Distributed XML processing: Theory and applications

Dirceu Cavendish, K. Selçuk Candan

Abstract

Basic message processing tasks, such as well-formedness checking and grammar validation, common in Web service messaging, can be off-loaded from the service providers’ own infrastructures. The traditional ways to alleviate the overhead caused by these tasks is to use firewalls and gateways. However, these single processing point solutions do not scale well. To enable effective off-loading of common 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. PASS allows autonomous and pipelined in-network processing of XML documents, where parts of a large message payload are processed by various PASS-nodes in tandem or simultaneously. The non-blocking, non-wasteful, and autonomous operation of PASS middleware is achieved by relying on the prefix nature of basic XML processing tasks, such as well-formedness checking and DTD validation. These properties ensure minimal distributed processing management overhead. We present necessary and sufficient conditions for outsourcing XML document processing tasks to PASS, as well as provide guidelines for rendering suitable applications to be PASS processable. We demonstrate the advantages of migrating XML document processing, such as well-formedness checking, DTD parsing, and filtering to the network via event driven simulations.

1. Motivation and related work

As web service standardization efforts get mature and many institutions embrace these standards and available services as means to reduce their development and maintenance costs, web services are becoming ubiquitous and irreplaceable components of (e-)businesses. Consequently, end-to-end delivery, processing, and routing of web service requests (and the underlying SOAP messages [40]) have significant impact on the response times observed by end-users of these services.

Most commercial web sites pay premium prices for solutions that help them reduce their response times as well as risks of failure when faced with high access rates. Delivering a web service requires careful design in many aspects: security, robustness, and sustained performance are three of these issues. A web service request and its reply consumes resources both at public networks (SOAP header processing and routing) as well as at the service provider’s own network and servers. A typical Web service provider might have to allocate significant resources (in terms of network, memory, CPU, as well as I/O) for verifying and processing each incoming service request. Consequently, a high request load may translate into poor service delivery performance for the Web service provider. In fact, denial of service attack, where a Web service provider fails to deliver services due to malicious, invalid, and unauthorized accesses to the system, is an extreme case of service degradation due to resource limitations.

Generally, the above challenges are addressed in two orthogonal, but complementary mechanisms: back-end solutions and front-end solutions. Many services benefit from both back-end solutions (such as load balanced server farms) as well as front-end solutions (including, edge-caches, proxy-servers, in-network query filters and query routers, and distributed view managers). Most high volume sites typically deploy a large number of servers and employ hardware- or software-based load balancing components to reduce the response time of their back-end servers. Although they guarantee some protection against surges in demand, such localized solutions cannot help reduce the delay introduced in the network during the transmission of the content to end-users. In order to alleviate this problem, content providers also replicate or mirror their content at edge caches; i.e. caches that are close to end users. If data or process can be placed in a proxy server or cache closer to end-users, when a user requests the service, it can be delivered promptly from the cache without additional communication with the Web server, reducing the response time.
approach also reduces the load on the original source as some of the requests can be processed without accessing the source. Thus, to protect themselves against performance degradations when facing high request load, Web service providers use solutions such as replication [2,1,19,35], caching [35,9,11,43,6,41], and even off-loading to content- and application-delivery networks [2,26,31]. Unfortunately, many services (especially transaction-oriented and data-driven ones) cannot easily benefit from caching and offloading as business data and transactions might require local processing. Therefore, in order to prevent service providers' throughput to be negatively affected by high request volumes, any wastage of Web service provider resources should be avoided.

Despite the challenges associated with off-loading entire web services, we note that there are parts of the service request processing that are common across many web service delivery and service composition deployments. As shown in Fig. 1, most service requests (even those that are transaction oriented and/or database-driven) consume service providers' resources for highly common and mundane tasks, such as XML message validation (well-formedness checking, DTD validation) as well as various web service value added features (e.g., content-based message filtering). Since there is little sharing and caching opportunities applicable, such low level document processing tasks are quickly becoming a bottleneck in service delivery infrastructures. Indeed XML parsing is now recognized to be a bottleneck in XML processing. Valuable network and CPU resources are wasted and service delays are incurred because processing of these tasks has to be done by the service provider or (at best) by XML appliances (including XML firewalls, such as WebSphere DataPower Security Gateway XSO40 [15] and Sarvega Security Gateway [37]). XML accelerators, such as WebSphere DataPower XA35 XML Accelerator [16] deployed at the edge of the service providers infrastructure.

Naturally, such solutions which delay XML processing to the end of the are subject to bottlenecks, single points of failure, and denial of service attacks. More importantly, they still waste valuable network resources. In fact, there is an increasing number of XML message process off-loading technologies that provide support for processing individual requests. However, these technologies are either high-level (usually proxy-based) publish/subscribe solutions (such as [42], JMS [24], XPASS [34], NiagaraCQ [12], SemCast [33], CoDD [3]), or are purely network-level intelligent message routing solutions (such as WebSphere DataPower Integration Appliance XI50 [18] and Sarvega XML Context Router [36]), which do not go beyond interpreting the request and reply message headers, and do not support XML document level processing.

With recent advances in network processors, on the other hand, hardware assisted in-network XML processing have started to be investigated. [29] has proposed to use Network Processors to execute XML processing. NPs can be placed at various places in the network, from edge routers, for caching and firewall applications, to routers, to “move application services into the network”. Once application services are identified, they are mapped into the NP pipeline structure (microengines), for efficient processing. In a related work, [14] investigates XML DOM parsers that can be implemented in Networked Embedded Systems with real time requirements. Their approach involves pre-allocation of memory objects, so that dynamic memory allocation is avoided, with its unpredictability in performance. The work underlines the importance of an efficient parsing and document validation tasks within XML processing. Since XML data is expected to become a significant part of network load [32], the development of scalable in-network solutions that can save both network and server resources, as well as eliminate spurious/faulty messages within the network becomes essential.

1.1. Contributions

In this paper we note that basic XML document processing tasks can be off-loaded to a middleware, even if the web services that rely on these tasks themselves have to be provided within the Web service provider infrastructure. Our goal is to go beyond header-level processing and enable efficient off-loading of basic XML payload processing tasks. Thus, we propose a novel lightweight middleware architecture called “Prefix Automata SyStem (PASS)” for distributed XML document processing. As depicted in Fig. 2(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 (Fig. 2(b)). In particular, PASS-enabled cooperating network nodes listen to SOAP request/reply messages that pass through and perform basic XML document processing.
processing on their payloads. Processing performed at PASS cooperating network nodes of the PASS middleware are:

- **non-blocking**: no network node is stalled with a single XML document. A PASS node can perform partial processing if there are not enough local resources to complete the entire task. The remaining portion of the task will be completed either by another PASS node or by the Web service provider.
- **non-wasteful**: when a PASS node performs part of a task, the work done at this node is not lost or repeated at any other PASS node or at the destination.
- **autonomous**: each PASS node can decide whether to process a message and how much time and resources it will allocate to its processing.

Therefore, the 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 [10,23,5,21], PASS middleware focuses and relies on the prefix nature [28] of many basic XML processing tasks.

Naturally PASS does not outsource the entire service workflow. PASS outsourcing is 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 outsources those low-level components of service request processing that are common across many web service delivery and service composition deployments and which can provide significant opportunities for back-end load reduction if performed in-network. By targeting the XML processing that is required by each and every request, processing savings can be obtained regardless of the workflow/transaction state. We note that as long as there remain processing to be done at servers, improvements at back-end servers are indeed necessary to increase performance (response time and throughput).

In Section 2, we discuss the computational foundations of PASS and show that basic XML processing tasks, such as well-formedness checking and DTD validation, can be executed as prefix computations. In Section 3, we present the details of PASS middleware architecture, which relies on the casting of XML document processing into prefix type of computation to offload basic XML processing tasks underlying most web services. In Section 4, we present a performance evaluation of PASS middleware, via event driven simulation. We conclude with a discussion on how our distributed XML processing approach relates to a traditional parallel computation paradigm.

### 2. Computational foundations for distributed XML processing

Enabling non-blocking, non-wasteful, and autonomous processing of SOAP messages require PASS-nodes to make independent decisions regarding whether to process an XML document and, if so, how much time and resources to allocate for this task. In this section, we show that some XML document processing tasks belong to the class of computations called “prefix computations”. We then provide constructive proofs which act as guidelines for developing protocols for rendering applications, which are prefix computations, PASS processable.

#### 2.1. XML documents

We abstract XML documents as trees over an alphabet $\Sigma$, as follows [13]:

**Definition 2.1** (Tree Document and its String Representation). A tree document over $\Sigma$ is a finite ordered tree with labels in $\Sigma$. Each tree document $T$ can be represented by a string $T$:

- if $T$ is a single node $a \in \Sigma$, then $T = [a]$;
- if $T$ consists of a tree rooted at node $a$ and with subtrees $T_1, T_2, \ldots, T_n$, then $T = [aT_1T_2\ldots T_n]$, where $a$ and $a$ are opening and closing tags.

Hence, every XML document $D$ over the alphabet $\Sigma$ has a string representation $D_\Sigma$ over $\Sigma$ such that $\forall a \in \Sigma, D_a \in \Sigma$ and $|\Sigma'| = |\Sigma|$. Hereafter, every XML document $D$ will be referred to by its string representation $D_\Sigma = [d_1d_2\cdots d_n]$, $d_i \in \Sigma \cup \Sigma$.

We refer to specific processing of a document $D$ by a function $f$ over the document string representation, or $f(D_\Sigma)$. The domain of function $f$ is the set of strings $w \in (\Sigma \cup \Sigma)^*$, with an arbitrary image set, depending on the function.

#### 2.2. Prefix computations and segmentation

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 [28].

**Definition 2.2** (Prefix Function). Let the pair $(A, \odot)$ 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 $P^\Sigma_\odot : A^* \mapsto A^*$ on input $x = (x_1, x_2, \ldots, x_n)$ produces an output $y = (y_1, y_2, \ldots, y_n)$, where $y_i = x_1 \odot x_2 \odot \cdots \odot x_i$, for $1 \leq j \leq n$.

Intuitively, $y_i$ is a running sum of $\odot$ on the first $j$ elements of input $x$. Again intuitively, since $(A, \odot)$ is associative, the prefix function can be segmented and composed in different ways enabling parallel implementations [28].

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

Intuitively, the control vector $c$ segments the original input vector $x$ into to-be-processed and not-to-be-processed vector segments. In a sense, the responsibility of the PASS middleware is to ensure that the PASS-nodes can select appropriate prefix segments autonomously, and in a way that the processing results of these segments can be composed in a non-wasteful manner.

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

$$h(x_1, x_2, \ldots, x_m) = f(g_1(x_1, x_2, \ldots, x_m), \ldots, g_l(x_{m-l+1}, x_{m-l+2}, \ldots, x_m))$$

where “||” is the vector composition operation.\(^1\)

\(^1\)Hereafter, we might omit the operator wherever concatenation can be easily understood, for conciseness.
Notice that this composition definition is rich, in the sense that allows many particular cases. In general, however, we are interested in whether we can infer about the associativity of the composite function \( h \), given that the associativity property holds for its \( f \) and \( g \) functions. In other words, let an input string \( S \) be partitioned into three arbitrary segments, \( S = S_1S_2S_3 \). We wish to know under which circumstances, we have \( h(S_1h(S_2S_3)) = h(h(S_1S_2)S_3) \). Let \( R_i = g(S_i) \) represent the vector result of applying \( g \) function over segment \( S_i \). Then, from Definition 2.4, we have
\[
h(S_1h(S_2S_3)) = h(S_1f(R_1R_2)) = f(R_1g(f(R_1R_2))).
\]
From the other direction, we have
\[
h(S_1h(S_2S_3)) = h(f(R_1R_2)S_3) = f(g(f[R_1R_2])R_3).
\]
Thus, if \( h \) is associative, then we need to have
\[
f(R_1g[f(R_1R_2)]) = f(g[f(R_1R_2)]R_3)
\]
for all \( R_1, R_2 \) and \( R_3 \). This, for instance, would be true if \( g = f^{-1} \) or if \( g \) is the identity function and \( f \) is associative. Note that especially the second case, where \( g \) is the identity function and \( f \) is an associative function implies that, as long as two functions use different symbol sets, they can be arbitrarily composed and processed by PASS.

Note that a common property of these two special cases is that \( f \) and \( g \) commute, in addition to both being associative. Thus, we can further generalize these two special cases as follows: let \( f \) and \( g \) be two functions that are associative and also commutative with each other. Let us also assume that \( gg = g \). Then,
\[
f(R_1g[f(R_1R_2)]) = f(g(S_1g[f(S_1S_2)])) = g(f(S_1)g(f(S_1S_2)))
\]
which is associative. Note that \( gg = g \) is not a restrictive requirement; the “non-wasteful” and “automonomus” behavior of PASS requires that if part of a string is processed by a function \( g \), then re-application of \( g \) on the processed segment should not result in any further useful computation. The well-formedness checking, grammar validation, and filtering functions that we describe next all satisfy this condition.

2.3. XML document processing with prefix computations

PASS-nodes perform XML-document processing tasks through segmented prefix computations. Let us consider a segmentation of a document string \( D \) into an arbitrary \( k \) number of segments, \( d_1, d_2, \ldots, d_k \). Naturally, for PASS-nodes to be autonomous in selecting the segments for computation, \( f \) needs to be associative over arbitrary non-overlapping segments of \( D \), so that for each three consecutive segments \( d_{i-1}, d_i, d_{i+1} \), \( f(d_{i-1}d_i d_{i+1}) = f(d_i) \). In this section, we exemplify how XML document processing tasks can be cast as functions that are associative over arbitrary non-overlapping segments. In particular, we discuss three of these common document processing tasks, well-formedness checking, grammar validation, and filtering, showing how they can be cast as segmented prefix computation tasks.

2.3.1. Well-formedness checking

Well-formedness checking is a basic messaging operation which eliminates faulty messages before valuable CPU cycles are spent in processing its content. Since most higher-level processing tasks, such as DTD-validation or filtering, assumes that the input documents are well-formed, this is a fundamental XML-document processing task, sometimes performed by firewalls [17,37]. The input to well-formedness check are the XML document tags, where each begin-tag needs to be matched against its end-tag.

**Definition 2.5 (Well-formedness Function).** Formally, we can state the well-formedness function \( w \) over string \( S \) as follows:
\[
w(S) = \begin{cases} S & \text{if } \exists a \in \Sigma \text{ s.t. } S = S′aS′′ \text{ otherwise.} \\
\end{cases}
\]

Intuitively, the well-formedness function \( w \) returns an empty string if all begin tags in the input string are properly matched against the corresponding end tags, or a non-empty string otherwise. Theoretically, well-formedness checking is known to be a context-free operation, which can be performed using a pushdown automaton (PDA), with a stack to maintain start-tags until a corresponding end-tag is observed. Although this computation model works well for centralized validation (e.g., at a firewall), enabling non-blocking, non-wasteful, yet autonomous processing requires us to cast the function using the prefix computation model. Thus, in what follows, we wish to prove the following theorem:

**Theorem 2.1** (Associativity of the Well-formedness Function). For an arbitrary string \( S \in \Sigma^* \), \( \Sigma = \Sigma \cup \bar{\Sigma} \), and arbitrary string partition \( S = S_1 \cup S_2 \cup S_3 \), we have
\[
w(S_1 \cup S_2 \cup S_3) = w(S_1 || w(S_2 || S_3)) = w(S_1 || S_2 || S_3). \quad (1)
\]
Eq. (1) comes straight from the definition of \( w \). Therefore, we do not need to prove it explicitly. On the other hand, Eq. (2), which is fundamental for autonomous computation of \( w \), requires a proof. Here, we will present this proof, using induction on the length of \( S \).

**Proof of Theorem 2.1.** The base case, where \( |S| = 0 \), is trivially true, since \( w(S) = 0 \). For the induction basis, we consider segments of length one. Let \( |S_1| = 1 \), \( |S_2| \leq M \), and \( |S_3| \leq M \). Then, we have the following cases for the locations of \( a \) and \( \bar{a} \):

1. \( S_1 = S′aS′′ \), \( S_2 = S_3 \)
2. \( S_1 = S′aS′′ \), \( S_3 = S_2 \)
3. \( S_1 = S_3 \), \( S_2 = S_1 \)
4. \( S_1 = S_3 \), \( S_2 = S_3 \)
5. \( S_1 = S_2 \), \( S_3 = S_1 \)
6. \( S_1 = S_3 \), \( S_2 = S_3 \)
7. \( S_1 = S_3 \), \( S_2 = S_3 \)

Here, we will use induction for cases (3) (both \( a \) and \( \bar{a} \) are contained in a single partition) and (4) (\( a \) and \( \bar{a} \) are split into
consecutive partitions). The other cases can be dealt with in a similar way.

- **Case (3).** In this case, both $a$ and $\bar{a}$ are contained in $S' = S'_{1}\bar{a}aS''_{1}$.
  Let us refer to $S'_{1} \parallel S_{2}$ as $S''_{1}$. Then, we have

$$w(S'_{1}\bar{a}aS''_{1} \parallel S_{2} \parallel S_{3}) = w(w(S'_{1}aS''_{1} \parallel S_{2}) \parallel S_{3})$$

Thus, by Eqs. (8) and (9), we have

$$w(S'_{1}aS''_{1} \parallel S_{2} \parallel S_{3}) = w(S'_{1} \parallel S''_{1} \parallel S_{2}).$$

(10)

Since $|S'_{1} \parallel S''_{1}| < M$, by inductive hypothesis, we also have

$$w(S'_{1} \parallel S''_{1} \parallel S_{2}) = w(S'_{1} \parallel S''_{1} \parallel S_{2}).$$

(9)

Now, let us consider $w(S'_{1}aS''_{1} \parallel w(S_{2} \parallel S_{3}))$. Since, by the definition of well-formedness checking function, $w(S_{2} \parallel S_{3})$ is a string over the alphabet $\Sigma' \cup \Sigma$, by Definition 2.5, we can state that

$$w(S'_{1}aS''_{1} \parallel w(S_{2} \parallel S_{3})) = w(S'_{1} \parallel S''_{1} \parallel w(S_{2} \parallel S_{3})).$$

(11)

Once again, by inductive hypothesis, we also have

$$w(S'_{1} \parallel S''_{1} \parallel w(S_{2} \parallel S_{3})) = w(S'_{1} \parallel S''_{1} \parallel S_{2} \parallel S_{3}).$$

(12)

Thus, by Eqs. (11) and (12), we have

$$w(S'_{1}aS''_{1} \parallel w(S_{2} \parallel S_{3})) = w(S'_{1} \parallel S''_{1} \parallel S_{2} \parallel S_{3}).$$

(13)

Also, by Definition 2.5, it is easy to see that

$$w(S'_{1}aS''_{1} \parallel S_{2} \parallel S_{3}) = w(S'_{1} \parallel S''_{1} \parallel S_{2} \parallel S_{3}) = w(S'_{1} \parallel S''_{1} \parallel S_{3}).$$

(14)

Finally, by Eqs. (10), (13) and (14), we can state that

$$w(w(S'_{1}aS''_{1} \parallel S_{2} \parallel S_{3})) = w(S'_{1}aS''_{1} \parallel w(S_{2} \parallel S_{3})) = w(S'_{1}aS''_{1} \parallel S_{2} \parallel S_{3}).$$

Thus, we can state that, for Case (2) and (3) and hence the associativity property holds.

- **Case (4).** In this case, $a$ and $\bar{a}$ are split into consecutive partitions. Thus,

$$w(S_{1}a \parallel \bar{a}S'_{2} \parallel S_{3}) = w(w(S_{1}a \parallel \bar{a}S'_{2} \parallel S_{3}))$$

Similarly, we have

$$w(S_{1}a \parallel \bar{a}S'_{2} \parallel S_{3}) = w(S_{1} \parallel \bar{a}S'_{2} \parallel S_{3}).$$

(15)

Therefore, to prove Eq. (2), we need to show that $w(S_{1}a \parallel \bar{a}S'_{2} \parallel S_{3})$ is also equivalent to $w(S_{1} \parallel \bar{a}S'_{2} \parallel S_{3})$. For this, we need the following lemma:

**Lemma 2.1.** $w(\bar{a}S) = \bar{a}w(S)$.

Intuitively, this lemma states that the well-formedness function can simply ignore unmatched end-tags in the prefix of its input. The proof of this lemma is also through an induction on the length of $S$ (see the Appendix for the proof).

Given this lemma, we have

$$w(S_{1}a \parallel \bar{a}S'_{2} \parallel S_{3}) = w(S_{1}a \parallel \bar{a}w(S'_{2} \parallel S_{3}))$$

$$= w(S_{1} \parallel \bar{a}w(S'_{2} \parallel S_{3}))$$

Thus, by Eqs. (15)–(17), we can state that, for Case (2) and (4) and the associativity holds.

This completes the proof of Theorem 2.1, which states that the property of associativity, fundamental for autonomous and non-wasteful segmented computation holds for well-formedness checking.

2.3.2. Grammar validation

Grammar validation is the process through which the service verifies the validity of a received message against a registered DTD to ensure that it is structurally valid and processable [13,38,22,24]. We use the following grammar representation of DTDs [13]: A DTD is represented as $G(N, T, R, P)$, where $N$ is the set of non-terminals, $T = \Sigma' \cup \Sigma$ is the set of terminals, $R$ is the root, and $P$ is a set of production rules.

**Example 2.1.** For instance, the DTD $D = (r, \Sigma, P)$, where $r$ is the root element, $\Sigma = \{a, b, c\}$ is its alphabet, and $P = \{r \rightarrow a^{*}b; a \rightarrow bc; b \rightarrow c + e; c \rightarrow c\}$ its production rules would generate the following grammar representation: $G_{\text{dtd}} = (N, \Sigma \cup \bar{\Sigma}, R, P)$, where the non-terminal set $N = \{R, A, B, C\}$, the terminal sets are $\Sigma = \{r, a, b, c\}$, $\bar{\Sigma} = \{r, a, b, c\}$, the root element is $R$, and the production rules are $P[R \rightarrow ra^{*}b]; A \rightarrow aBCa; B \rightarrow b(c + e); C \rightarrow cc$.

The strings represented by the right-hand sides of the production rules in XML DTDs correspond to regular expressions and, thus, can be recognized by finite state machines [4]. In fact, [4] uses this fact to develop an incremental, divide-and-conquer type of validation mechanism for XML documents. In particular, by focusing on efficient verification of updates to a given document, [4] introduces the concept of $k$-local inputs, which can be validated by inspecting a string region within $k$ distance from the update. Our current work can also be seen as an incremental, divide-and-conquer type of validation mechanism.

The focus is to validate updates by checking only an immediate neighborhood of the given update, our focus is the validation of the entire XML document in incremental chunks. Since we are not constrained by the positions of specific updates, we can pose the grammar validation processing recursively as we did for the well-formedness checking function in Definition 2.5:

**Definition 2.6.** (Grammar Validation Function). Formally, we can state the grammar validation function $v$ over string $S$ as follows:

$$v(S) = \begin{cases} v(S'\bar{a}S'') & \text{if } S_{a} \in T \text{ and } S'' \in N^{*} \\
S & \text{if } S_{a} \in T \text{ and } S'' \in N^{*} \\
S & \text{if } S_{a} \in T \text{ and } S'' \in N^{*} \text{ and } S_{a} \not\in RE_{A} \end{cases}$$

(19)

In the above recursion, $RE_{A}$ corresponds to the regular expression to the right-hand side of the production rule for the non-terminal $A$.

Intuitively, the grammar validation function $v$ either returns an empty string if the string is validated against the given DTD, or returns a non-empty string otherwise. In order to show that this function can be computed in a non-blocking, non-wasteful, and autonomous manner, we would like to show that $v$ is a prefix function; i.e., we wish to prove that $(v, N \cup T \cup E)$ form a semi-group.

**Theorem 2.2.** (Associativity of the Grammar Validation Function). For an arbitrary string $S \in T^{*}$, $T = \Sigma \cup \bar{\Sigma}$, and for any arbitrary string partition $S = S_{1} \parallel S_{2} \parallel S_{3}$, we have

$$v(S_{1} \parallel S_{2}) \in (N \cup T \cup E)^{*}, \quad v(S_{2} \parallel S_{3}) \in T^{*}, \quad v(S_{1} \parallel S_{2}) \parallel v(S_{2} \parallel S_{3}) = v(S_{1} \parallel S_{2}) \parallel S_{3}. \quad (18)$$

$$v(S_{1} \parallel S_{2} \parallel S_{3}) = v(S_{1} \parallel v(S_{2} \parallel S_{3})) = v(S_{1} \parallel S_{2} \parallel S_{3}). \quad (19)$$
Verification of Eq. (18) comes straight from the definition of \( v \) and does not require an explicit proof. Next, we present an inductive proof for Eq. (19).

**Proof of Theorem 2.2.** The base case, where \(|S| = 0\), required for induction follows from the definition of \( v \):

\[
v(\varepsilon \mid \varepsilon \mid \varepsilon) = v(\varepsilon \mid \varepsilon \mid \varepsilon) = v(\varepsilon \mid \varepsilon \mid \varepsilon) = \varepsilon.
\]

Next, let us consider non-empty strings. Our inductive hypothesis (IH) is that for \(|S| \leq M\), Eq. (19) holds. Let \( S \neq \varepsilon \) be such that \( aS \in \Sigma \) s.t. \( S = S' \overrightarrow{a} S'' \); then Eq. (19) follows straight from the definition of \( v \). On the other hand, if \( aS \in \Sigma \) s.t. \( S = S' \overrightarrow{a} S'' \), we need to show that equalities (19) hold for any arbitrary split of \( S \). Let \( S = S_1 \parallel S_2 \parallel S_3 \). Then, we have the following cases for the location of \( a \) and \( \bar{a} \):

\[
S_1 = S'_1 \overrightarrow{a} S''_1, \quad S_2 = S'_2, \quad S_3 = S''_2.
\]

\[
S_1 = S'_1 \overrightarrow{a} S''_1, \quad S_2 = S'_2, \quad S_3 = S''_3.
\]

\[
S_1, \quad S_2 = S'_2 \overrightarrow{a} S''_3, \quad S_3 = S''_3.
\]

Here, we address cases (20) (both \( a \) and \( \bar{a} \) are contained in a single partition) and (21) \((a \ and \ \bar{a} \ are \ split \ into \ different \ partitions\) as the other ones can be handled similarly to one of these two cases.

- **Case (20).** In this case, both \( a \) and \( \bar{a} \) are contained in \( S_1 = S'_1 \overrightarrow{a} S''_1 \); thus

\[
v(S'_1 \overrightarrow{a} S''_1 \parallel S_2 \parallel S_3) = v(S'_1 \overrightarrow{a} S''_1 \parallel S_2 \parallel S_3).
\]

Here, \( \bar{\zeta} = A \) if \( \overrightarrow{a} \bar{a} \) is recognized by \( RE_A \) or is equal to \( E \) otherwise. Given this definition of \( \zeta \), we have

\[
v(S'_1 \overrightarrow{a} S''_1 \parallel S_2 \parallel S_3) = v(S'_1 \overrightarrow{a} S''_1 \parallel S_2 \parallel S_3).
\]

Based on the above, we can state that, for Case (20), Eq. (19) and hence the associativity property holds.

- **Case (21).** In this case, \( a \) and \( \bar{a} \) are split into consecutive partitions. Thus,

\[
v(S'_1 \overrightarrow{a} S''_1 \parallel S'_2 \overrightarrow{a} S''_2 \parallel S_3) = v(S'_1 \overrightarrow{a} S''_1 \parallel S'_2 \overrightarrow{a} S''_2 \parallel S_3).
\]

where \( \bar{\zeta} = A \) if \( \overrightarrow{a} \bar{a} \) is recognized by \( RE_A \) or is equal to \( E \) otherwise. Given this definition of \( \zeta \), we also have

\[
v(S'_1 \overrightarrow{a} S''_1 \parallel S'_2 \overrightarrow{a} S''_2 \parallel S_3) = v(S'_1 \overrightarrow{a} S''_1 \parallel S'_2 \overrightarrow{a} S''_2 \parallel S_3).
\]

To complete the proof of Eq. (19), we need to show that

\[
v(S'_1 \overrightarrow{a} S''_1 \parallel S'_2 \overrightarrow{a} S''_2 \parallel S''_3) \text{ is also equivalent to } v(S'_1 \overrightarrow{a} S''_1 \parallel S'_2 \overrightarrow{a} S''_2).
\]

For this, we need the following lemma (see the Appendix for the proof):

**Lemma 2.2.** \( v(S_1 \overrightarrow{a} S_2) = S_1 \overrightarrow{a} V(S_2), \quad S_1 \in \Sigma^* \)

Intuitively, this lemma states that the grammar validation function can simply ignore the unmatched end-tags and unprocessed

non-terminal symbols in the prefix of its input. Given this lemma, we have

\[
v(S'_1 \overrightarrow{a} S''_1 \parallel S'_2 \overrightarrow{a} S''_2 \parallel S''_3) = v(S'_1 \overrightarrow{a} S''_1 \parallel S'_2 \overrightarrow{a} S''_2 \parallel S''_3).
\]

In summary, Theorem 2.2 states that the property of associativity, fundamental for autonomous and non-wasteful segmented computation holds for grammar validation.

### 2.3.3. Message filtering

Another document processing task commonly done at the firewalls is the filtering of the messages based on some registered criteria. Filtering involves identification of those messages which satisfy a given path expression; then depending on the registered action, matching message can be forwarded for further processing or may be eliminated from further consideration. Unlike header-based filtering, message filtering usually requires the investigation of the structural relationships between the tokens to identify if they satisfy a given path expression specified using XPath [44] or XQuery [45] languages.

Existing path expression-based message filtering schemes include YFilter [20], TurboXPath [25], FIST [27], AFilter [8], and FMware [7]. These schemes usually rely on an NFA- or PDA-based representation of the filter statements and require processing in a single node, such as a firewall or a specialized message queue handler.

We now show that the filtering task can be described as composition of two prefix computations. For the sake of simplicity and clarity of the presentation, we focus on a subset of the XPath composed of expressions of type \( \langle \langle / \rangle \rangle \). 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). In addition, we focus on the special case where the data and the XPath expression are not recursive; in other words each tag can occur once and only once along a given path. Note that extending this to the case with recursion is possible by adding step IDs, but we omit the details for the sake of clarity of the presentation.

We use the following representation of path expressions: a path expression is represented as \( PE(N, T, R, P) \), where \( N \) is the set of non-terminals, \( T = \Sigma \cup \overline{\Sigma} \) 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.

**Example 2.2.** Let us be given a query of the form \( \langle A/B/C/\star\rangle/D\rangle \). For the sake of convenience we will uniquely identify each “**“ wildcard symbol. Since, this query has only one “**“, we will rewrite it as \( \langle A/B/C/\star\rangle/D\rangle \).

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

- \( A \mapsto aS_1S_2\bar{a} \); where strings \( S_1 \) and \( S_2 \) are well-formed strings corresponding to subtrees.
- \( B \mapsto bS_3S_4\bar{b} \); where strings \( S_1 \) and \( S_4 \) are well-formed and these strings \( S_3 \) and \( S_4 \) are well-formed.
- \( C \mapsto cS_5W_1S_6\bar{c} \); where strings \( S_3 \) and \( S_4 \) are well-formed.
- \( W_1 \mapsto \overline{\Sigma}S_1S_2\bar{b} \); where strings \( S_1 \) and \( S_2 \) are well-formed.
- \( D \mapsto dS_7\bar{d} \); where the \( S_1 \) is a well-formed string corresponding to a (possibly empty) subtree.

Here, “\( \bar{\phi} \)“ represents an arbitrary symbol.
Based on the above example, we can define a filtering function as follows:

**Definition 2.7 (Filtering Function).** Formally, we can obtain the filtering validation function $f$ over string $S$ as follows:

$$f(S) = \begin{cases} f(S'S'') & \text{if } \exists a, \bar{a} \in T; B \in N \text{ s.t. } S = S'\bar{a}BS'bS'\bar{a}S'' \text{ and } w(S_a) = w(S_b) = \bot \text{ and } A/B \text{ is a query step in the filter statement} \\ f(S'S'') & \text{if } \exists a, \bar{a} \in T; B \in N \text{ s.t. } S = S'\bar{a}BS'bS'\bar{a}S'' \text{ and } w(S_a) = w(S_b) = \bot \text{ and } A/B \text{ is a query step in the filter statement} \\ f(S'W,S'') & \text{if } \exists \bar{a}, \bar{a} \notin T; B \in N \text{ s.t. } S = S'\bar{a}BS'bS'\bar{a}S'' \text{ and } w(S_a) = w(S_b) = \bot \text{ and } *_1/B \text{ is a query step in the filter statement} \\ f(S'W,S'') & \text{if } \exists \bar{a}, \bar{a} \notin T; B \in N \text{ s.t. } S = S'\bar{a}BS'bS'\bar{a}S'' \text{ and } w(S_a) = w(S_b) = \bot \text{ and } *_1/B \text{ is a query step in the filter statement otherwise.} \\ 0 & \text{if } S \neq \epsilon \text{ or } A/B \text{ is not a query step in the filter statement} \end{cases}$$

**Theorem 2.3 (Associativity of the Filtering Function).** For an arbitrary string $S \in T^*$, $T = \Sigma$, and for any arbitrary string partition $S = S_1 \parallel S_2 \parallel S_3$, we have

$$f(S_1 \parallel S_2) \in (N \cup T \cup E)^*,$$  
$$f(f(S_1 \parallel S_2) \parallel S_3) = f(S_1 \parallel f(S_2 \parallel S_3)) = f(S_1 \parallel S_2 \parallel S_3). \quad (28)$$

Theorem 2.3 holds. Let $S \neq \epsilon$ be such that none of the recursive clauses in the definition of $f$ is applied; then Eq. (29) follows straight from the definition of $f$. On the other hand, for an arbitrary string partition $S = S_1 \parallel S_2 \parallel S_3$, we have

$$f(S_1 \parallel S_2) = f(S_1 \parallel f(S_2 \parallel S_3)) = f(S_1 \parallel S_2 \parallel S_3) \quad (29).$$

**Proof of Theorem 2.3.** The base case, where $|S| = 0$, required for induction follows from the definition of $f$:

$$f(f \epsilon \parallel \epsilon) = f(f \epsilon \parallel \epsilon) = f(\epsilon \parallel \epsilon \parallel \epsilon) = \epsilon.$$

Next, let us consider non-empty strings. Our inductive hypothesis (IH) is that for $|S| \leq M$, Eq. (29) holds. Let $S \neq \epsilon$ be such that none of the recursive clauses in the definition of $f$ is applied; then Eq. (29) follows straight from the definition of $f$. On the other hand, for an arbitrary string partition $S = S_1 \parallel S_2 \parallel S_3$, we have

$$f(S_1 \parallel S_2) \in (N \cup T \cup E)^*,$$  
$$f(S_1 \parallel f(S_2 \parallel S_3)) = f(S_1 \parallel S_2 \parallel S_3) \quad (29).$$

**Proof of Theorem 2.3.** The base case, where $|S| = 0$, required for induction follows from the definition of $f$:

$$f(f \epsilon \parallel \epsilon) = f(f \epsilon \parallel \epsilon) = f(\epsilon \parallel \epsilon \parallel \epsilon) = \epsilon.$$

Next, let us consider non-empty strings. Our inductive hypothesis (IH) is that for $|S| \leq M$, Eq. (29) holds. Let $S \neq \epsilon$ be such that none of the recursive clauses in the definition of $f$ is applied; then Eq. (29) follows straight from the definition of $f$. On the other hand, for an arbitrary string partition $S = S_1 \parallel S_2 \parallel S_3$, we have

$$f(S_1 \parallel S_2) \in (N \cup T \cup E)^*,$$  
$$f(S_1 \parallel f(S_2 \parallel S_3)) = f(S_1 \parallel S_2 \parallel S_3) \quad (29).$$

The remaining cases can be handled similarly to the above ones.

**Case (30).** In this case, both $a$ and $\bar{a}$ are contained in $S_1 = S'\bar{a}BS'bS'\bar{a}S''$; thus

$$f(S'\bar{a}BS'bS'\bar{a}S'' \parallel f(S_2 \parallel S_3)) \quad (30).$$

Here, $\zeta = A$ for clause one and two, and $\xi = W_1$ for clause 4. Given this definition of $\zeta$, we have

$$f(S_1 \parallel S_2 \parallel S_3) = f(S_1 \parallel S_2 \parallel S_3) \quad (30).$$

For the opposite direction, we have:

$$f(f(S_1 \parallel S_2 \parallel S_3) \parallel S_3) = f(S_1 \parallel S_2 \parallel S_3) \quad (30).$$

Based on the above, we can state that, for Case (30), Eq. (29) and hence the associativity property holds.

**Case (31).** In this case, $a$ and $\bar{a}$ are split into consecutive partitions, right at a non-terminal symbol $B$ position. Hence $S'\bar{a}BS'bS'\bar{a}S'' \parallel f(S_2 \parallel S_3)$.

In the opposite direction, we need the following lemmas (see the Appendix for the proofs):

**Lemma 2.3 (Non-Terminal Left Pushout).** $f(BS'bS'\bar{a}S'') = Bf(S'bS'\bar{a}S'')$.

**Lemma 2.4 (Unmatched Barred Terminal Pushout).** $f(S'' \parallel S_3) = f(S'' \parallel S_3)$.

Hence, for the opposite direction, we have:

$$f(S'\bar{a}BS'bS'\bar{a}S'' \parallel f(S_2 \parallel S_3)) = f(S'\bar{a}BS'bS'\bar{a}S'' \parallel f(S_2 \parallel S_3)) \quad (30).$$

**Case (32).** In this case, $a$ and $\bar{a}$ are split into consecutive partitions, with partition boundary splitting $S_2$ into separate segments. Thus, for strings of the form $S = S'\bar{a}BS'bS'\bar{a}S''$, we have

$$f(S'\bar{a}BS'bS'\bar{a}S'' \parallel f(S_2 \parallel S_3)) = f(S'\bar{a}BS'bS'\bar{a}S'' \parallel f(S_2 \parallel S_3)) \quad (30).$$

Strings of the form $S = S'\bar{a}BS'bS'\bar{a}S''$, applicable to the other filter definition clauses, do not yield any new lemmas except **Lemma 2.5.**

**Lemma 2.5 (Unmatched Un-Barred Terminal Pushout).** $f(S'\bar{a}S'') = f(S' \parallel a \parallel f(S'' \parallel S_3))$.

In this section, we argued that three fundamental XML document processing operations, well-formedness checking, grammar validation, and filtering satisfy the associativity property, and thus are prefix functions. In the next section, we discuss how this knowledge is used by PASS to enable non-blocking, non-wasteful, and autonomous processing of these tasks.

### 3. PASS middleware

Prefix Automata Sytem (PASS) is a lightweight middleware for distributed processing of the XML payloads of SOAP messages. PASS constitutes a set of PASS-nodes interconnected by network links that process and relay segments of SOAP messages between nodes. The primary requirement for the PASS middleware architecture is the capability of an intermediate PASS-node $n_i$ to be able to perform XML document processing on behalf of the consumer $c$ in a way that once done at node $n_i$, the process is never repeated at further intermediate nodes the message travels towards the consumer. In addition, we want PASS-nodes to be autonomous and we do not want any PASS-node to be blocked
by any XML message; i.e., any PASS-node can choose if and what part of the document to process and when to stop the processing. The left-over processing is completed at a downstream PASS-node or (in the worst-case) by the destination (i.e., message recipient). Therefore, PASS establishes a cooperative chain of XML document processing between Web Service client and server.

PASS nodes avoid reading processed messages (or already-processed parts of the messages that are partially processed) using a small directory attached to the segments. This is important, in that, PASS nodes not only avoid document re-processing, but also segment scanning task whenever possible. Note also that a document does not have to be all in the memory to be processed. PASS middleware operates at a segment (or whenever they are available together at a sequence of segments) level.

### 3.1. SOAP messages and segments

SOAP messages \[40\] come in two styles, documents and remote procedure calls (RPC). Remote procedure call style messages have a short SOAP body, hence can usually be processed by a single PASS-node. Document-style SOAP messages, on the contrary, may carry large documents, such as purchase orders, and the like, which are amenable to distributed autonomous processing. A SOAP message \(M = \{h, x\}\), with a header \(h\) and XML document \(x\) attached to it, is produced at the sender end-point, and relayed through the network from SOAP message producer to consumer. 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. For instance, segmentation and reassembly may take place at the SOAP message/application layer. Alternatively, segmentation and reassembly may take place at the transport layer, for instance at the HTTP protocol. Note that HTTP/1.1 allows for a chunked transfer-coding, which may be used for segmentation/reassembly purposes. Fig. 3(a) illustrates a generic segmentation/reassembly feature.

### 3.2. Registering a service to PASS

Let us consider a service provider which would like to offload an XML document processing task to PASS. This service provider registers a set of URLs to PASS by specifying the originators of the XML documents to be processed by the provider, as well as the destination URLs that the service provider responds to. PASS assigns a unique providerID to this service. The PASS-nodes then actively listen for SOAP messages whose headers contain (from, to) URLs that match the URLs registered. When a PASS node receives one (or more message/segments) corresponding to the same XML document and (b) if it autonomously chooses to operate on this XML document, the node performs the processing tasks registered for that URL set on the behalf of the service provider.

Depending on the type and amount of intermediate processing delegated, PASS-nodes may effectively function as redundant servers of a given provider. The outsourcing of an XML document processing task may involve an authorization protocol between the SOAP message consumer and intermediate PASS-nodes so that Web service providers give explicit permission to PASS-nodes to execute specific processing tasks on behalf of the server. The security implications of PASS and mechanisms for implementation are discussed in Section 5.2.

### 3.3. PASS-nodes

PASS-nodes are application aware network nodes that perform both routing and basic XML document processing tasks on behalf of the recipient (Fig. 3(b)):

(i) 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.

(ii) For a consecutive sequence of segments, a PASS-node performs the relevant processing (well-formedness checking, grammar validation, filtering, etc.). The part of the document just processed is pushed back into the network, with appropriate flag directory information included in the segment data to inform other, downstream PASS-nodes, that part of the document has been processed.

(iii) 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. This allows easy reassembly of segments into larger segments and into the entire XML document, as well.

Notice that a PASS-node may process segments belonging to various XML documents simultaneously (Fig. 3(b)). In order to prevent wasting of resources

- **context switch is fast:** For each registered task, all the necessary internal structures (such as the finite state machines to verify the production rules of DTDs for grammar validation) are pre-generated and hashed against the document type (part of the unique identifier of the XML document). The amount of pre-processing of the payload needed before starting the processing (for instance, sequencing segments and tokenizing the XML payload before the state transitions of the finite state machine) are minimal.

- **the decision to whether stop processing a document is instantaneous:** The decision to halt a processing task is based on instantaneously verifiable criteria, such as the (a) local PASS-node resources, (b) based on document priorities, or (c) based on timeouts of specific timers.
3.4. PASS and well-formedness checking

Here, we describe how PASS leverages the constructive proof of the well-formedness Theorem of Section 2.3 to implement the well-formedness checking function on SOAP messages.

3.4.1. Flag directory for segmented well-formedness checking

An XML document segment contains both XML element start and end tags and other character strings. Well-formedness checking function, on the other hand, has to consider only the start and end tags in the document. Given an XML document segment, s, let us denote the tag-string composed only by the start tags (Σ) and end tags (Σ) in s with tag_str(s). Note that tag_str(s) ∈ (Σ ∪ Σ)*.

Given any such segment, PASS associates a flag directory, flag_dir(s) ∈ [0, 1, 2]**tag_str(s)** to it. 0 denotes an unverified tag, 1 denotes a verified tag, and 2 indicates a tag with a mismatch. The initial segments of a SOAP message injected into the network by the sender do not contain flags. When a PASS node receives a message containing a payload without any flag directory, it considers the corresponding payload, s, as if it had a flag directory composed of only 0s.

\[ \text{flag_dir}(s) = \underbrace{00 \ldots 0}_{|\text{tag_str}(s)|\text{many}0s}. \]

The payload of any PASS-processed message is modified such that each tag in the XML fragment is annotated with the corresponding flag directory entry.

**Example 3.1 (Flag Directory).** Let us consider an XML document segment, s = " <\Name><First>X. Selcuk</First><Last>Dandan</Last></Name> ". The corresponding (initial) annotated XML document segment is as follows:

\[
\begin{align*}
&\text{<Name 0>} <\text{First 0} X. \text{Selcuk} \text{/First 0}> <\text{Last 0} Dandan \text{/Last 0}> \text{/Name 0}> \\
&\text{The zeros here indicate that the tags in the document fragment have not been processed yet. A completely processed and verified XML fragment would look as follows:} \\
&\text{<Name 1>} <\text{First 1} X. \text{Selcuk} \text{/First 1}> <\text{Last 1} Dandan \text{/Last 1}> \text{/Name 1}>.
\end{align*}
\]

Note that these flag directory entries will be removed before the XML payload is passed to the application layer at the receiver.

Clearly there can be more efficient annotation and directory management schemes. In fact, the flag directory implementation is to be such that the directory associated with a segment can be read before the actual XML tokens are parsed. Our goal in the above example is mainly to present a feasible annotation scheme, not the most efficient one.

3.4.2. Well-formedness checking

Let us assume that a PASS-node receives a set of consecutive segments corresponding to an XML document addressed to a registered Web service provider. Let s be the string corresponding to the XML document segment contained within this set of consecutive segments and let flag_dir(s) be the (combined) flag directory corresponding to the tag-string tag_str(s). A PASS-node implements the well-formedness checking function on tag_str(s), by modifying flag_dir(s) into flag_dir′(s), as in Fig. 4, relying on the constructive proof of Section 2.3.

**Example 3.2 (Well-Formedness Checking).** Let us consider an XML document segment, s = "<a><b><c><d><e><f><g><h><i><j></i></h></g></f></e></d></c></b></a> ". We represent the corresponding tag string as, tag_str(s) = "ababacdefg". Let us also assume that the corresponding flag directory is flag_dir(s) = "11110000000". I.e., the first four tags have already been verified by another upstream PASS-node. For this example, we combine the flag_str(s) and the flag_dir(s) as "a1b1c1d1e1f1g1h1i1j1k1l1m1n1o1p1q1r1s1t1u1v1w1x1y1z1". The algorithm of Fig. 4 processes this segment as follows.

- The first four entries"a1b1c1d1" are ignored as the corresponding flags are all 1 (Step (ii)).
- The next entry "e1" is also modified, as it is an unprocessed end tag (Step (iii)).
- "c1" causes (6) to be pushed into the stack (Step (b)).
- "d1" causes (7) to be pushed into the stack (Step (b)).
- "d1" requires the top element of the stack to be popped. In this case, the top element is (d, 7), which matches "d1". Therefore, the corresponding flag_dir entries are modified to 1; i.e., "d1" and "d1" (Step (c)).
- "c0" causes the top element of the stack to be popped. In this case, the top element is (c, 6), which matches "c0". Therefore, the corresponding flag_dir entries are modified to 1; i.e., "c1" and "c1" (Step (c)).
- "g0" and f0 cause (g, 10) and (j, 11) to be pushed into the stack (Step (b)).

Since the segment ends at this point, the PASS node stops processing. The final flag directory (which will be encoded along with the outgoing segment) is flag_dir(s) = "1111011110010100".

Naturally, various optimizations on the algorithm depicted in Fig. 4 are possible. For example, if a long enough stretch of the string s is processed and verified, a PASS-node may push that stretch into the network while still processing the remaining parts of s. Similarly, if a new segment arrives while s is being processed, the payload of the new segment may simply be attached at the end of s. This flexibility is enabled by the prefix nature (i.e., associativity) of the well-formedness checking function.

3.5. PASS and grammar validation

Let us consider a service provider which wishes to ensure that the XML messages that arrive to it conform to a given DTD. This service registers a set of URLs to the PASS middleware along with the corresponding DTD specification. When a PASS-node receives one (or more) segments corresponding to the same XML document, it initiates a grammar validation function on the payloads of these segments. Next we discuss how PASS-nodes implement the grammar validation function, relying on the constructive proof of Section 2.3.

3.5.1. Flag directory for grammar validation

Given any XML document segment, s, let us denote the tag-string composed only by the start (Σ) and end tags (Σ) in s with tag_str(s). Given any such segment, PASS associates a tag directory, flag_dir(s) ∈ [0, 1, 2]**tag_str(s)** to it:

- 0 denotes an unverified tag.
- 1 indicates a fully-verified tag.
- 2 indicates a partially-verified tag, and
- 3 indicates a tag with mismatch.

3.5.2. Segment processing for grammar validation

Let s be the string corresponding to the XML document segment contained within a set of consecutive segments. Let flag_dir(s) be the (combined) flag directory corresponding to the tag-string tag_str(s). A PASS-node implements the grammar validation function on tag_str(s), by modifying flag_dir(s) into flag_dir′(s), as in Fig. 5.

3.6. PASS and filtering

The filtering service registers a set of URLs to the PASS middleware along with a corresponding filter specification. When a PASS-node receives one (or more) segments corresponding to the same XML document, it initiates a filtering function on the payloads of these segments. PASS-nodes implement the filtering function, relying on the proof in Section 2.3. The implementation of the filtering function is structurally similar to that of the grammar validation function.
Given a string $s$ and its flag directory $\text{flag.dir}(s)$

(i) $\text{flag.dir}'(s) = \text{flag.dir}(s)$

(ii) Relying on the associativity of well-formedness checking, a PASS-node ignores all processed tags at the beginning of $\text{tag.str}(s)$. In other words,

$$\left( \forall j \in [0, i] \text{flag.dir}(s)[j] \neq 0 \right) \rightarrow \text{flag.dir}'(s)[i] = \text{flag.dir}(s)[i]$$

(iii) Let $k$ be the index of the first tag, which is not processed. The PASS-node ignores all unprocessed end tags at the beginning of $\text{tag.str}(s)$. In other words,

$$\left( \forall j \in [0, i] (\text{flag.dir}(s)[j] = 0) \wedge (\text{tag.str}(s)[j] \in \Sigma) \right) \rightarrow \text{flag.dir}'(s)[i] = 0$$

(iv) Let $k'$ be the index of the first unprocessed start tag. A PASS-node initiates a well-formedness checking pushdown automaton and starts processing $\text{tag.str}(s)$ at index $k'$. The PASS-node iterates from $i = k'$ to $i = |\text{tag.str}(s)|$:

(a) if $\text{flag.dir}(s) = 1$ or $\text{flag.dir}(s) = 2$, then by the prefix function nature of the well-formedness check, such processed tags are ignored; i.e.,

$$\text{flag.dir}(s)[i] \neq 0 \rightarrow \text{flag.dir}'(s)[i] = \text{flag.dir}(s)[i]$$

(b) if $\text{tag.str}(s)[i] \in \Sigma$ and $\text{flag.dir}(s)[i] = 0$ (i.e., for each encountered unprocessed start tag) a pair of the form $(i, \text{tag.str}(s)[i])$ is pushed into the PDA stack.

(c) $\text{tag.str}(s)[i] \in \Sigma$ and $\text{flag.dir}(s)[i] = 0$ (i.e., for each encountered unprocessed end tag), the top of the stack is checked. Let the pair at the top of the stack be $(i, \alpha)$. If $\alpha$ is the start tag corresponding to end tag $\text{tag.str}(s)[i]$, then the match is correct; i.e., $\text{flag.dir}'(s)[j] = 1$ and $\text{flag.dir}(s)[i] = 1$.

(ii) otherwise, there is a mismatch and $\text{flag.dir}(s)[j] = 2$.

A PASS-node can stop this process at any time before or when the index counter reaches $|\text{tag.str}(s)|$. When it stops processing,

(i) The PASS-node re-segments the string $s$ and $\text{flag.dir}(s)$, and for each segment, $s_h$, it creates a payload annotated with the corresponding directory entries pairs.

(ii) These segments are then pushed back into the network.

3.6.1. Flag directory for filtering

Given any XML document segment, $s$, let us denote the tag-string composed only by the start ($\Sigma$) and end tags ($\bar{\Sigma}$) in $s$ with $\text{tag.str}(s)$. Given any such segment, PASS associates a tag directory, $\text{flag.dir}(s) \in \{0, 1, 2, 3\}^{\text{tag.str}(s)}$, to it:

- 0 denotes an unverified tag,
- 1 indicates a filter-verified tag,
- 2 indicates a processed-but-not-filter-verified tag, and
- 3 indicates a failure.

3.6.2. Segment processing for filtering

Given a string $s$ and a $\text{flag.dir}(s)$ corresponding to $\text{tag.str}(s)$, a PASS-node implements the filtering function on $\text{tag.str}(s)$, by modifying $\text{flag.dir}(s)$ into $\text{flag.dir}'(s)$, as in Fig. 6.

4. Distributed XML processing evaluation

In this section, we study the performance of the proposed PASS middleware via event driven simulations. We abstract the processing functions performed, so as to avoid conclusions that are dependent on specific processing functions:

Clients/Message senders. A client is abstracted by an element that generates XML documents of various sizes at specific time epochs. Document sizes are exponentially distributed with mean $\lambda$ bytes. Document gaps are also exponentially distributed, with mean $\mu$. Hence, we may define document load $DL = \lambda/(\lambda + \mu)$. Moreover, a document is partitioned into fixed size segments and injected into the PASS network domain, where PASS nodes are used to both relay and locally process segments on their way to a server.

Servers/Message receivers. A server is an element that receives processed or non-processed XML segments, processing those that have not yet been processed by the PASS network domain. Segment reception rate is limited to the byte/sec speed of the server network interface connecting itself to a PASS-node delivering the segments to the server.

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. Each interface is equipped with a work-conserving weighted round-robin port scheduler to regulate port transmission of segments. For instance, if three XML streams

---

4 Note that there are different ways a PASS-node can behave when a mismatch is observed: (a) as stated above, the mismatching end tag can be marked with 2, (b) the directory for the entire segment can be marked with 2, or (c) the segment can simply be dropped (together with all segments belonging to the document). The mismatch-management policy may be determined by the service provider at the registration time.
Given a string $s$, its flag directory $\text{flag.dir}(s)$, and a DTD

(i) $\text{flag.dir}'(s) = \text{flag.dir}(s)$

(ii) Relying on the associativity of the grammar validation function, the PASS-node ignores all processed tags (with $\text{flag.dir}'(s)$ value, 1, 2 or 3) at the head of $\text{tag.str}(s)$.

(iii) Let $k$ be the index of the first tag which is not processed. By the proof of Theorem 2.2, a PASS-node ignores all unprocessed end tags at the head of $\text{tag.str}(s)$.

(iv) Let $k'$ be the index of the first unprocessed start tag. The PASS-node starts processing $\text{tag.str}(s)$ at index $k'$. The PASS-node iterates from $i = k'$ to $i = \lfloor \text{tag.str}(s) \rfloor$ as follows:

(a) if $\text{flag.dir}'(s) = 1, 2$ or $3$, then by the prefix function nature of the grammar validation, such processed tags are ignored.

(b) if $\text{tag.str}(s)[i] \in \Sigma$ and $\text{flag.dir}'(s)[i] = 0$ (i.e., for each encountered unprocessed start tag) a pair of the form $(i, \text{tag.str}(s)[i])$ is pushed into the PDA stack.

(c) $\text{tag.str}(s)[i] \in \Sigma$ and $\text{flag.dir}'(s)[i] = 0$ (i.e., for each encountered unprocessed end tag), the top of the stack is checked. Let the pair at the top of the stack be $(\alpha, j)$.

   (i) if $\alpha$ is the start tag corresponding to end tag $\text{tag.str}(s)[i]$, then the tags are matching, but the string between these tags has to be verified against the production rules

   (ii) let $\text{tag.active}$ be the string composed of the active tags, $\text{tag}(s)[j]$, where $j + 1 \leq l \leq i - 1$ and $\text{flag.dir}'(s)[l] = 2$.

   (iii) verify the string $\text{tag.active}$ against the regular expression corresponding to the tag $\alpha$

   (iv) if $\text{tag.active} \in \text{RE}_\alpha$, then

   $\text{flag.dir}'(s)[j] = \text{flag.dir}'(s)[i] = 2$

   $\text{flag.dir}'(s)[j] = 2$

   $\text{flag.dir}'(s)[i] = 1$.

   (v) otherwise, there is a mismatch and $\text{flag.dir}'(s)[j] = 3$

A PASS-node may stop this process at any time before or when the index counter reaches $\lfloor \text{tag.str}(s) \rfloor$. When processing is halted, the PASS-node re-segments the string $s$ and $\text{flag.dir}'(s)$, and, for each segment, it creates a payload annotated with the corresponding directory entries. These segments are then pushed back into the network.

Fig. 5. Grammar validation by a PASS-node.

---

Given a string $s$, its flag directory, and a filter statement

// Below a “filter-tag” corresponds to a tag included in the filter statement. If the filter contains a "*" wildcard, then all symbols are filter-tags for that wildcard /*

(i) Relying on the associativity of the filtering function, the PASS-node ignores all processed tags.

(ii) Let $k$ be the index of the first filter-tag which is not processed. The PASS-node ignores all unprocessed end tags at the head of the string.

(iii) Let $k'$ be the index of the first unprocessed start filter-tag. The PASS-node starts processing the string at index $k'$:

(a) by the prefix function nature of the filtering, the node ignores all processed tags.

(b) an encountered unprocessed start filter-tag and this tag’s position in the string are pushed into the PDA stack.

(c) an encountered unprocessed end filter-tag cause the start symbol at top of the stack (and the associated index) being popped:

   (i) if the tags are matching, the string between these tags has to be verified against the production rule for the corresponding step in the filter statement per Definition 2.7.

   (ii) if the production rule (including the well-formedness requirements and the non-terminals) is verified, then the start and end tags are marked as filter-verified with the corresponding non-terminal

   (iii) otherwise, the filter step is not matched; the start and end tags are marked processed-but-not-filter-verified.

   (iv) otherwise, the string is not well-formed; the start and end tags are marked with a special symbol indicating failure.

A PASS-node may stop this process at any time. When processing is halted, the PASS-node re-segments the string and, for each segment, it creates a payload annotated with the corresponding directory entries. These segments are then pushed back into the network.

Fig. 6. Filtering by a PASS-node.
are being served by a given output interface, weights \( w_1 = 0.2, w_2 = 0.3, w_3 = 0.4 \) will reserve 20\% of the interface speed for stream 1, 30\% for stream 2, 40\% for stream 3, and the remaining 10\% will be shared equally by the three streams, provided that there are segments stored in their respective output queues. Work conserving feature ensures that, in the absence of segments of a given stream, its reserved bandwidth is shared equally among all active streams.

The segment processing speed is limited to a maximum processing capacity of \( m \) bytes/sec. A PASS-node processing scheduler decides which segments to process at any given time. Two scheduling strategies are implemented: \textit{shared processing}, where processing power is equally shared by all streams served by the node, and \textit{batch processing}, where segments of a given XML document are processed until there are no more segments from that document in the queue; then, a segment from a document of another XML stream is selected, and processed in the same manner. The shared processing ensures fairness across XML streams, although it is likely to spread document processing among many PASS-nodes, whereas the batch mode causes a given XML document to be processed by fewer PASS nodes.

For simplicity, all networking interfaces and segment processing units have the same speed across all network elements. In addition, 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

PASS middleware enables savings on two fronts: network transmission resources at intermediate nodes and processing resources at servers. In order to evaluate the PASS middleware architecture, we use the following performance measures:

- Number of segments received by the servers (\( \text{rec(server,)} \)) is a rough measure of network savings. It measures the amount of segments that has been discarded on the way in the network. A decrease in the number of segments received is also a measure of load reduction at the servers.
- Number of unprocessed segments received by the servers (\( \text{unpro\_rec(server,)} \)) is a rough measure of the processing savings at the servers. It measures the amount of segments that are left for the servers to process.
- Amount of work done at the switches (\( \text{work(switch,)} \)) 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. 7. 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 at any of the interfaces. Other parameters are: \textit{document average size} is 50 segments; network link delay is 6 segments. On a network of 1 Gbps link speed, these parameters represent a metro area network, with 20 Km separation between PASS nodes. The simulation was run for 20 000 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.

Note that, in order to observe the effect of PASS-nodes in improving good-put of the system (i.e., the amount of bad messages dropped by the system early on), we also simulated various segment error rates (e.g., probability of segments to contain errors). We experimented with error rates as high as 0.8; although in reality this is not likely (unless on a DoS attack scenario), this number serves to show the scalability of PASS to high error rates.

### 4.3. Results without composition

Tables 1–3 show the results, in a setup where the processing capacity of each PASS node is equal to the bandwidth 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. The entries in the table show that the numbers of unprocessed segments arriving to the servers are processed during the time taken to transmit one segment at any of the interfaces. Other parameters are: \textit{document average size} is 50 segments; network link delay is 6 segments. On a network of 1 Gbps link speed, these parameters represent a metro area network, with 20 Km separation between PASS nodes. The simulation was run for 20 000 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.

Note that, in order to observe the effect of PASS-nodes in improving good-put of the system (i.e., the amount of bad messages dropped by the system early on), we also simulated various segment error rates (e.g., probability of segments to contain errors). We experimented with error rates as high as 0.8; although in reality this is not likely (unless on a DoS attack scenario), this number serves to show the scalability of PASS to high error rates.

### 4.3. Results without composition

Tables 1–3 show the results, in a setup where the processing capacity of each PASS node is equal to the bandwidth 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. The entries in the table show that the numbers of unprocessed segments arriving to the servers are 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 drop significantly. In other words, the PASS distributed framework is able to capture and eliminate faulty XML documents. Shared

### Table 1

<table>
<thead>
<tr>
<th>Segments received [BATCH, 1×]</th>
<th>Unprocessed segments received [BATCH, 1×]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err. rate</td>
<td>Server 1</td>
</tr>
<tr>
<td>0.0</td>
<td>9933</td>
</tr>
<tr>
<td>0.2</td>
<td>6272</td>
</tr>
<tr>
<td>0.5</td>
<td>4060</td>
</tr>
<tr>
<td>0.8</td>
<td>1703</td>
</tr>
</tbody>
</table>

---

### Table 2

<table>
<thead>
<tr>
<th>Unprocessed segments received [SHARED, 1×]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err. rate</td>
</tr>
<tr>
<td>0.0</td>
</tr>
<tr>
<td>0.2</td>
</tr>
<tr>
<td>0.5</td>
</tr>
<tr>
<td>0.8</td>
</tr>
</tbody>
</table>

---

### Table 3

<table>
<thead>
<tr>
<th>Unprocessed segments received [SHARED, 1×]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err. rate</td>
</tr>
<tr>
<td>0.0</td>
</tr>
<tr>
<td>0.2</td>
</tr>
<tr>
<td>0.5</td>
</tr>
<tr>
<td>0.8</td>
</tr>
</tbody>
</table>

---

Fig. 7. Network scenario: Arrows denote the client server routes. Filled squares denote interfaces where segments are stored for processing and transmission.
scheduling enables PASS to process slightly more segments relative to the batch scheduling (especially for XML documents going through fewer nodes).

Table 2 provides more details regarding the amount of work done by the nodes. 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 PASS nodes (the numbers in parenthesis denote the numbers of segments destined for the corresponding servers).

<table>
<thead>
<tr>
<th>Unprocessed segments received [BATCH][Err. rate 0.0]</th>
<th>None</th>
<th>1×</th>
<th>2×</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proc. power</td>
<td>(9933)</td>
<td>(10 122)</td>
<td>(9983)</td>
</tr>
<tr>
<td>Server 1</td>
<td>0</td>
<td>544</td>
<td>0</td>
</tr>
<tr>
<td>Server 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Server 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

5. Discussions

5.1. Distributed autonomous systems and parallel computing

Parallel computation is a research field that intersects with autonomous distributed systems, and hence our autonomous prefix computation paradigm. In parallel computations, processing elements, which communicate via some communication infrastructure, are used to collectively process a computation task written in
some given language. The language itself typically includes annotations and directives that help the parallel processing of the task across the network of processors. Thus, it is instructive to contrast some given language. The language itself typically includes annotations and directives that help the parallel processing of the task across the network of processors. Thus, it is instructive to contrast the autonomous distributed system with the various elements of a parallel computing environment\footnote{An in depth evaluation of the autonomous prefix computation as a parallel computation infrastructure is outside the scope of this paper.} [39]:

- **Parallelism** refers to the computation of a specific part of a program by distinct data or distinct parts of a program by various processors (and possibly distinct data, as well). Segmented prefix computation can be seen as the execution of a single program on multiple (parts of) data.

- **Decomposition** refers to the partition of the computation task into sub-tasks, to be executed on a parallel infrastructure. Our prefix autonomous system replicates the processing code on various processing nodes, so that the XML data is partitioned on the fly (on demand). Hence we may argue that our data decomposition is done at “run-time”.

- **Mapping** refers to the placement of the parts resulting from the decomposition into specific processing elements. Placement is important, because it may affect the communication overhead between processing elements in need to interact. In our autonomous prefix computation model, the entire data is carried across the various processing nodes. This simple mapping avoids the problem of optimizing placement, and hence simplifying management, but at the expense of communication overhead. In our infrastructure, data has to move around participating nodes even if for each node, only a small fraction of the data is used for local computation. Note also that no fixed mapping is required in that any processing components that receive data can choose to work on it.

- **Communication** refers to the exchange of information/intermediate results between processing elements during the parallel computation. In general, processing elements need to know which other elements to send these results to, for achieving a coherent computation. In our autonomous prefix computation model, intermediate results do not need to be shipped to any specific node. The streaming nature of the data provides a data bus, from/to which intermediate input/output data is retrieved/placed. Therefore, any explicit communication need is avoided, reducing communication management cost.

- **Synchronization** relates to the need of processing elements executing in parallel to synchronize at a given point of their respective computations. Synchronization issues may cause processors to stop processing until some other processor produce a given intermediate result, needed for the stalled processor to proceed. In our autonomous prefix computation model, either each processing node has full capability to process an input segment, or, in case of composition, a given node does not process the part of the segment that needs prior processing by another node. However, in the later case, the processing node in question does not stall, avoiding synchronization issues altogether.

In this context, [30] is related to our distributed XML processing framework. Taking advantage of multi-processor hardware platforms, they propose parallel processing of large XML documents, by assigning nodes and sub-trees to threads, to be executed by various processors. Each thread has its own task queue, from which it pops tasks to be executed as part of a DOM XML document processing. Moreover, a thread becomes a thief when its own task queue becomes empty, which means it starts stealing tasks from other threads. By popping from the bottom of other threads’ task queues, whereas the queue thread owner pops its next task from the top of the queue, contention for the task queue structure is minimized. They illustrate their approach with a parallel XML serializer, where sub-trees are assigned to various threads, and intermediate results are later glued by traversing a tree of task stubs, pointing to intermediate results. Our prefix based approach to XML processing can be used to generate the algorithms used by each thread for arbitrary partition of the XML document into sub-trees to be processed in parallel.

### 5.2. PASS and security

Transport layer security protocols, such as SSL/TLS or network layer security protocols (IPSec) lay at the foundation of secure messaging in web services. These protocols enable client/server authentication, data integrity, and confidentiality. PASS itself can be protected against attacks using existing low-level security protocols. The server and PASS nodes need to establish “trusted” connections, similar to the web server/client pair. The establishment of the server to PASS node trusted sessions should follow the same paradigm as the original server/client establishment. Commonly, clients and servers have their communications encrypted by AES algorithm, with private/public keys. Once the server private key for the original server/client trusted session reaches the PASS nodes, they can decrypt the client SOAP messages exactly the same way as the server can. Under these conditions, security is no worse than the original Web server/client security.

At the higher levels, XML signature and XML encryption can be used within SOAP to provide message-level authentication and...
persistent confidentiality. IBM, Microsoft, and VeriSign proposed WS-Security specification to describe how to attach signatures and encryption headers to SOAP messages. Other high-level XML-based specification languages and protocols, being developed by IBM and Microsoft, include WS-authorization, WS-Policy, WS-Trust, WS-Privacy, WS-Secure conversation, and WS-Federation. We see that PASS can help with the delivery of these high-level security services, which require significant amount of XML management and filtering.

6. Conclusions and discussions

In this paper, we introduced the concept of distributed XML processing, by which XML documents are processed in piecemeal manner. We further 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. PASS nodes perform document processing tasks that can be cast as "prefix computations", distributively and autonomously. Prefix computations, with the associativity property, lend themselves to autonomous distributed computation architecture. The associativity property provides the autonomous, non-wasteful, non-blocking properties so important in a distributed computation environment of low management cost. Obviously, not every computation task may render itself to an implementation satisfying associativity. In this paper, we not only showed that common XML document processing tasks satisfy this property, but we also provided constructive proofs which act as blueprints for designing PASS processing algorithms for suitable outsourced tasks. We also presented event-driven simulation results, which illustrates how a PASS middleware can provide significant network and server resource savings, under varying traffic conditions. A natural extension of our work is to investigate generic automata networks that may be able to compute such tasks.

Appendix

Lemma 2.1. \( w(\tilde{a}s) = \tilde{a}w(S) \).

Proof of Lemma 2.1. By induction on the length of \( S \). The base case \( w(\tilde{a}) = \tilde{a} \) is trivially true. The inductive hypothesis is that \( w(\tilde{a}s) = \tilde{a}w(S) \) for \( |S| \leq M \). For \( |S| = M + 1 \), we have two cases: Case a \( \exists b \in \Sigma . t.s. S = Sb\tilde{w}s^x \). Then

\[
\begin{align*}
  w(\tilde{a}s) &= w(\tilde{a}s^x b^x) = w(S^x b^x) \\
  &= w(S^x) \Rightarrow \tilde{a}w(S^x) = \tilde{a}w(S) \Rightarrow \tilde{a}w(S) = \tilde{a}w(S) \\
  &= \tilde{a}w(S^x b^x) = \tilde{a}w(S). \\
\end{align*}
\]

Lemma 2.2. \( v(S_1aS_2) = S_1\tilde{a}v(S_2), S_1 \in N^* \).

Proof of Lemma 2.2. By induction on the length of \( S \). The base case \( v(\tilde{a}) = \tilde{a}v(\epsilon) \) is trivially true. The inductive hypothesis is that \( v(S_1aS_2) = S_1\tilde{a}v(S_2) \) for \( |S_2| \leq M \). For \( |S_2| = M + 1 \), we have two cases:

Case a \( \exists b \in \Sigma . t.s. S_2 = S_2b\tilde{w}s^x \). Then

\[
\begin{align*}
  v(S_1aS_2) &= v(S_1aS_2 b\tilde{w}s^x) = v(S_1aS_2) v(S_2) \\
  &= v(S_1aS_2) v(S_2) \\
  &= v(S_1aS_2) v(S_2). \\
\end{align*}
\]

Lemma 2.3 (Non-terminal Left Pushout). \( f(B_S\tilde{a}s^x) = Bf(S_0\tilde{a}s^x) \).

Proof of Lemma 2.3. This follows from the facts that there is no recursive clause in Definition 2.7 for a string with a non-terminal \( S, S' \in T \). Hence, the non-terminal "survives" any processing of the string it is attached to.

Lemma 2.4 (Unmatched Barred Terminal Pushout). \( f(S_0\tilde{a}s^x) = f(S) \parallel a \parallel f(S)^x \).

Lemma 2.5 (Unmatched Un-Barred Terminal Pushout). \( f(S'aS_0) = f(S') \parallel a \parallel f(S') \).

References


---

6 For \( S' \), \( \tilde{a} \) will never match with a right segment by definition of matching. For \( S'' \), we recall that \( w(S'') = \perp \).

7 For \( S' \), \( \tilde{a} \) will never match with a left segment by definition of matching. For \( S'' \), we recall that \( w(S') = \perp \).


Dirceu Cavendish received his Bachelor degree in electronics from Federal University of Pernambuco, Brazil, in 1986. He then spent five years as a development engineer at the Business Communication Division of Philips. He received his M.S. in Computer Science from Kyushu Institute of Technology, Japan, in 1994, and his Ph.D. from the Computer Science Department of the University of California, Los Angeles in 1998, working on congestion control and routing in QoS supportive high-speed networks. He was with NEC Laboratories America, Computer and Communications Research Laboratories, Princeton, New Jersey, and Cupertino, California, from 1998 to 2006. He is currently an Adjunct Professor at Kyushu Institute of Technology, Japan. His research interests include management of packet networks, broadcast systems, Web Services, and Distributed Computing.

K. Selçuk Candan is an associate professor at the Department of Computer Science and Engineering at the Arizona State University and a visiting research scientist at NEC Laboratories America. He joined the ASU in August 1997, after receiving his Ph.D. from the Computer Science Department at the University of Maryland at College Park. His research interests include XML, multimedia, and web data management,adaptive data processing and indexing for sensory data and distributed data management for efficient content delivery. He has published over 100 articles in respected journals and conferences in related areas. Most recently, he is an editorial board member of the Very Large Databases (VLDB) journal and the ACM SIGMOD Digital Symposium Collection (DiSC). He was also an organization committee member of the ACM SIGMOD Conference 2006 and a PC chair of ACM Multimedia 2008. He received his B.S. degree in computer science from Bilkent University in Turkey in 1993.