PASS Middleware for Distributed and Autonomous XML Message Processing

Dirceu Cavendish and K. Selçuk Candan
Arizona State University

Abstract

Basic message processing tasks, such as well-formedness checking and grammar validation, can be off-loaded from the service providers' own infrastructures. To enable effective off-loading of processing tasks, we introduce the Prefix Automata SyStem - PASS*, a middleware architecture which distributively processes XML payloads of web service SOAP messages during their routing towards Web servers. PASS is based on a network of automata, where PASS-nodes independently but cooperatively process parts of the SOAP message XML payload.*

1 Motivation and Related Work

We propose a novel lightweight middleware architecture called "*Prefix Automata SyStem (PASS)*" for distributed XML document processing. As depicted in Figure 1(a), services provided by PASS middleware lie beneath high level web service execution tasks (including service composition, business application execution, and data access) and above the traditional and/or intelligent network routing.

PASS enables distributed processing of the web service request and reply documents within the public network itself, before they even arrive at their destinations (Figure 1(b)). In particular, PASS-enabled cooperating network nodes listen to SOAP request/reply messages that *pass through* and perform basic XML document processing on their payloads. Processing performed at PASS cooperating network nodes of the PASS middleware are: non-blocking; non-wasteful; autonomous. PASS middleware allows pipelined in-network processing of SOAP messages. Pipelining allows different pieces of a large XML document to be processed by various PASS nodes in tandem or simultaneously. To enable such autonomous and pipelined processing of XML documents, unlike most existing work in communicating automata [3, 7, 2, 6], PASS middleware relies on the *prefix* nature [8] of basic XML processing tasks.

2 Computational Foundations

We now argue that some XML document processing tasks belong to the class called "*prefix computations*".

2.1 XML Documents

We abstract XML documents as trees over an alphabet Σ, as follows [5]:

Definition 2.1 (Tree Document). *A tree document over Σ is a finite ordered tree with labels in Σ. Each tree document T can be represented by a string T_s:*

- *if T is a single node $a \in \Sigma$, then $T_s = \{a, \bar{a}\}$;*
- *if T consists of a tree rooted at node a and with subtrees $ST1, ST_2, \cdots, ST_k$, then $T_s = \{aST_1ST_2 \cdots ST_k\bar{a}\}$, where a and $\bar{a}$ are opening and closing tags.*

Hence, every XML document D over the alphabet Σ has a string representation D_s over alphabet $\Sigma \cup \bar{\Sigma}$, where $\forall a \in \Sigma, \exists\ \bar{a} \in \bar{\Sigma}$ and $|\Sigma| = |\bar{\Sigma}|$. Hereafter, every XML document D will be referred to by its string representation $D_s = \{d_1 d_2 \cdots d_n\}, d_i \in \Sigma \cup \bar{\Sigma}$.

Let $f(D_s)$ denote the processing of a document D by a function f over the document string representationi D_s. The domain of function f is the set of strings $w \in (\Sigma \cup \bar{\Sigma})^*$, with an arbitrary image set, depending on the function.

2.2 Prefix Computations

The class of computational tasks referred to as "prefix computation" were first introduced in the design of logic circuits as a way to parallelize circuits and hence speed up computation such as addition and multiplication [8].

Definition 2.2 (Prefix Function). *Let the pair $(A, \odot)$ of set A and operator $\odot$ form a semi-group; i.e, over the entire set A: i) if x_1 and x_2 are in A, then $x_1 \odot x_2 \in A$; ii) For all $x_1, x_2, x_3 \in A$, $(x_1 \odot x_2) \odot x_3 = x_1 \odot (x_2 \odot x_3)$.*

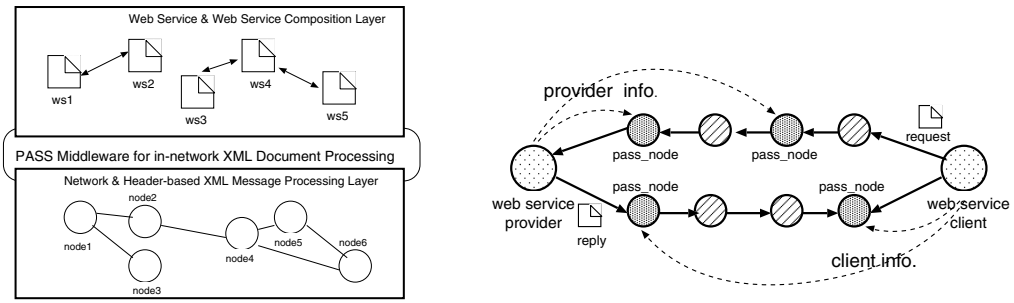
Then, a prefix function $P^n_{\odot}$: $A^n \mapsto A^n$ *on input $\mathbf{x} = (x_1, x_2, \cdots x_n)$ produces an output $\mathbf{y} = (y_1, y_2, \cdots, y_n)$, where $y_j = x_1 \odot x_2 \odot \cdots \odot x_j$, for $1 \leq j \leq n$.*

Intuitively, y_j is a *running* sum of $\odot$ on the first j elements of input $\mathbf{x}$. Again intuitively, since $(A, \odot)$ is associative, the prefix function can be segmented and composed in different ways enabling parallel implementations [8].

Definition 2.3 (Segmented Prefix Computations). *Let the function $P^n_{\odot}$ be defined over two input vectors $\mathbf{x}$ and $\mathbf{c}$, where $x_i \in A$ as before, and $\mathbf{c}$ is a flag vector. The value of output vector component $y_i \in \mathbf{y}$ is defined as:*

$$y_i = \begin{cases} x_i \text{ if } c_i = 1, \\ x_k \odot x_{k+1} \odot \cdots \odot x_i \text{ if } c_k = 1 \text{ and } c_j = 0, \quad k < j \leq i. \end{cases}$$

that is, the flag vector defines segments of 0 values over which running sums on the input vector $\mathbf{x}$ are computed.



(a) PASS middleware (b) PASS-nodes in the network

Figure 1. Prefix Automata SyStem (PASS) middleware architecture: (a) PASS lies between Web service composition/execution and low level network and (header-based) SOAP message processing layers. (b) PASS nodes perform basic XML document processing (e.g., document filtering) on the request and reply messages exchanged by web service clients and providers.

Intuitively, the control vector $\mathbf{c}$ segments the original input vector $\mathbf{x}$ into *to-be-processed* and *not-to-be-processed* vector segments.

Definition 2.4 (Composition of Prefix Computations). *Let $F\colon A^n \mapsto A^n$, and for all i (where $1 \leq i \leq l$), $G_i\colon A^{m_i} \mapsto A^{m_i}$, where $n, m_1, \ldots, m_l \in N^+$, $m_1 + \cdots + m_l = n$. A composite function $H\colon A^m \mapsto A^m$ is defined as*

$$h(x_1, x_2, \cdots, x_m) = f(g_1(x_1, x_2, \cdots, x_{m_1}) \,||\, \cdots \cdots \,||\, g_l(x_{m-m_l+1}, x_{m-m_l+2}, \cdots, x_m))$$

where "||" is the vector composition operation [1].

2.3 XML Document Processing

Let us consider a segmentation of a document string D_s into an arbitrary k number of segments, ds_i, $D_s = \{ds_1 ds_2 \cdots ds_k\}$. Naturally, for PASS-nodes to be autonomous in selecting the segments for computation, f needs to be *associative over arbitrary non-overlapping segments* of D_s, so that for each three consecutive segments ds_{i-1}, ds_i, ds_{i+1}, $f(ds_{i-1}ds_i) \in \Sigma \cup \bar{\Sigma}$, and $f(f(ds_{i-1}ds_i)ds_{i+1}) = f(ds_{i-1}f(ds_i ds_{i+1}))$. In this section, we use XML filtering to exemplify how XML document processing tasks can be cast as functions that are associative over arbitrary non-overlapping segments.

Document filtering can be described as a composition of two prefix computations. For sake of simplicity, we focus on XPath expressions of type $P^{\{/,//,*\}}$. Such path expressions are composed of query steps, each consisting of an axis (parent/child, "/" or ancestor/descendant, "//") test between data elements and a label test (including the "*" wildcard). We focus on the special case where the data and the XPath expression are not recursive; in other words, each tag can occur only once along a given path. A path expression is represented as $PE(N, T, R, P)$, where N is the set of non-terminals, $T = \Sigma \cup \bar{\Sigma}$ is the set of terminals, R is the root of the expression, and P is a set of filtering rules, where R is the non-terminal symbol corresponding to the root of the document.

[1] Hereafter, we might omit the operator wherever concatenation can be easily understood, for conciseness.

Example 2.1. *Let a query be given as "$A/B//C/*/D$". For the sake of convenience we will uniquely identify each "*" wildcard symbol. Since, this query has only one "*", we will rewrite it as "$A/B//C/*_1/D$".*

This filter pattern would generate the following filter representation: $PE = \{N, \Sigma \cup \bar{\Sigma}, R, P\}$, where the non-terminal set is $N = \{A, B, C, D, W_1\}$, the terminal sets are $\Sigma = \{a, b, c, d\}$ and $\bar{\Sigma} = \{\bar{a}, \bar{b}, \bar{c}, \bar{d}\}$, the root element is A, and the filtering rules in P are

- *$A \mapsto aS_1BS_2\bar{a}$; where strings S_1 and S_2 are well-formed strings corresponding to subtrees*
- *$B \mapsto bS_1S_2CS_3S_4\bar{b}$; where strings S_1 and S_4 are well-formed and the string S_2S_3 is well-formed.*
- *$C \mapsto cS_1W_1S_2\bar{c}$; where strings S_1, S_2 are well-formed.*
- *$W_1 \mapsto \diamondsuit S_1 D S_2 \bar{\diamondsuit}$; where strings S_1, S_2 are well-formed.*
- *$D \mapsto dS_1\bar{d}$; where the S_1 is a well-formed string corresponding to a (possibly empty) subtree.*

Here, "$\diamondsuit$" represents an arbitrary symbol.

Definition 2.5 (Filtering Function). *We can obtain the filtering validation function f over string S as follows:*

$$f(S) = \begin{cases} f(S'AS'') & \text{if } \exists\, a, \bar{a} \in T;\ B \in N \text{ s.t. } S = S'aS_aBS_b\bar{a}S'' \text{ and } w(S_a) = w(S_b) = \perp \text{ and } A/B \text{ is a query step in the filter statement} \\ f(S'AS'') & \text{if } \exists\, a, \bar{a} \in T;\ B \in N \text{ s.t. } S = S'aS_aS_bBS_cS_d\bar{a}S'' \text{ and } w(S_a) = w(S_d) = w(S_bS_c) = \perp \text{ and } A//B \text{ is a query step in the filter statement} \\ f(S'W_iS'') & \text{if } \exists\, \diamondsuit, \bar{\diamondsuit} \notin T;\ B \in N \text{ s.t. } S = S'\diamondsuit S_aBS_b\bar{\diamondsuit}S'' \text{ and } w(S_a) = w(S_b) = \perp \text{ and } *_i/B \text{ is a query step in the filter statement} \\ f(S'W_iS'') & \text{if } \exists\, \diamondsuit, \bar{\diamondsuit} \notin T;\ B \in N \text{ s.t. } S = S'\diamondsuit S_aS_bBS_cS_d\bar{\diamondsuit}S'' \text{ and } w(S_a) = w(S_d) = w(S_bS_c) = \perp \text{ and } *_i//B \text{ is a query step in the filter statement} \\ S & \text{otherwise} \end{cases}$$

Theorem 2.1 (Associativity of the Filtering Function). *For an arbitrary string $S \in T^*, T = \Sigma \cup \bar{\Sigma}$, and for any arbitrary string partition $S = S_1||S_2||S_3$, we have*

$$f(S_1||S_2) \in (N \cup T \cup E)^*, \quad f(S_2||S_3) \in T^*, \quad (1)$$

$$f(f(S_1||S_2)||S_3) = f(S_1||f(S_2||S_3)) = f(S_1||S_2||S_3) \quad (2)$$

Verification of Eq. 1 comes straight from the definition of f and does not require an explicit proof. The inductive proof for 2 is omitted. In an extended version of this paper, we show that three fundamental XML document processing operations, well-formedness checking, grammar validation, and filtering are prefix functions.

3 PASS Middleware

Prefix Automata SyStem (PASS) middleware consists of a set of PASS-nodes interconnected by network links that process and relay segments of SOAP messages between nodes. PASS nodes avoid reading processed messages (or even processed parts of the partially processed messages) through a small *directory* attached to the segments.

3.1 SOAP Messages and Segments

We focus on document-style SOAP messages, which carry large documents, such as purchase orders, and the like [1]. SOAP messages may be stored in its entirety, and part of it be processed at a PASS node before relaying the entire message to a next node. Alternatively, PASS nodes may segment and reassemble an XML document carried by a SOAP message body in a number of different ways. HTTP/1.1 allows for a chunked transfer-coding, which may be used for segmentation/reassembly purposes.

3.2 PASS-Nodes

PASS-nodes perform both routing and basic XML document processing tasks on behalf of the recipient

1. If a PASS-node receives one or more segments belonging to an XML document and decides to process them, then it locally stores the segments.
2. For a consecutive sequence of segments, a PASS-node performs the relevant processing. The part of the document just processed is pushed back into the network, with appropriate flags included in the segment data to inform other, downstream PASS-nodes, that part of the document has been processed.
3. A PASS-node may stop processing segments of a particular XML document at any point in time. However, the segments of a given document are kept in order, i.e., processed and unprocessed segments leave the PASS node in the order at which they have arrived.

Further details and other XML document processing tasks, such as well-formedness checking and grammar validation, are discussed in an extended version of the paper[4].

4 Distributed XML Processing Evaluation

We abstract the processing functions performed, so as to avoid conclusions dependent on specific functions:

Clients/Message senders. A client is abstracted by an element that generates XML documents of various sizes at specific time epochs. Each document is partitioned into fixed size segments and injected into the PASS network domain.

Servers/Message receivers. A server is an element that receives processed or non-processed XML segments.

PASS-nodes. A PASS node is abstracted by an element with two major functions: segment switching and segment processing. Segment transmission speed is limited on outgoing network interfaces. There is a dedicated queue for each XML stream on each PASS-node output interface.

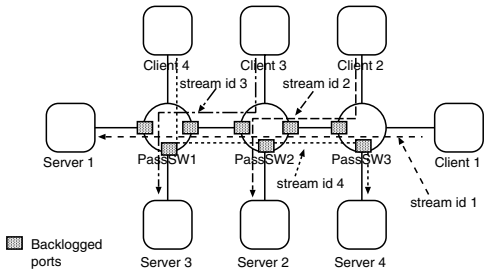


Figure 2. Network scenario: arrows denote the client server routes. Filled squares denote interfaces where segments are stored for processing and transmission.

A PASS-node processing scheduler decides which segments to process at any given time. Two scheduling strategies are implemented: *shared processing*, where processing power is equally shared by all streams served by the node, and *batch processing*, where segments of a given XML document are processed until there are no more segments from that document in the queue. Segments belonging to a given document are routed through a single route between the client and the server and switched by each PASS-node in a first-in-first-out manner.

4.1 Performance Measures

We use the following performance measures: Number of segments received by the servers ($rec(server_i)$) is a rough measure of network savings. Number of unprocessed segments received by the servers ($unpro\_rec(server_i)$) is a rough measure of the processing savings at the servers. Amount of work done at the switches ($work(switch_j)$) measures work done by the switches that would have to be done at the servers if PASS processing was absent.

4.2 Network Setup

The PASS middleware network scenario used in the evaluation is depicted in Fig. 2. There are four XML streams, between (C_1, S_1), (C_2, S_2), (C_3, S_3), (C_4, S_4) pairs of client/servers. There are three PASS-nodes in the network. Node interface speeds are uniform (and in the results we have normalized them to 1 segment per second). Document processing speed has been normalized to the interface transmission speed. That is, $2\times$ means that two segments are processed during the time taken to transmit one segment. Other parameters are: *document average size* is 50 segments; network link delay is 6 segments. On a network of 1Gbps link speed, these parameters represent a metro area network, with 20Km separation between PASS nodes. The simulation was run for 20000 time units (1 TU = time to transmit a segment), with a warm up and cool off periods of 200 TUs. This means that an average of 200 documents were generated per client throughout the entire simulation.

4.3 Results

Tables 1- 3 show the results, in a setup where the processing capacity of each PASS node is equal to the band-

Segments received [BATCH,1×]					Unprocessed segments received [BATCH,1×]					Partially processed segments received [BATCH,1×]				
Err.Rate	Server1 (9933)	Server2 (10122)	Server3 (9983)	Server4 (8970)	Err.Rate	Server1 (9933)	Server2 (10122)	Server3 (9983)	Server4 (8970)	Err.Rate	Server1 (9933)	Server2 (10122)	Server3 (9983)	Server4 (8970)
0.0	9933	10122	9983	8970	0.0	0	3045	3362	0	0.0	5731	1113	1091	3100
0.2	5816	6009	1220	3568	0.2	0	4323	299	0	0.2	258	1823	258	356
0.5	3876	3937	320	1715	0.5	0	135	47	0	0.5	73	1013	63	82
0.8	1676	1767	70	348	0.8	0	84	3	0	0.8	9	449	13	3

Segments received [SHARED,1×]					Unprocessed segments received [SHARED,1×]					Partially processed segments received [SHARED,1×]				
Err.Rate	Server1 (9933)	Server2 (10122)	Server3 (9983)	Server4 (8970)	Err.Rate	Server1 (9933)	Server2 (10122)	Server3 (9983)	Server4 (8970)	Err.Rate	Server1 (9933)	Server2 (10122)	Server3 (9983)	Server4 (8970)
0.0	9933	10122	9983	8970	0.0	0	0	3402	0	0.0	0	9830	1994	3370
0.2	5905	6070	1211	3664	0.2	0	174	318	0	0.2	82	1819	247	389
0.5	3837	3909	299	1756	0.5	0	138	42	0	0.5	66	962	55	72
0.8	1622	1655	61	330	0.8	0	66	4	0	0.8	17	406	6	8

Table 1. The number of segments received by the servers and the portion of those segments that are not yet processed or partially processed by PASS (*the numbers in parenthesis denote the number of segments destined for the corresponding server*)

Unprocessed segments received [BATCH][Err.rate 0.0]					Partially processed segments received [BATCH][Err.rate 0.0]				
Proc.Power	Server1 (9933)	Server2 (10122)	Server3 (9983)	Server4 (8970)	Proc.Power	Server1 (9933)	Server2 (10122)	Server3 (9983)	Server4 (8970)
none	9933	10122	9983	8970	none	0	0	0	0
1×	0	3045	3362	0	1×	5731	1113	1091	3100
2×	0	577	0	0	2×	324	327	1	0

Table 3. The amount of segments received unprocessed and partially processed by the servers as a function of the processing power available at the PASS nodes (*the numbers in parenthesis denote the numbers of segments destined for the corresponding servers*)

Work done at PASS-nodes[BATCH,1×]				Work done at PASS-nodes[SHARED,1×]			
Err.Rate	Node1	Node2	Node3	Err.Rate	Node1	Node2	Node3
0.0	13687	19808	19888	0.0	15383	19990	19895
0.2	6445	12900	15060	0.2	6324	14122	15020
0.5	4542	10051	14907	0.5	4485	10265	14895
0.8	2415	6925	14875	0.8	2187	6543	14873

Table 2. The amount of work done at each PASS node

width of a single network interface. In particular, Table 1 presents the number of segments received by the servers under different error rates and process scheduling strategies. We see that the number of unprocessed segments arriving at the servers is minimal relative to the numbers of segments injected into the network by the clients. Furthermore, when the error rate in data increases, the number of segments arriving to the destinations drops significantly. In other words, the PASS distributed framework is able to capture and eliminate faulty XML documents. Shared scheduling enables PASS to process slightly more segments relative to the batch scheduling (XML documents going through fewer nodes).

Table 2 shows that the non-blocking, non-wasteful, and autonomous nature of the PASS nodes enable PASS middleware to shift the processing load across available nodes under different traffic conditions. For instance, as the error rate increases, the loads of the first and second nodes vary significantly to maintain the amount of segments received unprocessed by the servers close to 0.

Finally, Table 3 presents the amount of unprocessed segments received by the servers as a function of the processing power available at the nodes. As expected, as the processing power of the nodes increases, the number of unprocessed segments arriving at the servers drop.

5 Conclusion

In this paper, we presented a novel PASS middleware for outsourcing basic SOAP message processing tasks (such as well-formedness checking, grammar validation, and filtering) that are traditionally done by the receiver at the expense of server and network resources and end-to-end service delivery delays. The PASS nodes perform document processing tasks that can be cast as *"prefix computations"*, distributively and autonomously. We also presented event-driven simulation results which shows that a PASS middleware provides significant network and server resource savings, under varying traffic conditions.

References

[1] SOAP specifications. http://www.w3.org/TR/soap/, 2003.

[2] J. Barnard, J. Whitworth, and M. Woodward. Communicating x-machines. *Information and Software Technology*, 38(6):401–407, 1996.

[3] T. L. Casavant and J. G. Kuhl. A communicating finite automata approach to modeling distributed computation and its application to distributed decision-making. *IEEE Trans. Comput.*, 39(5):628–639, 1990.

[4] D. Cavendish and K.S. Candan. A prefix based framework for distributed XML document processing. *Submitted for publication,* 2006.

[5] C. Chitic and D. Rosu. On validation of xml streams using finite state machines. In *WebDB*, 2004.

[6] E. W. Endsley and D. M. Tilbury. Modular verification of modular finite state machines. In *IEEE Conference on Decision and Control*, volume 1, pages 972–979, 2004.

[7] R. Hierons. Checking states and transitions of a set of communicating finite state machines. *Microprocessors and Microsystems, Special Issue on Testing and testing techniques for real-time embedded software systems*, 24(9):443–452, 2001.

[8] R. Ladner and M. Fischer. Parallel prefix computation. *J. Assoc. Comput. Mach.*, pages 831–838, 1980.