiting the number of small data segments that might be sent-and-unacknowledged on the TCP connection. If an application performs a socket send of a small number of bytes and there is already unacknowledged data outstanding on the connection, the Nagle algorithm disallows this new data from being immediately transmitted. Instead, the new data queues and is pushed out some time in the future, in response to receipt of acknowledgements for the already-outstanding data. The Nagle algorithm is enabled by default. You can disable it on a per-connection basis using the TCP_NODELAY SetSockOpt.

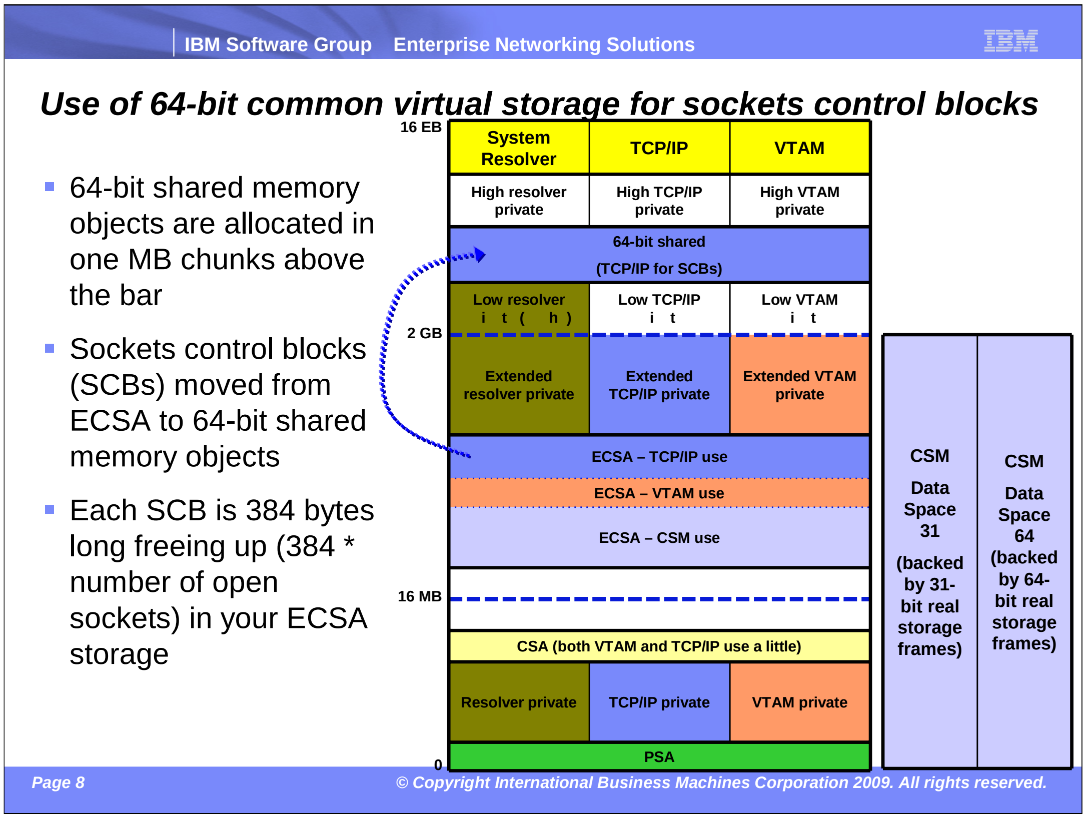
On the receive side, the concept of delayed acknowledgements is in place to both conserve bandwidth and minimize processor consumption. Most TCP stacks employ an ACK-EVERY-OTHER packet scheme. If a workload pattern is "**inbound data packet-outbound data packet**", it's common for no immediate, stand-alone, ACK to be generated for the inbound data packet. Rather the ACK of the inbound packet is piggy-backed on the next outbound data packet, thereby avoiding one flow on the link. This approach works well to reduce the number of packets flowing. Since there is some processor consumption involved in generating and processing a stand-alone ACK, this approach also reduces processor consumption on both sides of the connection.



We'll now describe a type of traffic pattern that is very exposed to traffic stalls, due to interaction of the Nagle Algorithm with delayed acknowledgements. The traffic pattern is not of a symmetric **inbound packet-outbound packet** variety; rather this traffic pattern has multiple small packets heading in one or both directions. In this example, the first flow from the client is a control packet that primes the server for the next flow. The second flow from the client is the actual **request** portion of the transaction. The server receives the request, does some processing, and generates a **response**. So conceptually, the transaction is back-to-back send flows from client to server, with a single send flow from server back to client.

Now here's what happens in reality. On the local side, the data from the first Socket send flows on the wire, unaffected by the Nagle algorithm. There is no unacknowledged data on the connection at this point, so the Nagle algorithm lets it through. But this first Send flow is just a control flow that gets the server prepared for the important second flow. The remote node uses a delayed-ACK scheme, meaning it does not immediately acknowledge this first inbound data packet; rather, it starts a 200 millisecond clock. Back on the local side, the application now issues the second socket send, which is the actual **request** flow for the transaction. But now since there is unacknowledged data on the connection (because the control flow is not yet acknowledged), the Nagle algorithm disallows immediate transmission of this send data. This new send data has to wait on the connection until the acknowledgement to the first send is received. And since the remote node uses a delayed-ACK scheme, it takes approximately 200 milliseconds before the ACK arrives. So this type of workload flow takes at least 200 milliseconds to transmit a complete request to the remote node. Transaction rate is therefore be limited to five transactions per second at best. Since back-to-back sends of small data are present in the workload, the application developer really should have disabled the Nagle algorithm by issuing the TCP_NODELAY SetSockOpt.

Over the last few years, we've encountered many cases where a new application is developed and deployed, and is exposed to the Nagle/Delay Ack stall. The stall occurs because this new application employs back-to-back socket sends and does not disable the Nagle algorithm. The solution is to relax the Nagle algorithm just enough to avoid the traffic stalls, while still preserving the algorithm's value (which is bandwidth conservation). In z/OS V1R11, the Nagle algorithm now allows exactly two small packets on a connection to be outstanding. Allowing the second packet to flow results in the remote node (which uses a delayed Acknowledgement scheme) to immediately generate an ACK once the second packet arrives. This avoids the 200 millisecond stalls. Performance testing in the z/OS Communications Server lab has demonstrated the dramatic effect of having removed Nagle stalls. The transaction rate jumped from 3 transactions per second to 2650 transactions per second in preliminary testing.



Both the TN3270 server and SNA/EE provides separate ECSA relief line items in this releases.

The single most important ECSA relief comes from moving the socket control blocks out of ECSA and into 64-bit common storage.

Each socket control block occupies 384 bytes – a system with just 2730 sockets occupies roughly one MB of ECSA for this purpose.

64-bit Common storage resides on a 2GB boundary, and the size is a 2GB multiple

Minimum size is 2GB

Maximum size is 1TB

Default size is 64GB (the default unless your installation changes it through an IEASYSxx PARMLIB option).

The 64-bit Common storage size can be specified by way of the HVCOMMON keyword in IEASYSxx, or system parameter. HVCOMMON=10G (in this example the size of the 64-bit common area is 10GB)

The 64-Bit Common Area resides on a two gigabyte boundary, and the total size must be at least two gigabytes and the maximum size is one terabyte. This 64-Bit common storage is located just below the "Shared Area".

You can monitor the use of 64-bit shared memory object with RMF.

Monitoring use of 64-bit memory objects

- You can monitor the system's use of 64-bit memory objects through the RMF monitor III STORM (option 7A) report:

```
RMF V1R11  Storage Memory Objects          Line 1 of 8
Command ===>                                             Scroll ===> CSR

Samples: 60      System: 3090  Date: 06/12/09  Time: 10.01.00  Range: 60    Sec

------------------------------- System Summary -------------------------------
-- Memory Objects --   --------- Frames ---------    --- Area Used % ----
Common Shared  Large   Common  Fixed Shared   1 MB    Common Shared   1 MB
     6      0             800      0      0              0.0    0.0

------------------------------------------------------------------------------
           Service          ---- Memory Objects ---  Frames  ----- Bytes -----
Jobname  C Class      ASID  Total  Comm   Shr Large   1 MB   Total  Comm   Shr

SMSPDSE  S SYSTEM     0008     12     0     0                76.0M     0     0
TRACE    S SYSTEM     0004      8     0     0                8192K     0     0
GRS      S SYSTEM     0007      4     0     0                 140G     0     0
ABCRESO  S SYSSTC     0040      4     0     0                4096K     0     0
JESEAUX  S SYSSTC     0027      3     3     0                3072K 3072K     0
ZFS      S SYSSTC     0049      2     0     0                22.0M     0     0
*MASTER* S SYSTEM     0001      1     1     0                1024K 1024K     0
TCPCS    S SYSSTC     0058      1     1     0                1024K 1024K     0
```

System resolver uses 64-bit private memory objects for name server cache

TCP/IP uses 64-bit common memory objects for SCBs

You can also monitor TCP/IP's use with the D TCPIP,,STOR command

```
10.22.08  d tcpip,tcpcs,stor
10.22.09  EZZ8453I TCPIP STORAGE
EZZ8454I TCPCS    STORAGE           CURRENT   MAXIMUM     LIMIT
EZZ8455I TCPCS    ECSA                9645K    10074K   NOLIMIT
EZZ8455I TCPCS    POOL               13949K    14047K   NOLIMIT
EZZ8455I TCPCS    64-BIT COMMON          1M        1M   NOLIMIT
EZZ8459I DISPLAY TCPIP STOR COMPLETED SUCCESSFULLY
```


In this example, the system resolver (address space name ABCRESO) uses 4MB of 64-memory, but not common memory (in the application region part of the 64-bit addressing space). This memory is in z/OS V1R11 used for name server caching by the system resolver.

The TCP/IP address space (address space name TCPCS) in this example uses a single 1MB common 64-bit memory object. This is for the SCB control blocks. If TCP/IP needs space for more than roughly 2730 SCBs, it obtains another 1MB common memory object.

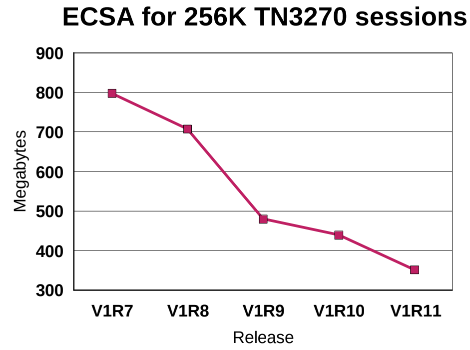
This example uses an RMF monitor III report.

The D,,TCPIP,STOR command has also been enhanced to show how much 64-bit common TCP/IP uses at any point in time.

You can also see a reduction in TCP/IP's use of ECSA storage corresponding to the amount it now uses in 64-bit common.

TN3270 server ECSA usage improvement through z/OS V1R11 Communications Server

Release	ECSA for 256K TN3270 sessions
V1R7	798M
V1R8	708M
V1R9	480M
V1R10	440M
V1R11 (1)	352M



The numbers are configuration dependent, but they should give you an idea of the magnitude of the savings achieved in the recent releases.

Note (1): The V1R11 number is a preliminary number - it may change before general availability of z/OS V1R11 Communications Server

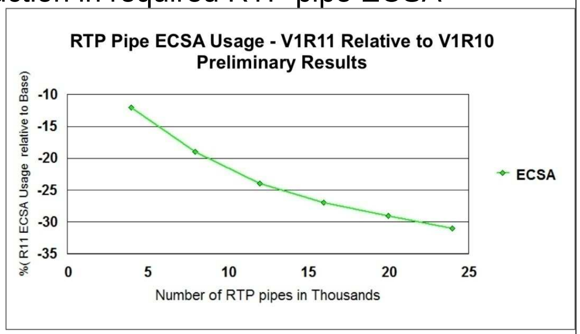
The TN3270 server has significantly lowered the amount of extended common storage it needs to support large numbers of TN3270 clients.

This has been a focus area through the last four releases, and the preliminary z/OS V1R11 numbers indicate that ECSA usage in R11 is around 44% of what it was in z/OS V1R7.

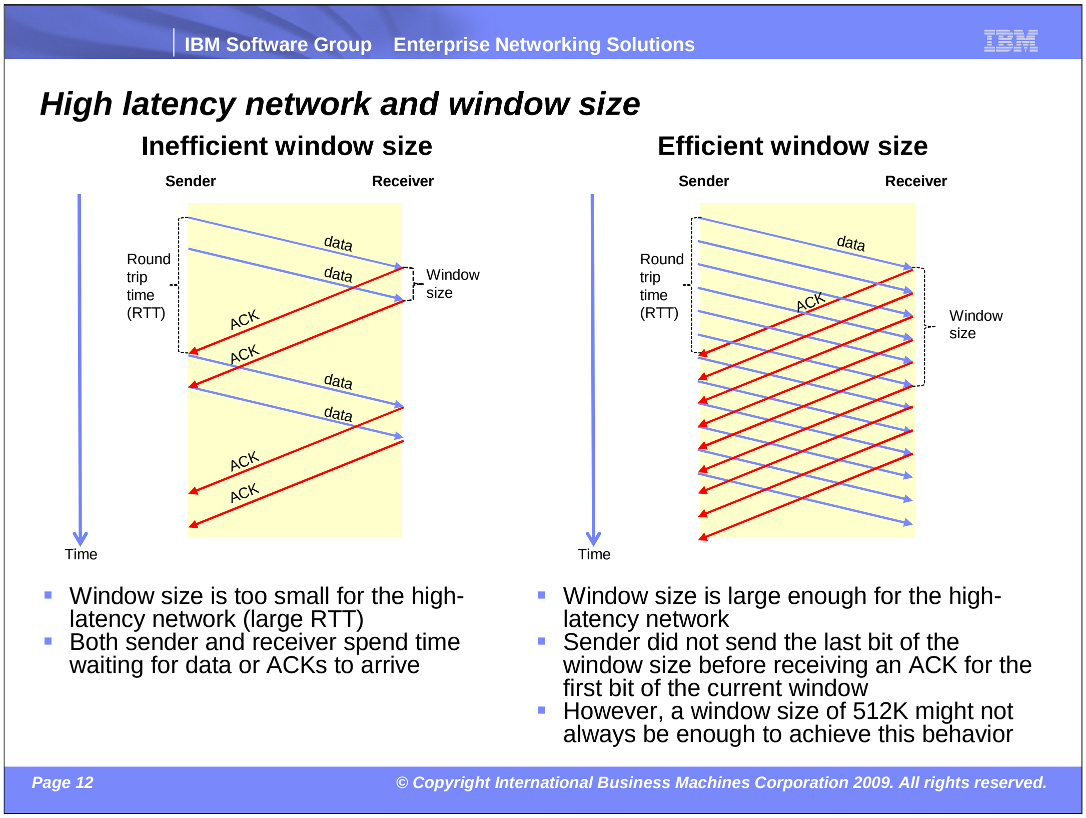
Reduce CSA requirements for rapid transport protocol pipes

- Before V1R11, each RTP pipe is represented by a control block in ECSA
- In V1R11, a large portion of the RTP control block was moved to an extension control block in VTAM® private storage. This resulted in a significant ECSA savings for installations with a large number of RTP pipes
- Preliminary estimates of the reduction in required RTP pipe ECSA storage for various RTP counts:

RTP pipes	ECSA reduction
4000	11.5%
12000	23.5%
20000	29.5%



Before V1R11, each rapid transport protocol (RTP) control block used a block of ECSA of over two kilobytes. V1R11 identified all of the RTP's fields which were required to stay in ECSA, and moved the remaining fields to an extension control block in VTAM private storage (and anchored out of the RTP control block). This reduces the ECSA footprint of the RTP control block to less than one kilobyte. The table and graph on the chart show approximate ECSA savings for various RTP pipe counts. These numbers should be considered preliminary estimates only until the final performance report is available sometime after V1R11 general availability.



Bandwidth-delay product is defined as the product of the bandwidth (or data link capacity) in bits/sec and the end-to-end latency (or round trip time) in seconds. The TCP window size is the amount of data that a sender is allowed to transmit over a TCP connection before it receives an acknowledgement for the first piece of that data. The TCP window size is affected by both the congestion window and the advertised window. The congestion window is flow control imposed by the sender (for example, TCP slow start). The advertised window is flow control imposed by the receiver and indicates how much data the receiver is willing to accept.

In an ideal situation, the TCP window size for a TCP connection equals the bandwidth-delay product and therefore optimizes throughput by keeping the data pipe full.

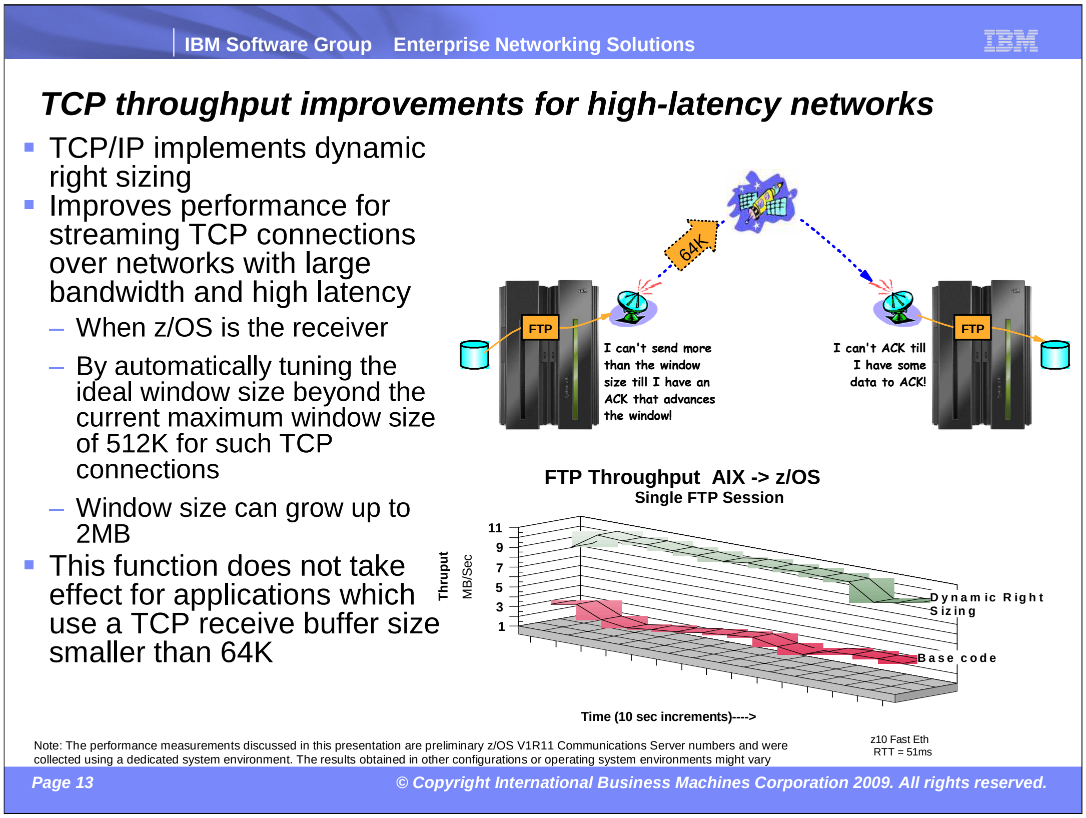
The receiver advertises a TCP window size which can be no larger than the receive buffer size (which the z/OS stack currently caps at 512K). The sender sets its window size as the minimum of three factors: (1) the congestion window, (2), the advertised receive window, and (3) the send buffer size.

The best known way to address this is to increase the window size. The window size maximum is currently 512K on z/OS. On very high bandwidth-delay product networks, such as satellite links, a window size of 512 K is not sufficient to efficiently use the network capacity

The blue arrows represent the client sending data packets to the server. The red arrows represent the server sending ACKs for the data. The z/OS stack typically sends an ACK for every other packet, but the basic concept illustrated here still applies.

In this example, the round trip time (RTT) is relatively long and in the left scenario the window size is relatively small. Therefore, the client fills the window before it receives an ACK for the data at the start of the window. This forces the client to delay sending additional data until it receives an ACK or a window update. Over a long distance connection, this can cause transmission stalls and suboptimal performance.

On the right scenario, the window size is large enough for the client to have not yet sent the last bit of its window size before it receives the ACK of the first bit. Or in other words, the sender is able to keep the high-latency pipe full all the time.



Streaming workload over large bandwidth and high latency networks (such as satellite links) is typically constrained by the TCP window size. The problem is that it takes time to send data over such a network. At any point in time, data filling the full window size is 'in-transit' and cannot be acknowledged until it starts arriving at the receiver side. The sender can send up to the window size and then must wait for an ACK to advance the window before the next chunk can be sent.

If it were possible to dynamically adjust the window size to what it takes to fill the network in-between the sender and the receiver, higher throughput might be achieved.

This support will, on the receiver side, dynamically adjust the window size upward (beyond 180K if needed) in an attempt to 'fill' the pipe between the sender and the receiver. The goal is that as soon as the sender has sent the end of its window, the sender receives an ACK from the receiver. That ACK allows the sender to advance the window and send another chunk onto the network.

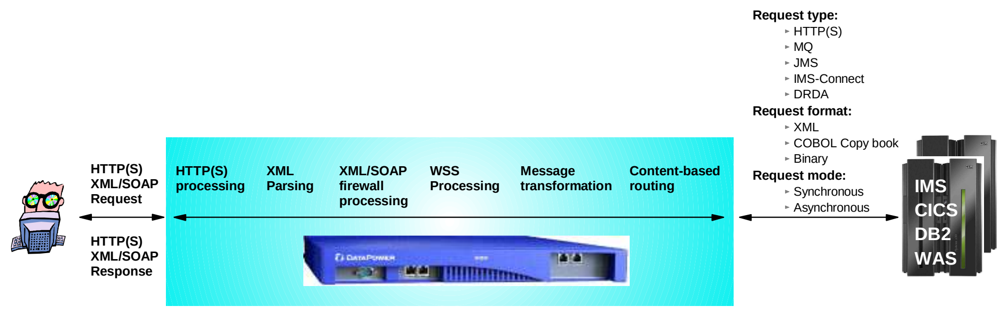
The dynamic right sizing (DRS) algorithm is based on a paper that was published by Los Alamos National Laboratory. The goal of DRS is to keep the pipe full and prevent the sender from being constrained by the advertised window.

The window size can grow as high as 2MB. The TCP/IP stack disables the function if the application doesn't keep up.

A netstat all report shows the DRS-adjusted receive buffer size. The report shows the current window size in the RcvWnd field, and the fact that DRS is in effect for the connection is indicated with the x40 bit in the TcpPrf byte.

What is DataPower?

- DataPower® can perform advanced Web services operations, contents-based routing, and transformation of requests to traditional z/OS applications
  - Security: XML/SOAP firewall capability, Web services security processing
  - XML offload: XML parsing on specialty device
  - Message transformation
    - Transform XML/SOAP to traditional z/OS application data formats
    - Interface with existing z/OS applications such as HTTP, MQ, JMS, IMS-Connect, or DB2® DRDA®
  - Contents-based routing: based on data in request (including data protected by Web services security)
    - Select proper target server, request type, format, and mode
  - Monitoring and SOA governance



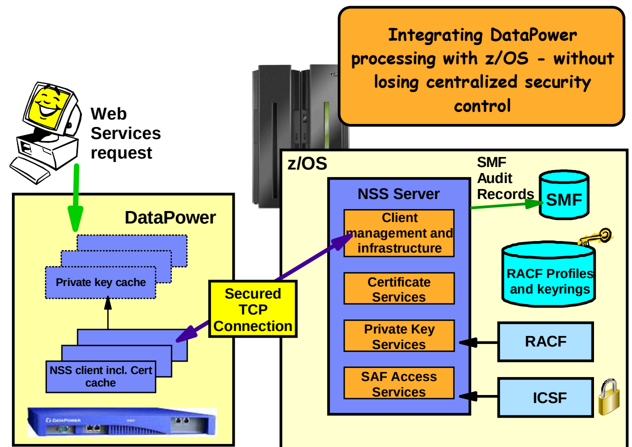
DataPower can be used to provide Web services access to other platforms, such as z/OS.

DataPower provides a full Web services protocol stack, including support for Web services security. DataPower can be customized to act as a Web services gateway to z/OS using traditional transaction interfaces to existing z/OS applications such as MQ, IMS-Connect, and DB2-DRDA. In such a setup, DataPower can provide the ability to integrate existing z/OS transactions into a Web services environment.

IBM Software Group   Enterprise Networking Solutions

DataPower integration with z/OS security - extending NSS

- The integrated infrastructure provides these basic operations:
  - Central z/OS-resident NSS services to perform operations that require access to a private key:
    - Data decryption
    - Signing of outbound XML messages (DataPower provides hash to sign)
  - Download private keys where the DataPower device is configured to perform private key operations locally
    - Supported when private keys are managed by RACF®
    - Not allowed when private keys are secured and managed by ICSF
  - Services to download and maintain local certificate cache
  - Services to provide access for DataPower to SAF-based identification, authentication, and access control functions were delivered as part of z/OS V1R10:
    - “Remote SAF checks”



Page 15

Integration of DataPower and NSS provides:

Offloaded XML and Web services security processing

With centralized z/OS-resident management of keyrings and certificates

DataPower use of SAF-based authentication and access control

z/OS V1R10 Communications Server delivered the SAF access support for DataPower.

z/OS V1R11 Communications Server extends that support to services that require access to support authorized sharing of x.509 Certificates and non-ICSF-protected RSA Private Keys between a z/OS SAF KeyRing and authorized DataPower clients.

In addition, the Network Security Services (NSS) server can perform RSA-based signature generation and RSA-based decryption operations using ICSF-protected RSA Private Keys for authorized XML appliance clients.

These new security features effectively extend the centralized certificate services available in the NSS server to XML appliances.

Trademarks, copyrights, and disclaimers