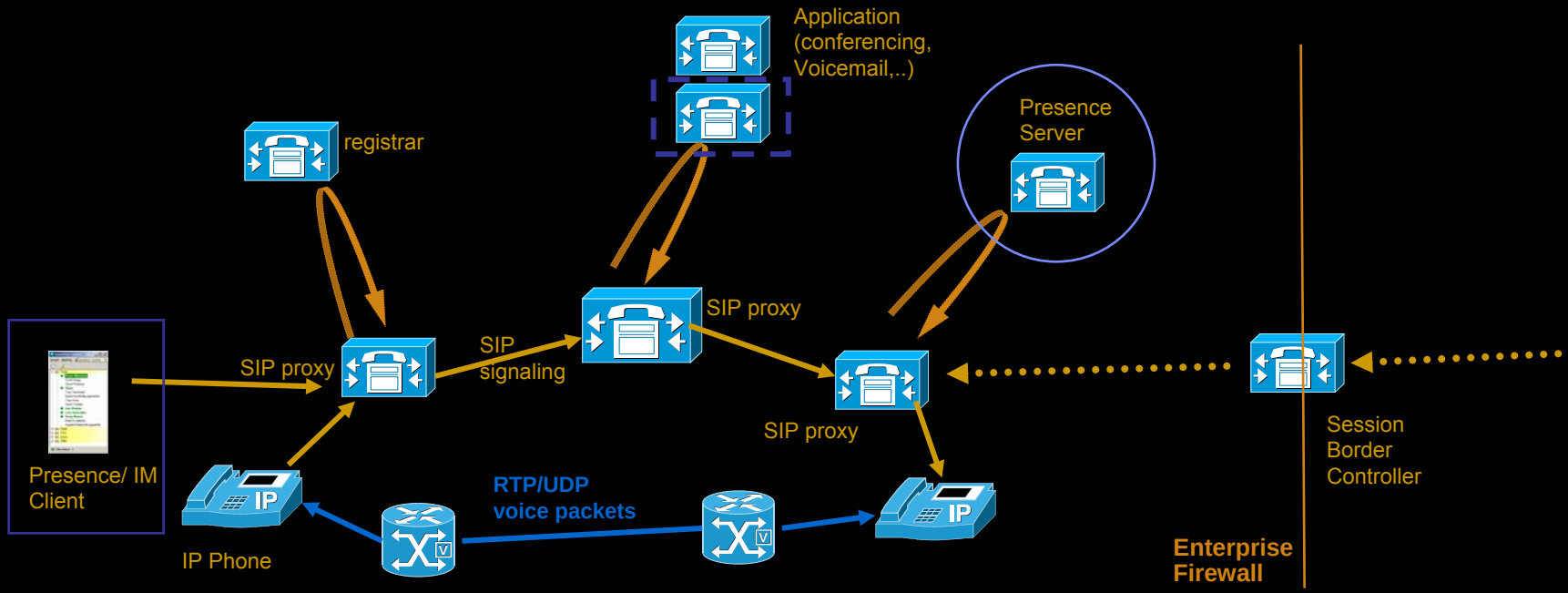
A Programmable Message Classification engine for Session Initiation Protocol

Arup Acharya Xiping Wang Charles Wright

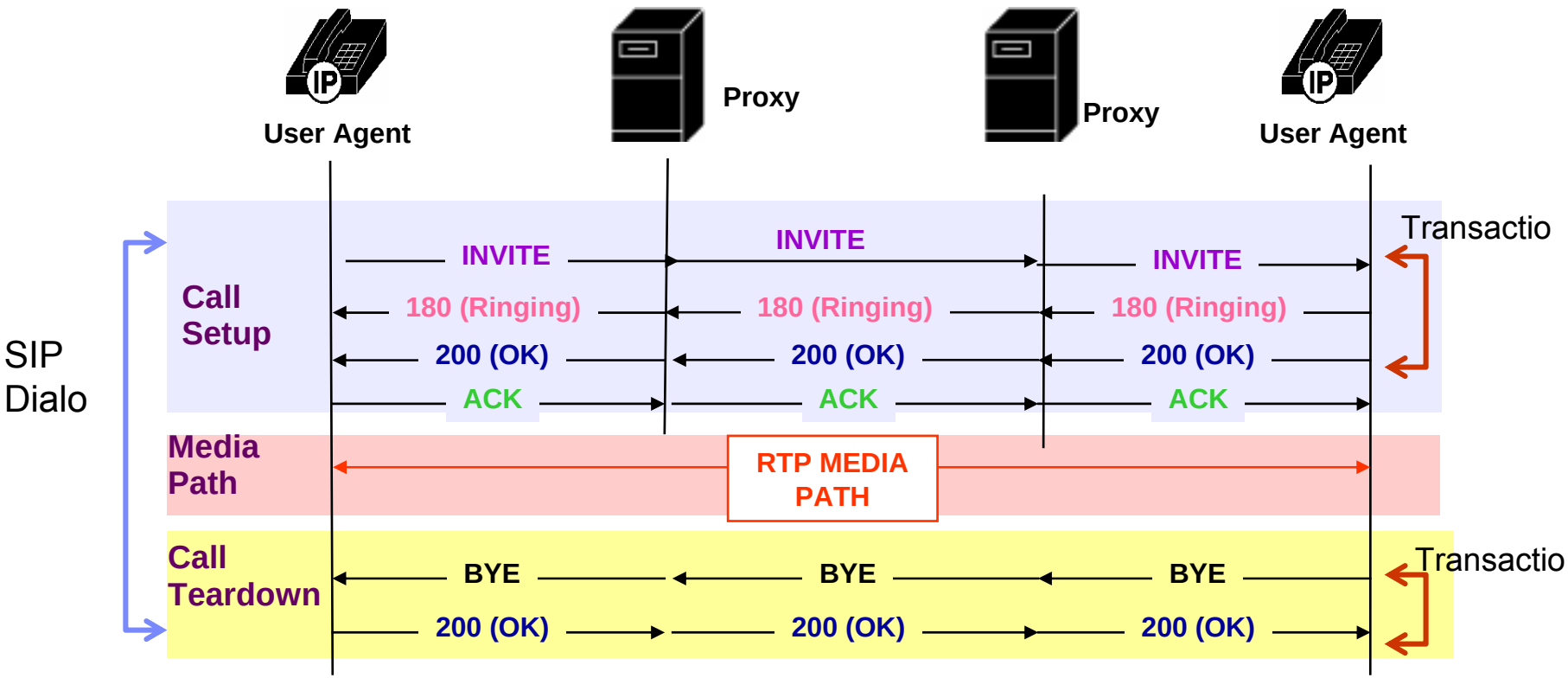
Internet Infrastructure & Computing Utilities
IBM TJ Watson Research Center

SIP and SIMPLE

- Session Initiation Protocol (SIP) - an Internet signaling protocol for setting up multimedia sessions
  - Defines a new control /signaling layer and control servers, in parallel to the media
  - It is a client to client (peer to peer) technology mediated through control servers – it is NOT a client-server protocol (unlike HTTP).
- SIMPLE : SIP extensions to support Presence and Instant Messaging
  - Publish/Subscribe mechanism
  - Instant Messages : Page mode , Session Mode
- SIP is the basis for IMS (IP Multimedia Subsystem)
- SIP has been around since 2001 – large VoIP and IM/Presence deployments already exists (SIP, XMPP, proprietary)
  - IETF came out with RFC 2543 in '99, RFC 3261 in 2002
  - SIP is the basis for IP Multimedia Subsystems (IMS) being deployed by cable, wireline and wireless operators

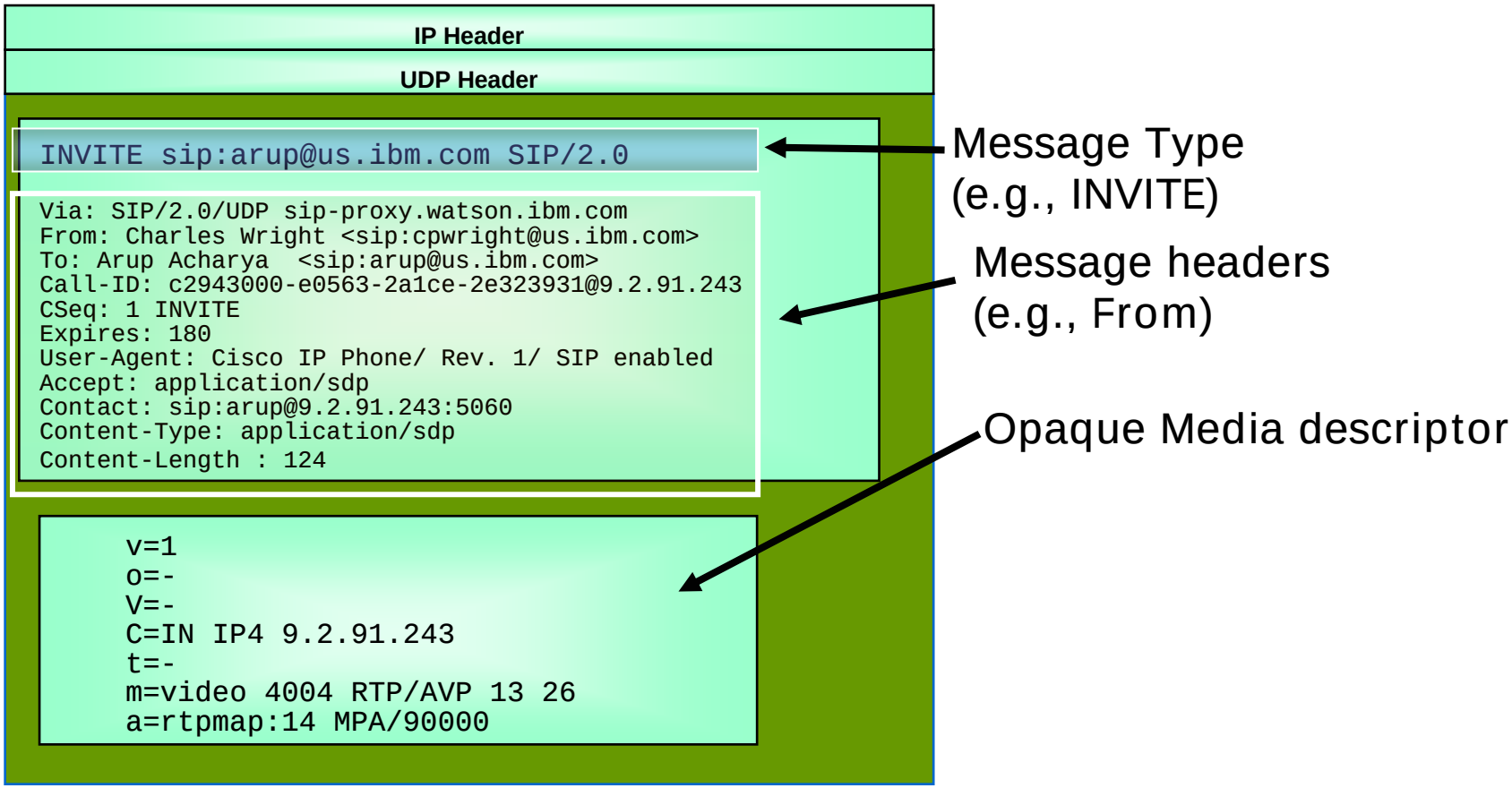


Sample SIP session : Voice-over-IP call



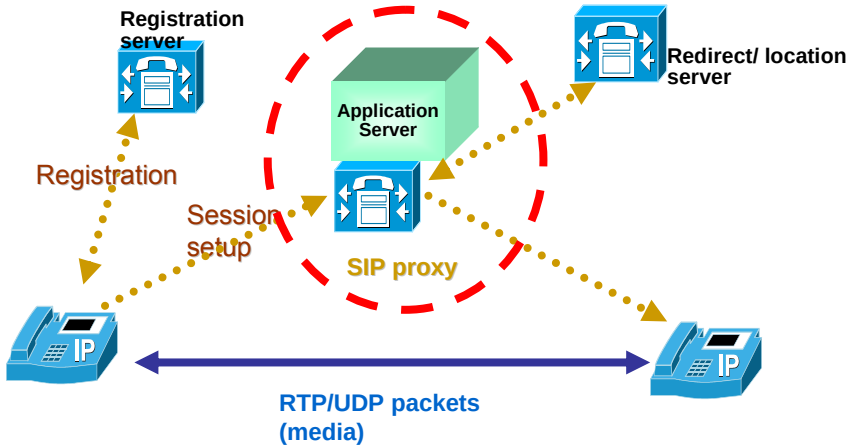
SIP message structure

- Syntactically similar, Semantically more complex than HTTP
- HTTP is client-server; SIP is client-to-client intermediated by multiple servers
  - Message may undergo transformation at each hop
- SIP is a control protocol; media is separate

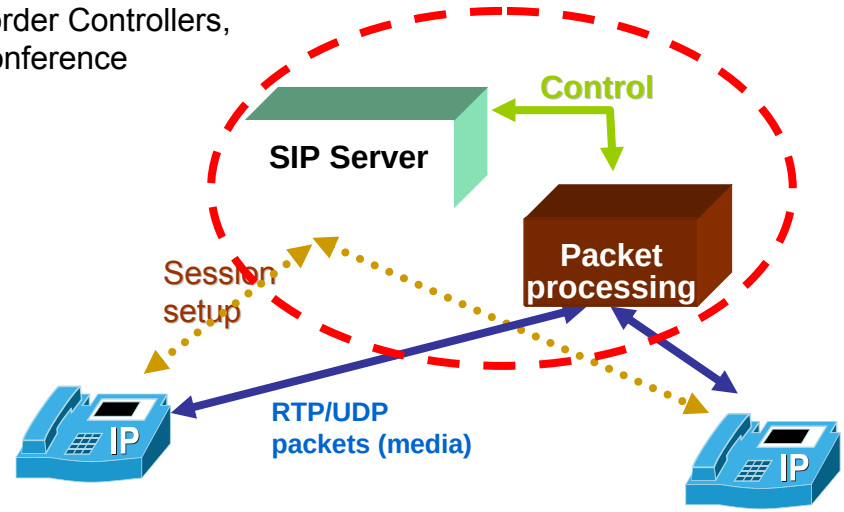


Scaling systems and software for SIP

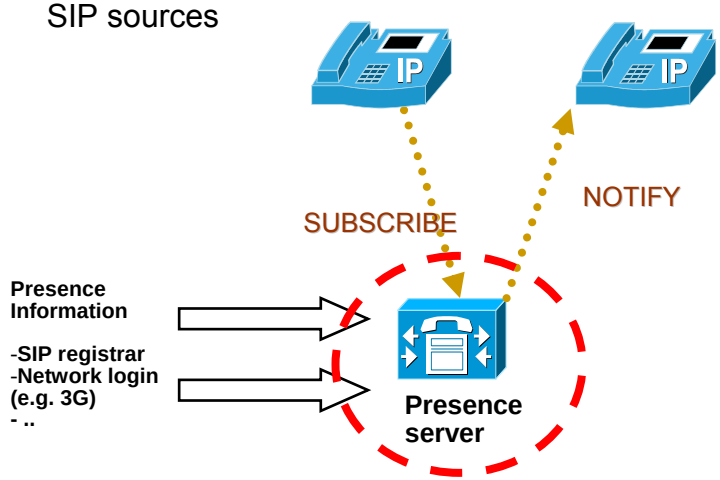
- Session Setup (voice/ video)



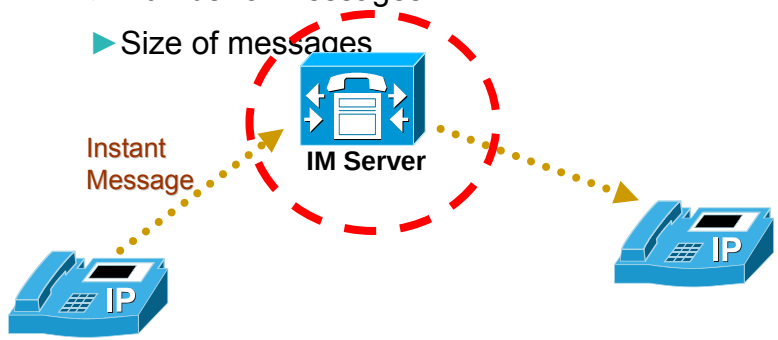
- Coupled Signaling & Media Interactions, e.g. Session Border Controllers, Conference



- Presence
  - Rate of subscription / notification requests
  - Updates in presence information, including non-SIP sources

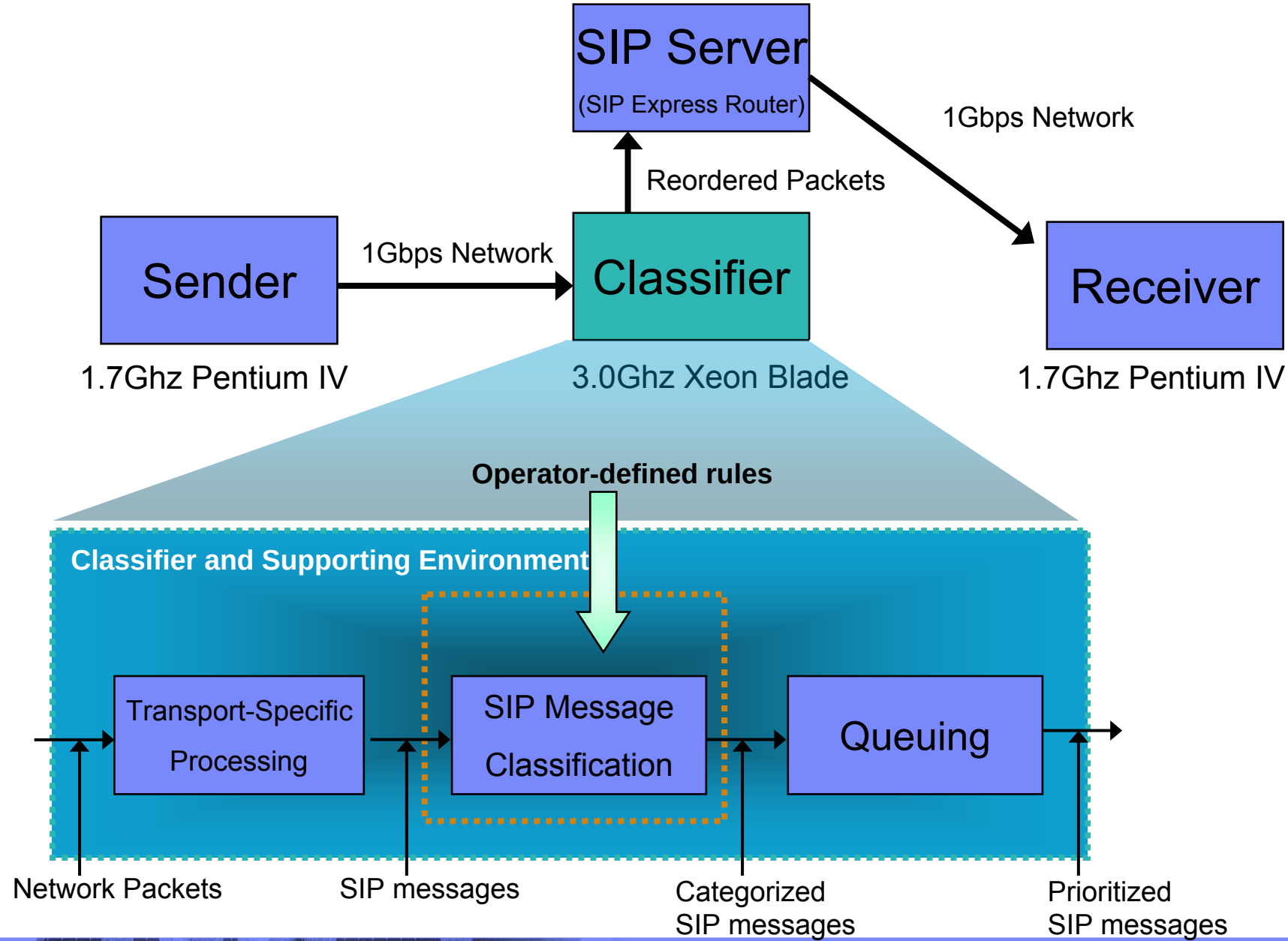


- Instant Messaging
  - Number of messages
  - Size of messages



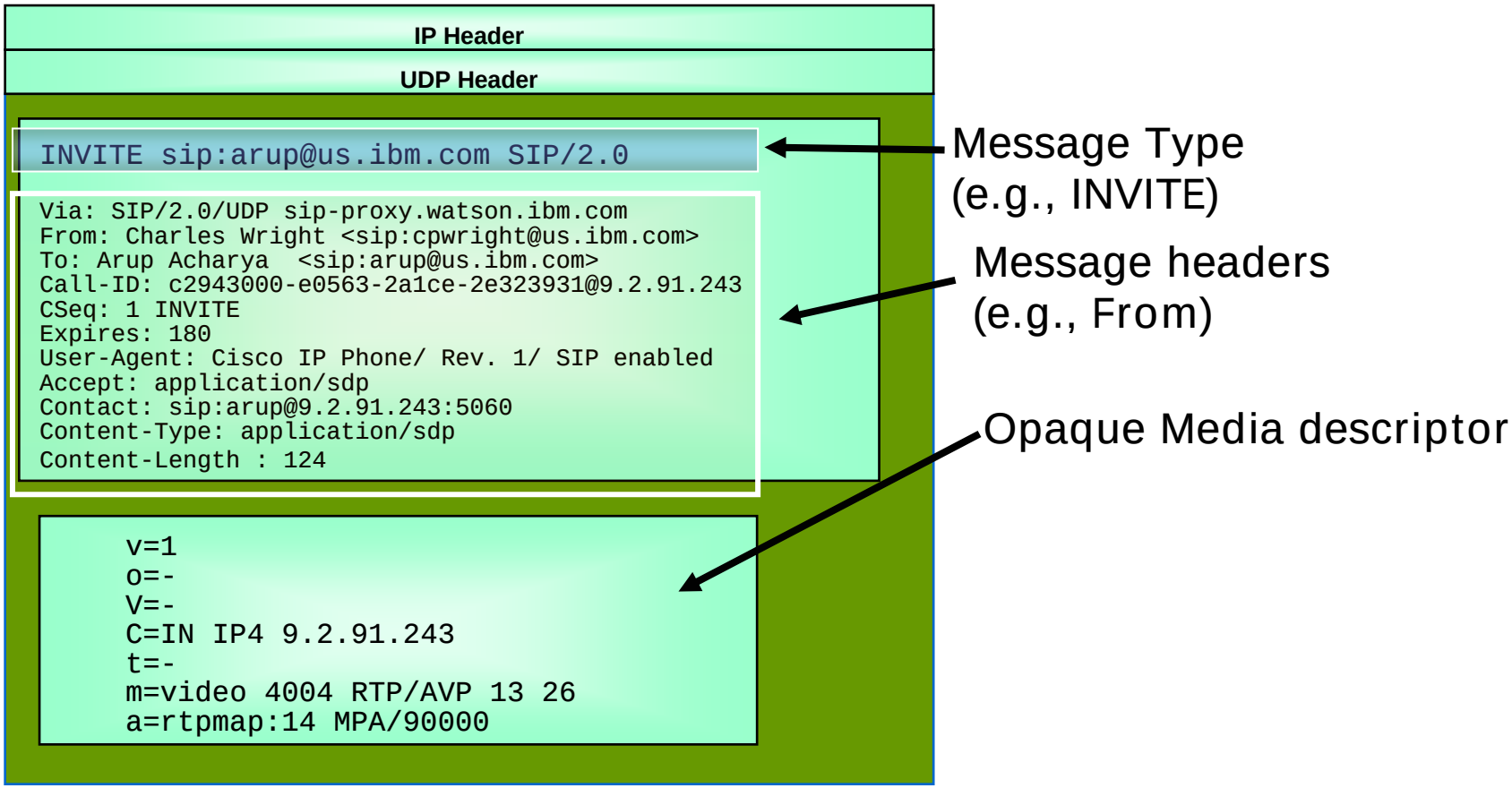
A Programmable SIP message Classification Engine

- Goal : Classify incoming SIP messages according to user-defined rules (and actions) before they are processed by a server
  - **Use-case : maximize utility / revenue, not raw throughput under server overload**
- Challenge: *fast & efficient* SIP traffic classification
- Key contributions :
  - Designed a novel ALGORITHM specifically exploiting SIP message headers
  - Classification algorithm is PROGRAMMABLE
    - Multi-purpose : Overload control, Denial-of-Service protection, Prioritization,..
  - Classification engine is STATEFUL
  - SIP Server needs no modification; can work with multiple types of SIP servers
  - In-kernel Linux implementation for EFFICIENCY


SIP Server
(SIP Express Router)
1Gbps Network
Reordered Packets
Sender
1Gbps Network
Classifier
Receiver
1.7Ghz Pentium IV
3.0Ghz Xeon Blade
1.7Ghz Pentium IV
Operator-defined rules
Classifier and Supporting Environment
Transport-Specific Processing
SIP Message Classification
Queuing
Network Packets
SIP messages
Categorized SIP messages
Prioritized SIP messages

SIP message structure

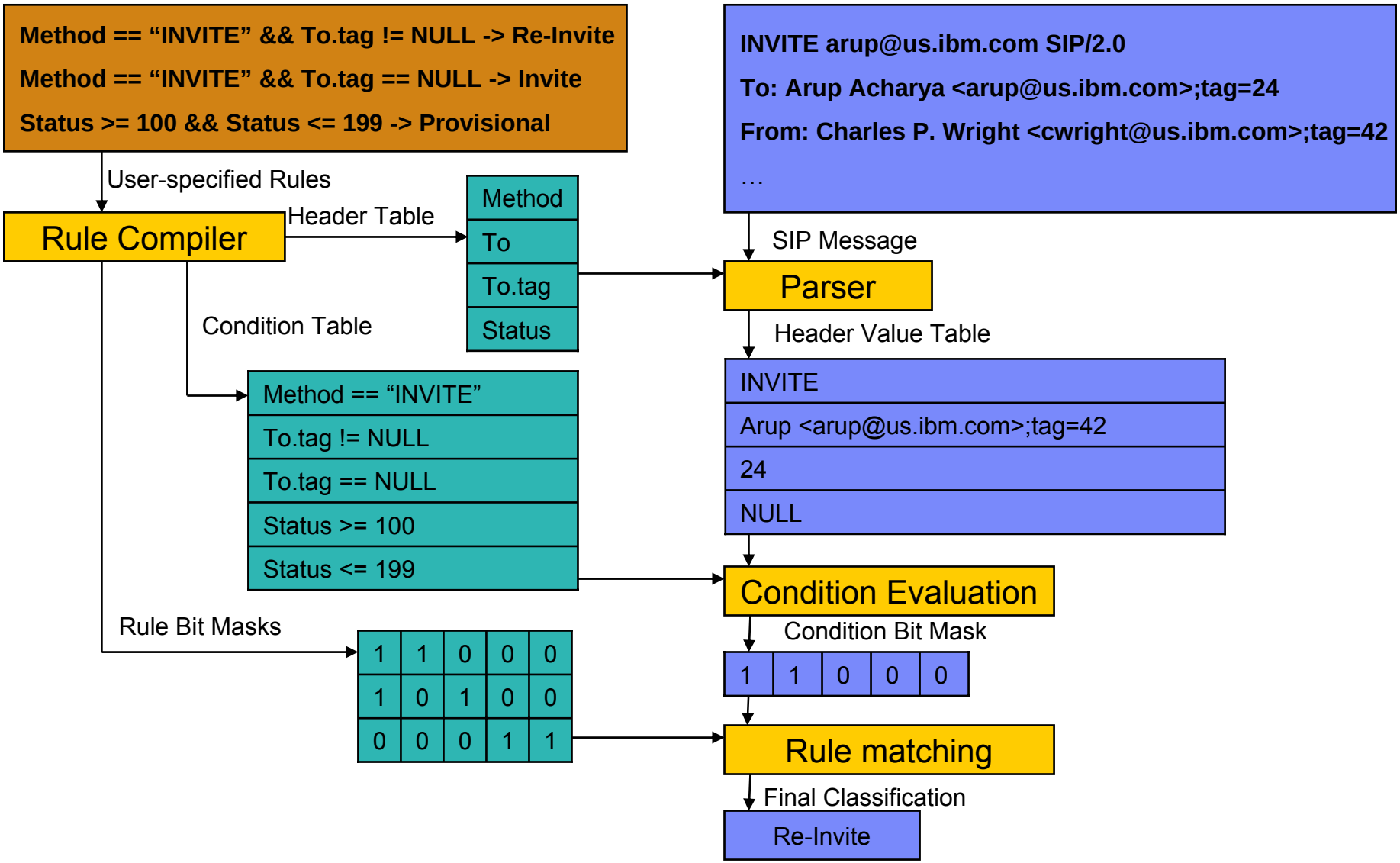
- Syntactically similar, Semantically more complex than HTTP
- HTTP is client-server; SIP is client-to-client intermediated by multiple servers
  - Message may undergo transformation at each hop
- SIP is a control protocol; media is separate



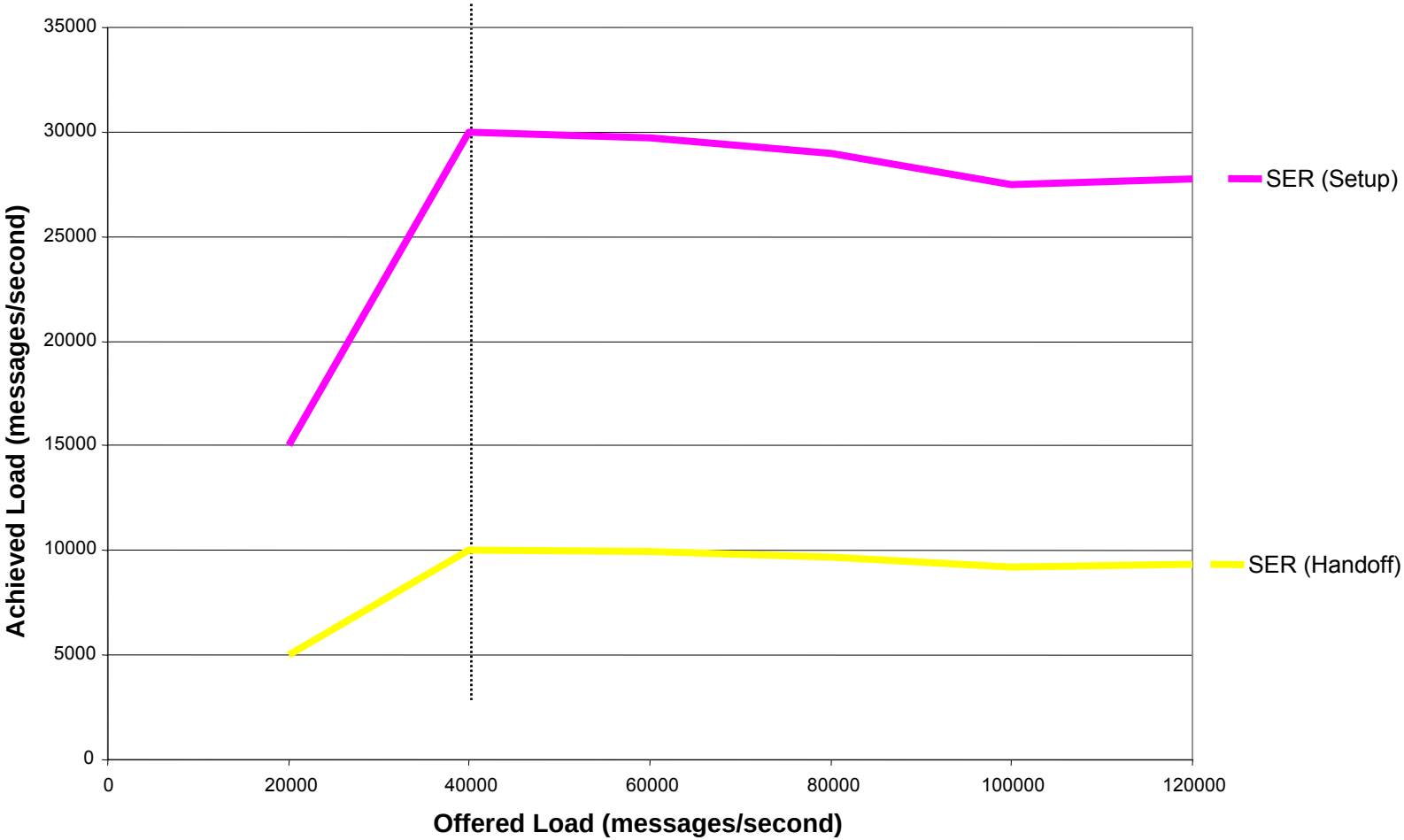
Example scenario for classification: Overload control

- Input rule set specifies call handoff messages are more valuable than call-setup (as would be the case for a mobile operator)
  - Rule conditions specify how to distinguish between handoff and setup messages
  - Rule actions prioritize handoff ahead of call setup during overload
- Classification is multi-purpose due to its programmability – Swiss army knife for IMS
  - Value-driven overload control is only one possible usage

In-Kernel SIP Classification Algorithm : overview



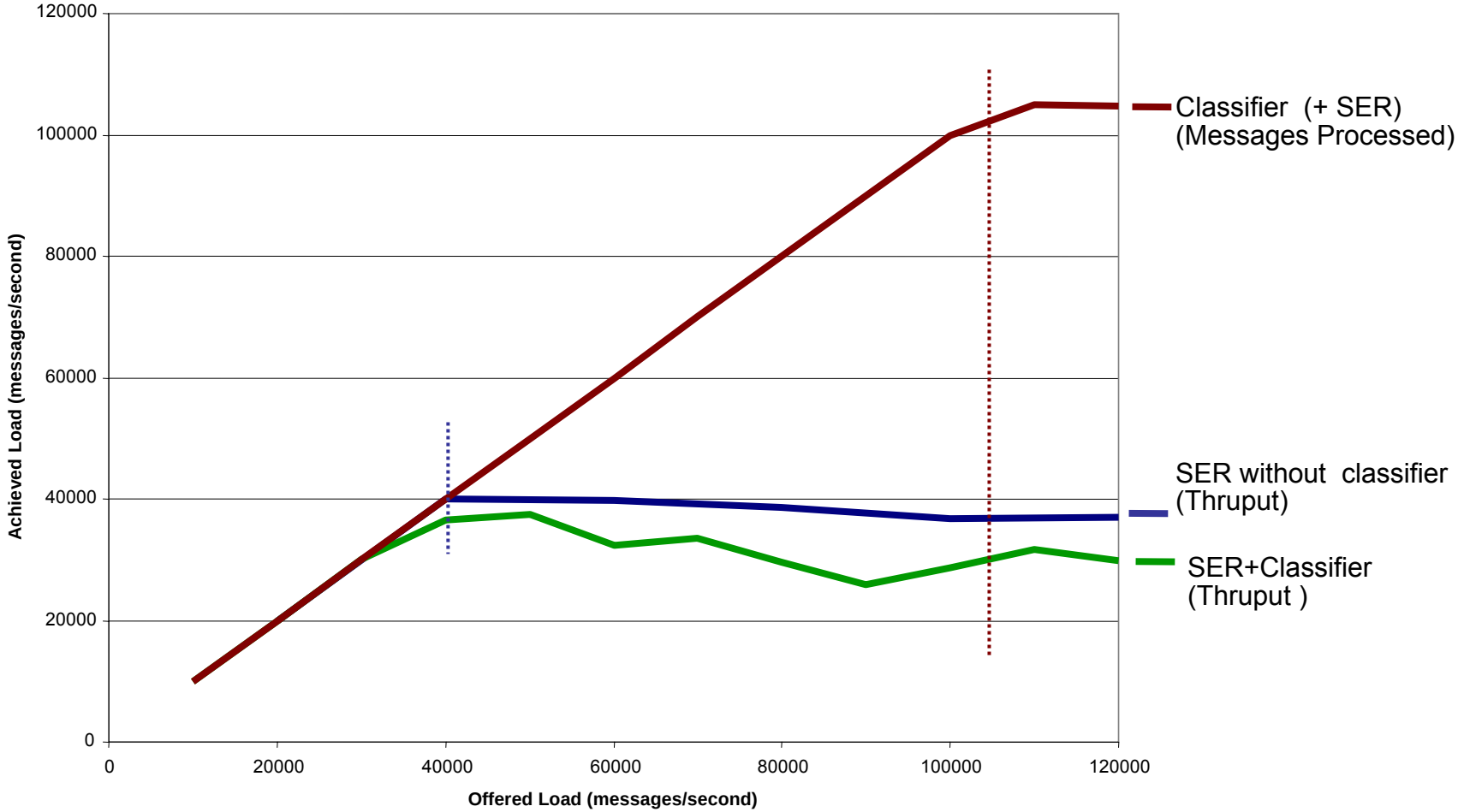
SER Performance without classifier [25% handoff]



SER's peak msg handling rate ~ 40K msgs/sec

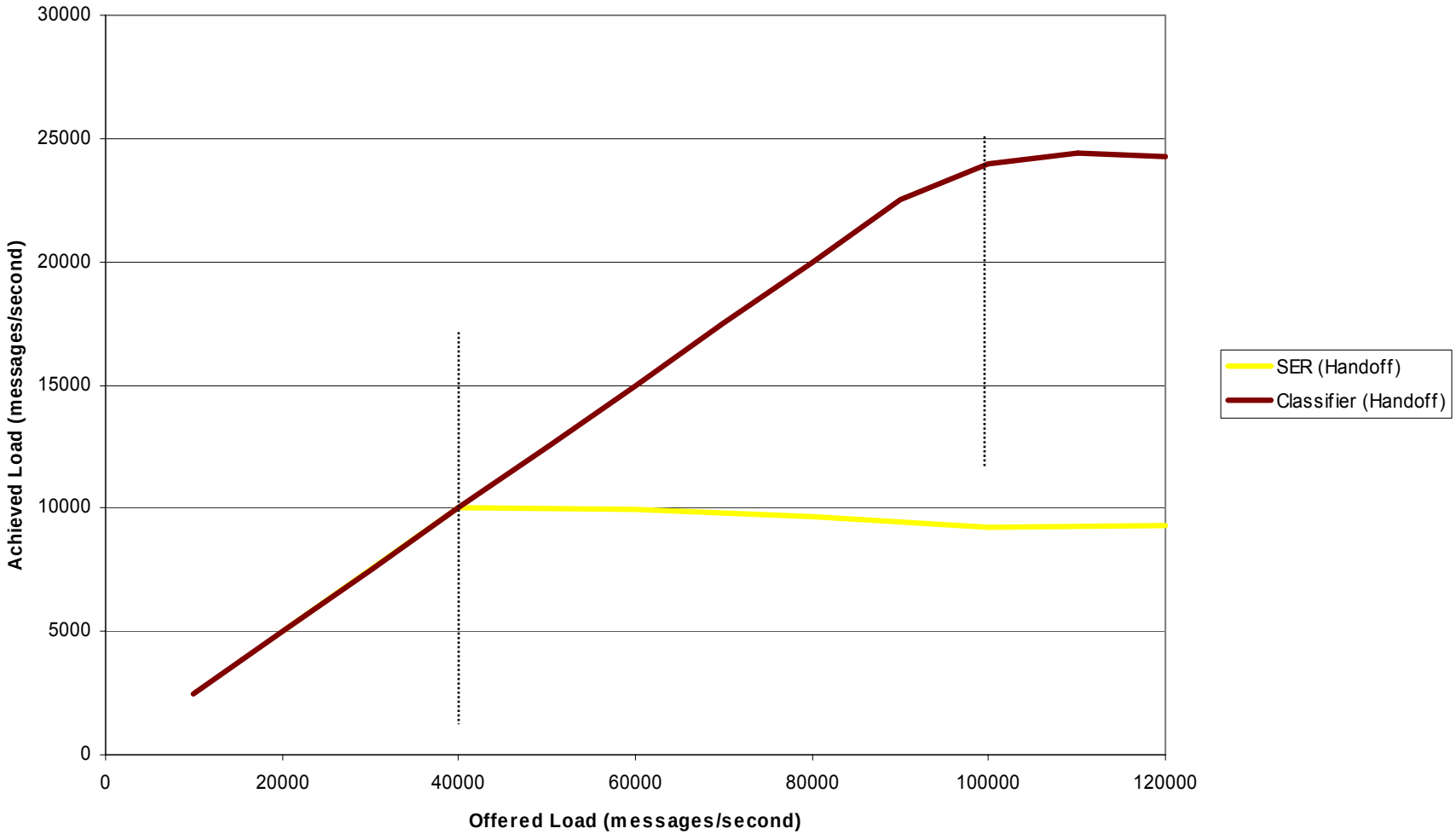
Traffic mix : 25% handoff msgs

Comparison : Thruput and quality of thruput

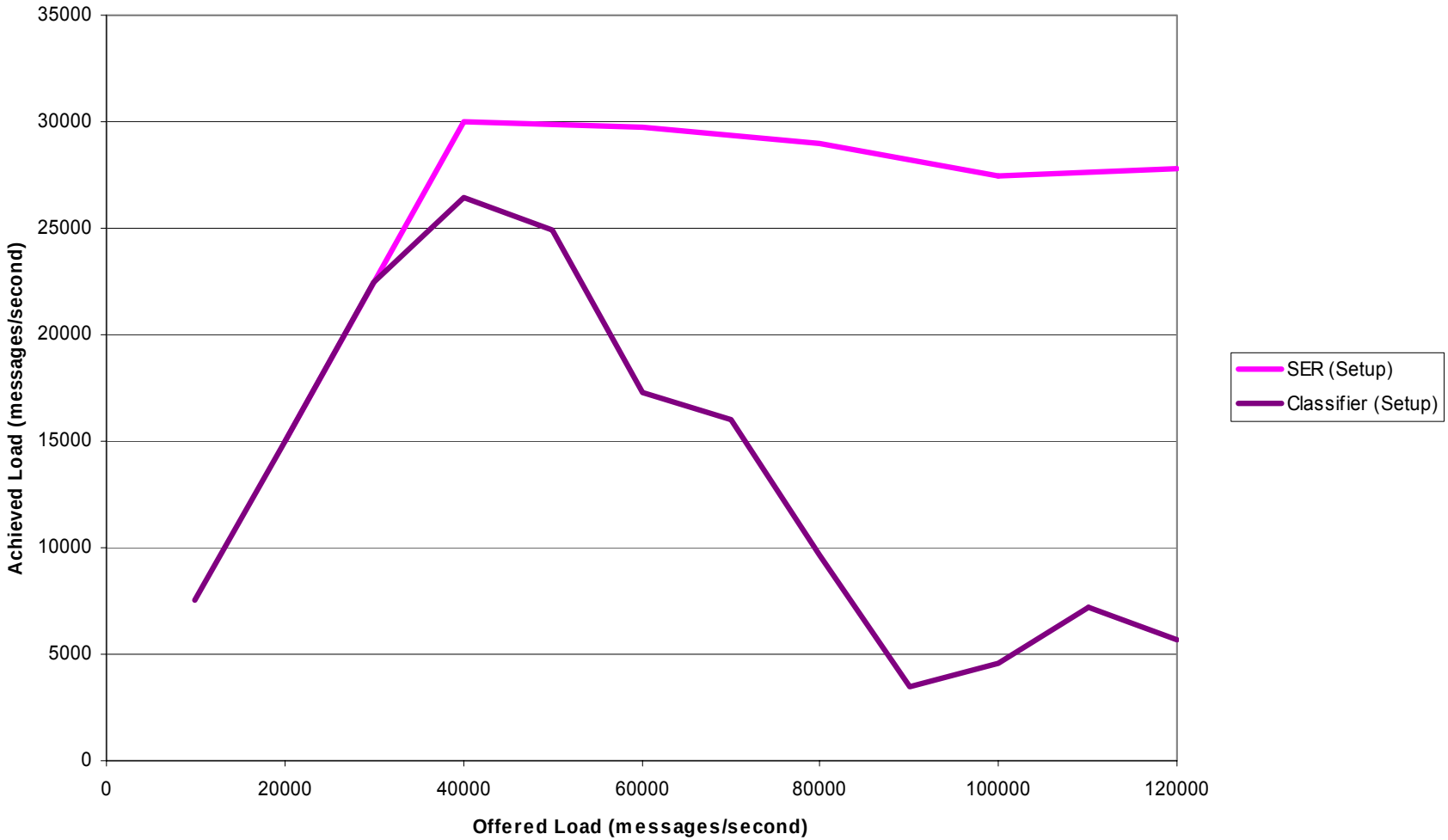


SER with classifier is able to classify about 105K msgs/sec, with some drop in peak thruput to 31K msgs/sec compared to a peak thruput (40K msgs/sec) for SER without classifier.

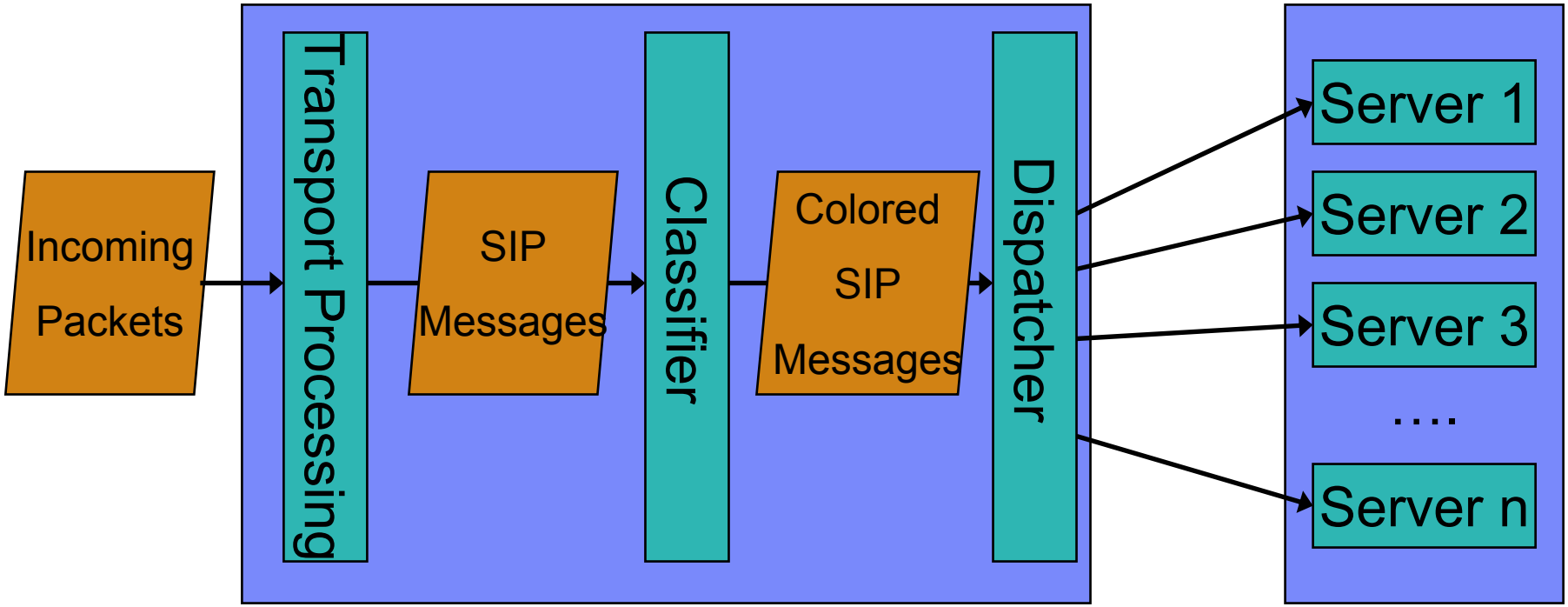
Call Handoff (High-Priority) Performance : with/out classifier



Call Setup (Low-Priority) Performance : with/out classifier

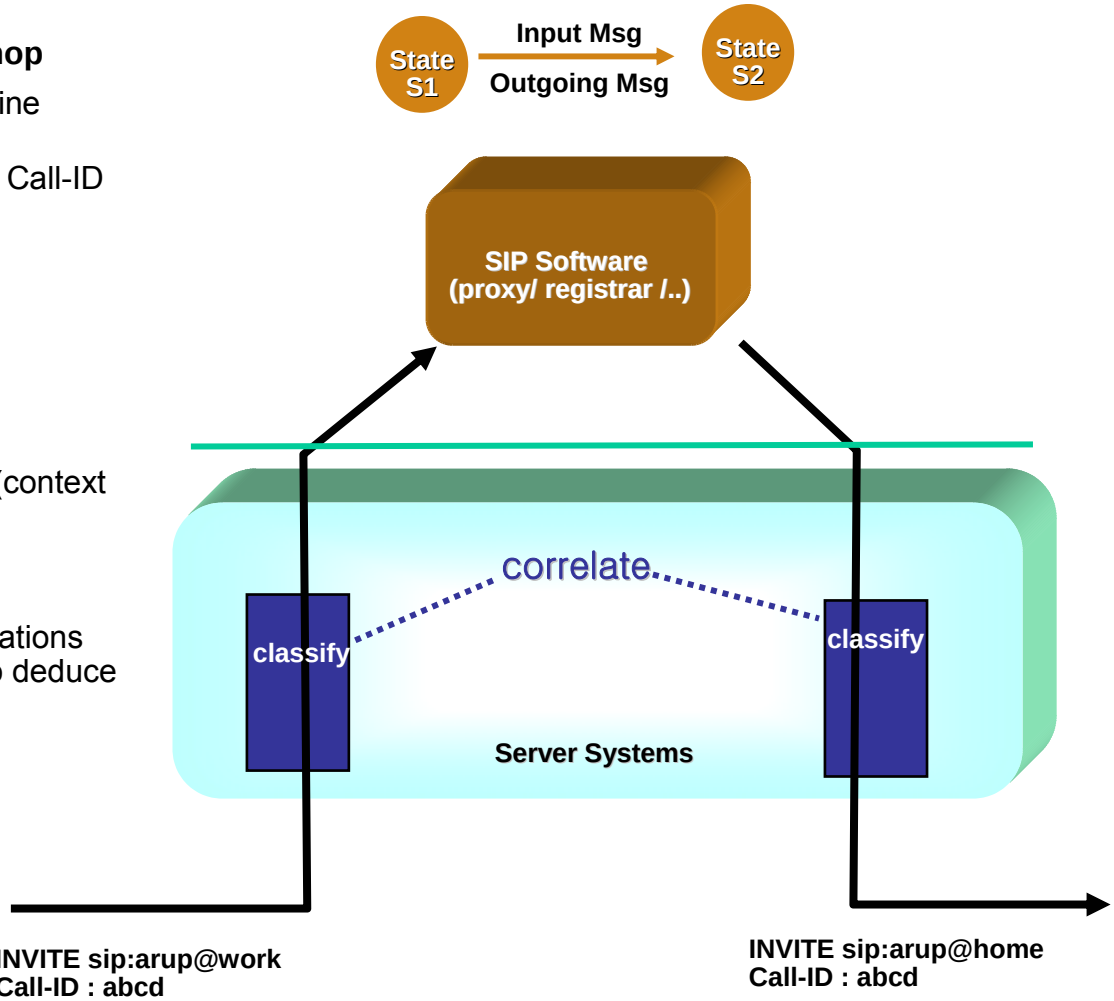


Classifier use-case : Session Dispatching



Classifier use-case : real-time monitoring of SIP call-flows

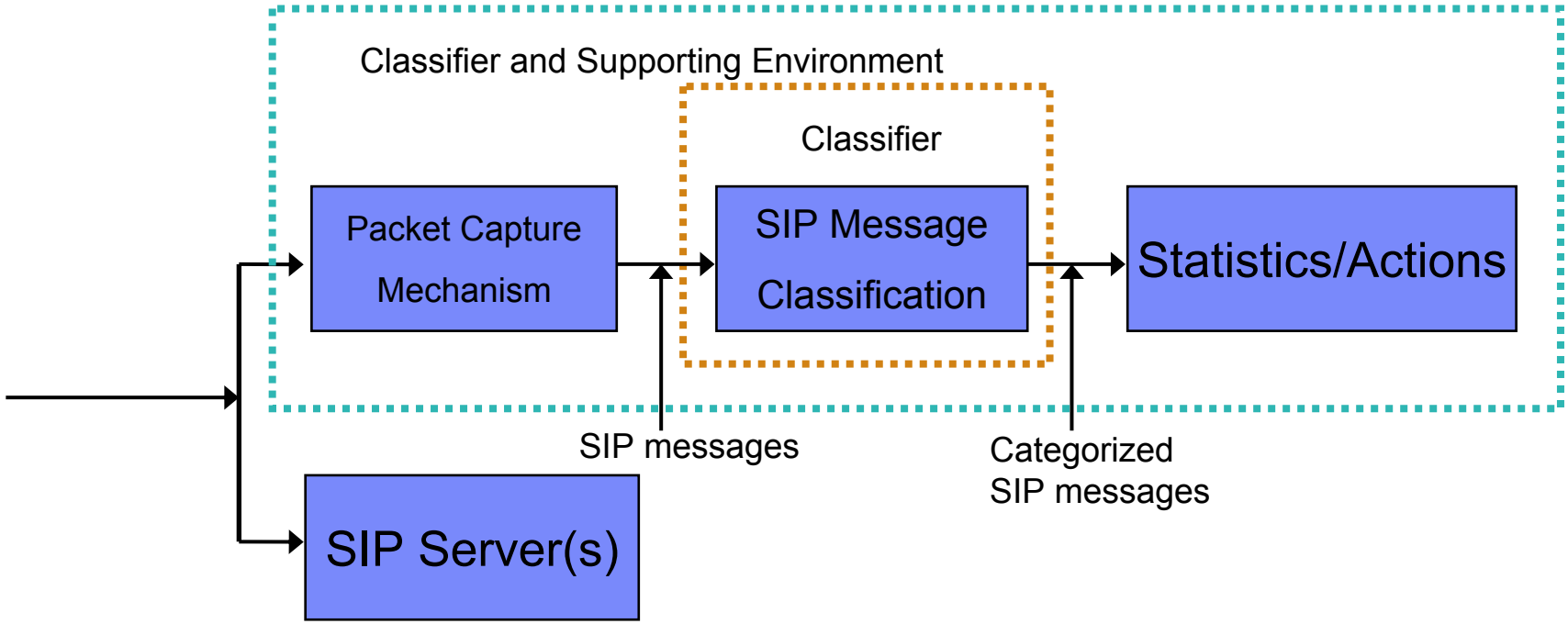
- If a SIP call fails, can we debug in real-time?
  - **SIP messages get transformed at each hop**
  - Each server runs a per-session state-machine
- Not always possible to correlate messages using Call-ID field, e.g.
  - Call-forking – each leg has same call-id but different To headers
  - When joining a conference
  - Call transfer (REFER)
- Different header subsets needed for correlation (context dependent)
  - Programmability of classifier useful
- Solution approach : trace the message transformations on each hop, trace call flow in real-time and try to deduce state-changes on the server



ACM Workshop on Mining Network Data (Mine-Net) 2007.
A. Acharya, X.Wang, C. P. Wright, N. Banerjee, K. Dasgupta

Classifier use-case : monitoring SIP servers

- *Monitoring* traffic entering a SIP server (eg CSCF in IMS)
  - Measured values can be fed back to an system-wide monitor
  - Advance warning about overload, anomalies (measure response times)
- *Regulating* traffic entering a SIP server, eg prioritization, overload control
  - Classification rules (and associated actions) can be downloaded to the classifier, in accordance with system-wide policy
- Key point : Classifier is programmable and so can be tailored for multiple scenarios



Current work in progress

- Classification engine is a tool – exploring how to program it for different scenarios
  - Integrating classification engine within DataPower
    - DataPower is an XML appliance with an embedded Linux kernel
  - SIP monitoring agent

- Collaboration with academia
  - Penn State Univ
    - Overload control : compare in-kernel vs application-level approaches
  - Georgia Tech
    - Security : VoIP DoS attack prevention, Anomaly detection

Future work

- We have studied one use-case (value-driven overload control) so far, which looked promising - huge scope to explore further
- Study how complexity of rules impact performance
- Explore stateful use-cases and impact of state maintenance on performance
- Expand design to handle SIP over TCP and SSL connections
- Better interlock between SIP Proxy / Application and Classifier
- Combine SIP header classification with message body inspection
  - Presence messages carry XML bodies (describing event that is published/subscribed)
  - Inject SIP awareness into XML appliances
- If you have ideas, we will be happy to collaborate with you

Thanks for your time!

Questions?

Contact :
Arup Acharya
arup@us.ibm.com

Details of this project and other project can be found at http://research.ibm.com/people/a/arup/

Possible instantiations

- Linux kernel module co-located with server machine
- Front-end box with classifier
- FPGA / Network-processor based implementation of classification engine
  - Server Network interface card
  - Additional hw/sw engine for Datapower
- Programmability provided by external rule-sets (downloaded to classification engine)

HTV Structure

- **All values in the classifier are maintained as header table value structures (HTVs)**
  - Primitive HTV Types: String (e.g., “Method” or “From”), Integer (e.g., “Status”), String List (e.g., “Via”), Null (e.g., a header that was not found)
  - Complex HTV Types: Tuple, Pointer, Structure, Array
    - Tuple: An ordered list of *N* HTVs (where N is fixed)
    - Pointer: A pointer to another HTV
    - Structure: User-Space Type with named members, that is translated into a Tuple at rule compilation time
    - Array: An associative array of elements. Given a key HTV returns a value HTV.

Associative Arrays

- Enable the classifier to maintain complex state for each dialog, transaction, etc.
- Maps a key to a value
- Every array stores tuples, the first element is the key, the remaining elements are the value
  - The key and value may also be tuples
- Implemented using linear hashing [Larson 1988]
  - Dynamically sizes hash table, without the need to rehash every bucket when the load factor is exceeded

Actions

- **Actions are defined using a simple three-address code**
  - Operator
    - Arithmetic: ADD, MULTIPLY, SUBTRACT, MOD, DIVIDE
    - Arrays: INSERT, REMOVE (find is done of belongs-to operator in rules)
    - Tuples: TUPALLOC, TUPEXTRACT, TUPASSIGN
    - Other: TIME, COLOR (for overload control)
  - Operands are all HTVs
    - May be immediate, a Header (from the Header Value Table) or a Variable
  - Next pointer
- **Rule compiler currently provides named subroutines**
- **Future work: provide simpler C-like syntax in rule-compiler rather than assembler-like syntax**

Parsing SIP Messages

- Each header or pseudo-header has a priority (lower numbers are executed first)
- Negative numbers are used for the SIP request/response line, which validates the message is in fact SIP
- Standard SIP headers have a priority of “zero” and are concurrently located in the message using a series of C switch statements
  - Experimentally determined that simple switch was more efficient than hashing the header and performing a lookup or multi-pattern matching structures like SBOM
  - Intuition: enables us to match 4 characters at a time, with a series of simple comparisons
  - If we are not matching on set-valued headers, can terminate the search after all of the headers are found
- Sub-headers and tuples have a positive priority, and are derived from the standard headers using simple functions

Session Dispatching

Struct Session = {String ID, Int Server Int Expire}

Global Session: %ActiveSessions

Local Session: $NewSession, *CurrentSession

Local Int: $MyServer

Global Int: $CurrentServer, $nServers

Init -> $CurrentServer = 0, $nServers = 3, ExpiryThread(%ActiveSessions, Expire)

10:*CurrentSession = Call-ID belongs-to %ActiveSessions -> *CurrentSession.Expire = Now() + 900, Color *CurrentSession.Server

20: NOT Call-ID belongs-to %ActiveSessions -> $MyServer = $CurrentServer++ % $nServers, $NewSession = (Call-ID, $MyServer, Now() + 900), Insert(%ActiveSessions, $NewSession), Color $MyServer

30: Response >= 200 && CSeq.Method == “BYE” && *CurrentSession = Call-ID belongs-to %ActiveSessions -> Remove(%ActiveSessions,*CurrentSession), Color *CurrentSession->Server

Session Dispatching

Struct Session = {String ID, Int Server Int Expire}

Global Session: %ActiveSessions

Local Session: $NewSession, *CurrentSession

Local Int: $MyServer

Global Int: $CurrentServer, $nServers

Initialization: The number of servers is configured (0 through 2), and we dispatch the first request to server 0. A thread to expire sessions is created.

Init -> $CurrentServer = 0, $nServers = *3*, ExpiryThread(%ActiveSessions, Expire)

10:*CurrentSession = Call-ID belongs-to %ActiveSessions -> *CurrentSession.Expire = Now() + 900, Color *CurrentSession.Server

20: NOT Call-ID belongs-to %ActiveSessions -> $MyServer = $CurrentServer++ % $nServers, $NewSession = (Call-ID, $MyServer, Now() + 900), Insert(%ActiveSessions, $NewSession), Color $MyServer

30: Response >= 200 && CSeq.Method == “BYE” && *CurrentSession = Call-ID belongs-to %ActiveSessions -> Remove(%ActiveSessions,*CurrentSession), Color *CurrentSession->Server

Session Dispatching

Struct Session = {String ID, Int Server Int Expire}

Global Session: %ActiveSessions

Local Session: $NewSession, *CurrentSession

Local Int: $MyServer

Global Int: $CurrentServer, $nServers

If a packet matches an existing session, then use the existing server. Update the expiration timer to 15 minutes in the future.

Init -> $CurrentServer = 0, $nServers = *3*, ExpiryThread(%ActiveSessions, Expire)

10:*CurrentSession = Call-ID belongs-to %ActiveSessions -> *CurrentSession.Expire = Now() + 900, Color *CurrentSession.Server

20: NOT Call-ID belongs-to %ActiveSessions -> $MyServer = $CurrentServer++ % $nServers, $NewSession = (Call-ID, $MyServer, Now() + 900), Insert(%ActiveSessions, $NewSession), Color $MyServer

30: Response >= 200 && CSeq.Method == “BYE” && *CurrentSession = Call-ID belongs-to %ActiveSessions -> Remove(%ActiveSessions,*CurrentSession), Color *CurrentSession->Server

Session Dispatching

Struct Session = {String ID, Int Server Int Expire}

Global Session: %ActiveSessions

Local Session: $NewSession, *CurrentSession

Local Int: $MyServer

Global Int: $CurrentServer, $nServers

If a packet matches an existing session, then use the existing server. Update the expiration timer to 15 minutes in the future.

Init -> $CurrentServer = 0, $nServers = *3*, ExpiryThread(%ActiveSessions, Expire)

10:*CurrentSession = Call-ID belongs-to %ActiveSessions -> *CurrentSession.Expire = Now() + 900, Color *CurrentSession.Server

20: NOT Call-ID belongs-to %ActiveSessions -> $MyServer = $CurrentServer++ % $nServers, $NewSession = (Call-ID, $MyServer, Now() + 900), Insert(%ActiveSessions, $NewSession), Color $MyServer

30: Response >= 200 && CSeq.Method == “BYE” && *CurrentSession = Call-ID belongs-to %ActiveSessions -> Remove(%ActiveSessions,*CurrentSession), Color *CurrentSession->Server

Session Dispatching

Struct Session = {String ID, Int Server Int Expire}

Global Session: %ActiveSessions

Local Session: $NewSession, *CurrentSession

Local Int: $MyServer

Global Int: $CurrentServer, $nServers

If a packet does not match an existing session, then insert an entry into %ActiveSessions with the value of $CurrentServer and dispatch it to the same

Init -> $CurrentServer = 0, $nServers = *3*, ExpiryThread(%ActiveSessions, Expire)

10:*CurrentSession = Call-ID belongs-to %ActiveSessions -> *CurrentSession.Expire = Now() + 900, Color *CurrentSession.Server

20: NOT Call-ID belongs-to %ActiveSessions -> $MyServer = $CurrentServer++ % $nServers, $NewSession = (Call-ID, $MyServer, Now() + 900), Insert(%ActiveSessions, $NewSession), Color $MyServer

30: Response >= 200 && CSeq.Method == “BYE” && *CurrentSession = Call-ID belongs-to %ActiveSessions -> Remove(%ActiveSessions,*CurrentSession), Color *CurrentSession->Server

Session Dispatching

Struct Session = {String ID, Int Server Int Expire}

Global Session: %ActiveSessions

Local Session: $NewSession, *CurrentSession

Local Int: $MyServer

Global Int: $CurrentServer, $nServers

On receipt of the BYE, remove the call from %ActiveSessions, and send the call to the proper server.

Init -> $CurrentServer = 0, $nServers = *3*, ExpiryThread(%ActiveSessions, Expire)

10:*CurrentSession = Call-ID belongs-to %ActiveSessions -> *CurrentSession.Expire = Now() + 900, Color *CurrentSession.Server

20: NOT Call-ID belongs-to %ActiveSessions -> $MyServer = $CurrentServer++ % $nServers, $NewSession = (Call-ID, $MyServer, Now() + 900), Insert(%ActiveSessions, $NewSession), Color $MyServer

30: Response >= 200 && CSeq.Method == “BYE” && *CurrentSession = Call-ID belongs-to %ActiveSessions -> Remove(%ActiveSessions,*CurrentSession), Color *CurrentSession->Server