March 31, 2009

RT0851
Computer Science 18 pages

Research Report

Optimized XML/HTML parsing within single thread

Masayoshi Teraguchi, Sachiko Yoshihama

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Invention Disclosure Publication

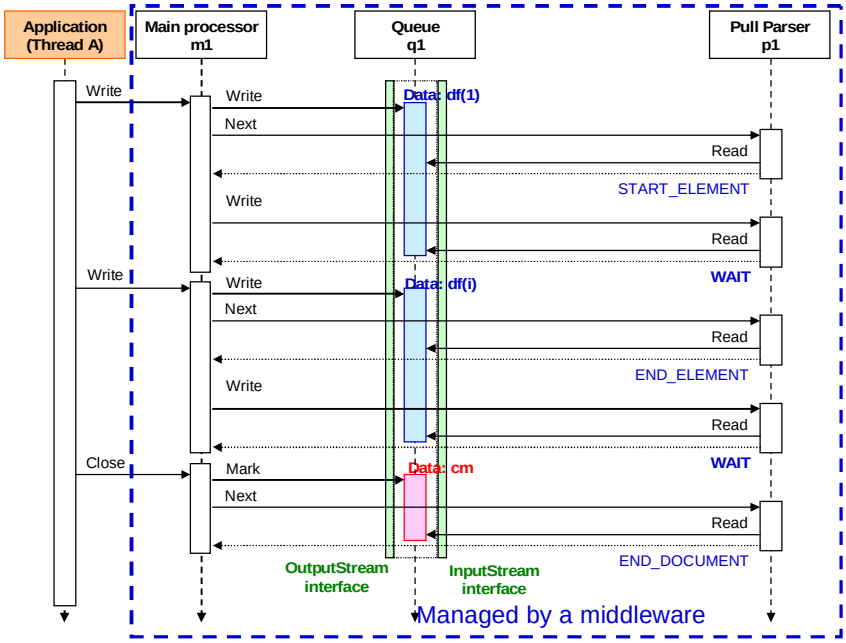
Title of disclosure)
Optimized XML/HTML parsing within single thread

Inventor(s)
Masayoshi Teraguchi (teraguti@jp.ibm.com), Sachiko Yoshihama (sachikoy@jp.ibm.com)

Overview of the idea

Disclosed is a method for parsing a XML/HTML document efficiently within a single thread in the middleware. It requires no modification of the existing Web application. It consists of the following three components:

1. A XML/HTML parser (p1) which implementes the extented version of standard API. The extended version of standard API includes an event which allows the parsing process to suspend when q1 becomes empty and an API which alloes the process to resume the parsing smoothly.
2. A queue (q1) which stores a fragment (df(i)) of the XML/HTML document.
3. A main processor (m1) which manages overall process, such as buffering of the XML/HTML fragment into q1 and invocation of p1.



When m1 receives a XML/HTML fragment df(i), m1 passes df(i) to q1 first and invokes p1 to parse df(i). p1 itself acts like the implementation of the standard API while p1 can read df(i) from q1. If there is no data in q1 (i.e. the last data in df(i) is read), p1 suspends the parsing process by storing any parser internal information completely with no regression and returns the event which lets m1 know the parsing process is suspended. If m1 receives the next XML/HTML fragment df(i+1) and passes it to q1, m1 calls p1 again to resume the parsing process from the first data of df(i+1). If m1 receives an EOF signal, which tells the end of the XML/HTML document, m1 passes the signal to q1 so that p1 can know the end of the XML/HTML document through q1 and finish the parsing process.

p1 comforms to the XML/HTML standards and uses an automaton to manage internal fine-grained states of the parsing process so that p1 can suspend and resume the parsing process under all conditions.

q1 is implemented as an output stream in order to receive a XML/HTML fragment df(i) by m1. When m1 passes df(i) to q1 by a write method of the output stream, q1 stores df(i) in an internal queue and set a data pointer to the first data of df(i). When m1 tells an EOF signal to q1 by a close method of the output stream, q1 stores the signal internally.

q1 is also implemented as an input stream. When p1 requires data in the queue by a read method of the input stream, q1 does one of the followings:
(a) If the EOF signal is stored in q1, q1 returns -1 in order to let p1 know the end of the XML/HTML document.
(b) If the data pointer reaches the end of df(i), q1 discards df(i) returs 0 in order to let p1 know that there is no data in the queue.
(c) If q1 has remaining data in its queue, q1 loads the data from the queue, moves the data pointer according to the data length, and returns the data and its length to p1.

The method disclosed here doesn't require buffering of whole XML/HTML document before the parsing process starts even when a program runs within single thread. Therefore, it allows the program to parse very huge XML/HTML documents with no concern about memory usage. It will also minimize performance overhead because of the mechanism to suspend and resume the parsing process efficiently.

Main Idea (in Japanese)

1. Background

近年マルチコアを搭載した CPU が主流になってきている. 現在は 2 つ (DualCore) や 4 つ (QuadCore) といったものが主流であるが, 近い将来にもっと多くのコアを搭載した CPU がでてくることが分かっている. このように CPU のマルチコア化が進むにつれて, マルチコア CPU に適した処理の高速化/効率化を考えることが必要不可欠である.

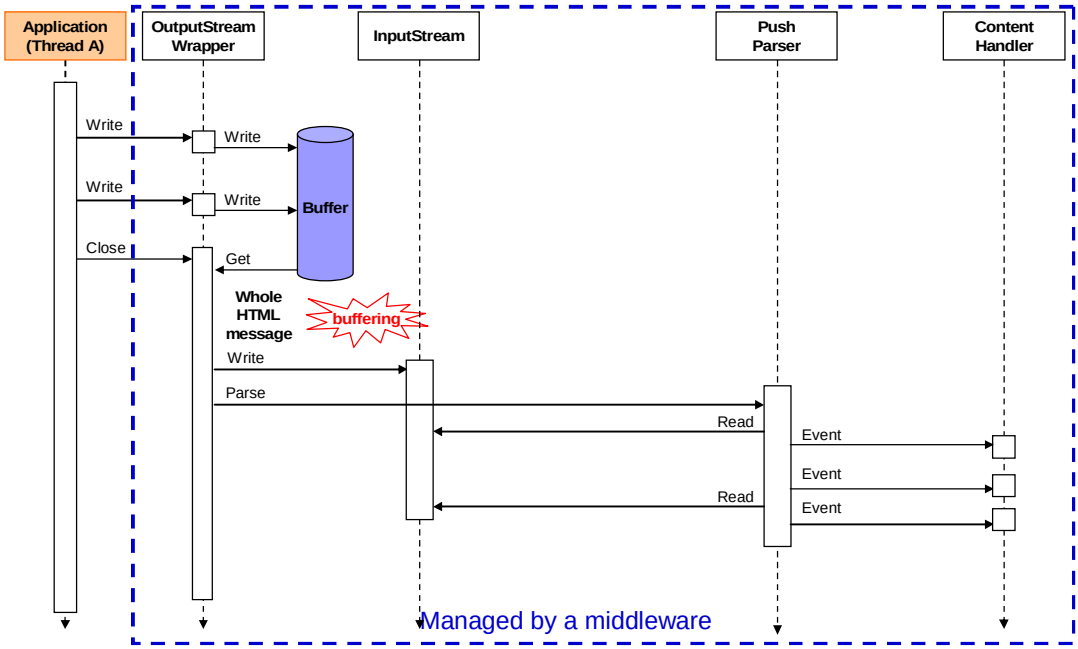
Web アプリケーションサーバにおいて, 一つのリクエストの処理を高速化/効率化する技術の一つとして複数スレッドモデルがある. 複数スレッドモデルとは, パイプライン化等により複数スレッドを利用することで高速化/効率化する技術であり, 現在の計算機環境では一般的に利用されているが, 共有メモリへのアクセスには同期を必要とする, 一つのスレッドがボトルネックになった場合に性能が劣化する, といった問題がある. 複数スレッドモデルをマルチコア CPU に応用して, 各スレッドを各コアに割り当てることで処理を高速化/効率化することも考えられるが, 以下に示すような問題が生じることにより CPU 全体で見たときの処理能力がそれほど向上しない, といった現象が起こりうる.

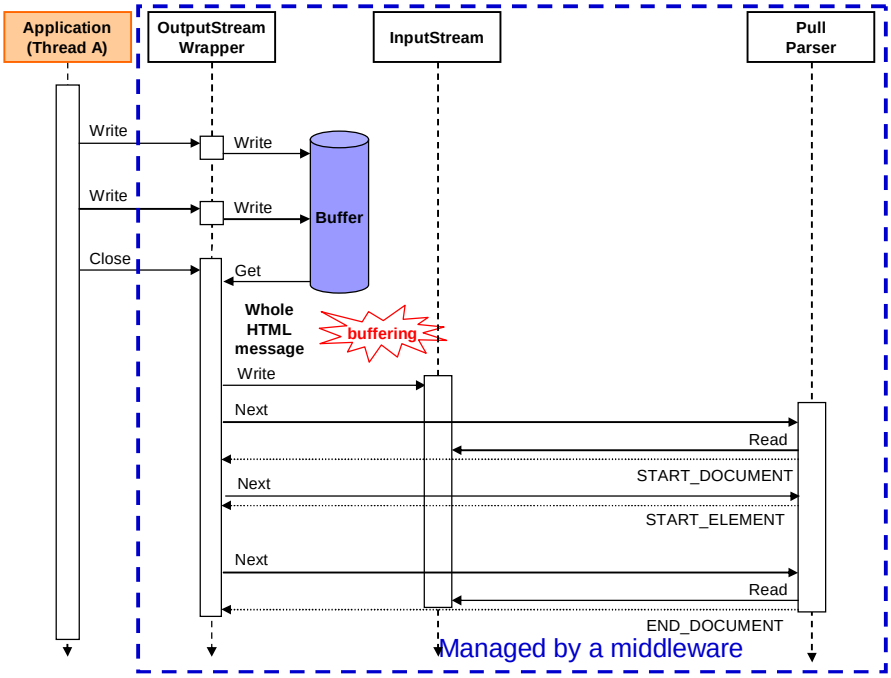
- 共有メモリアクセスの競合によるコア使用率の低下
- あるコアに割り当てられたスレッドの処理がボトルネックになることによるコア使用率の低下

この問題を解決するために, 以前から提唱されている単一スレッドを利用したプロセスモデルに回帰する流れがある. プロセスモデルとは, 一つの処理を単純に単一プロセス (単一スレッド) で処理する技術であり, 独立に処理可能な各プロセスを各コアに割り当てることで CPU のマルチコアを効率よく利用し, 高速化を図る. ただし, 従来の CPU と違って, 各 CPU コアが利用可能なキャッシュやメモリに大きな制約があるため, 一つのプロセスにおける処理も同様に効率化しない限り, マルチコア CPU の性能を十分に引き出すことは難しい.

また, 一方で Web アプリケーションの脆弱性をついて XSS に代表されるような悪意のある攻撃が頻繁に行われるようになってきている. このような攻撃を防ぐためには, Web アプリケーション自体の脆弱性を修正するのとは別に, Web アプリケーションサーバといったミドルウェアで脆弱性を利用させないための枠組みも必要となる. これは既存の Web アプリケーションを変更できないような状況下でも役に立つ枠組みである. そのための一つの仕組みとして, Web アプリケーションが構築した XML/HTML をミドルウェア内部で精査し, 脆弱性をついた攻撃コードが含まれている場合にはそれを取り除いて無害化するサニタイズという手法が存在する. XML/HTML のサニタイズを実現するためには, ミドルウェア内部で一度 XML/HTML をパーズし, XML/HTML の内部構造を正しく認識する必要がある.

本開示書では, 単一スレッドにおける処理のうち, この XML/HTML のパーズという処理に着目し, そのパーズ処理をアプリケーションを変更することなく, ミドルウェア内部のみで最適化するための技術を開示する. XML/HTML のパーズには, SAX, DOM, StAX といった Standard API を利用する必要があるが, これらの API を単一スレッドで利用する場合には, パーズを開始する前にあらかじめ対象となる XML/HTML ドキュメント全体をメモリ上にバッファしておかなければならない. 以下に, ミドルウェア内部でアプリケーションによって作成された XML/HTML ドキュメントを OutputStream レベルでパーズする際の実装例を示す.





しかし，各 CPU コアが利用可能なメモリ量を超えるような XML/HTML ドキュメントをバッファリングできないため，巨大な XML/HTML ドキュメントが来た場合に効率よく処理できないことになる．複数スレッドモデルが有効な環境下では，2 つのスレッドを用いることでパーズできる方法 (付録 A 参照) が一般に知られている．しかし，単一スレッドによるプロセスモデルが有効な環境下では，既存の手法が適用できない．この問題を解決するために，アプリケーション（の実装およびロジック）を変更することなく、

ミドルウェア内部で単一スレッドでも巨大なXML/HTMLドキュメントも効率よく処理可能なXML/HTMLのパーズ手法を開示する． また、組込み機器など、使用できる計算機資源（メモリ、スレッド数）に制限がある環境でも、本開示書で記述した効率的な XML/HTML パーズ手法は有効である。

さらに，巨大なXML/HTMLドキュメントでもバッファリング可能な環境下でパーズを行う場合と比較して，本開示書で述べる手法を用いた場合に生じうる性能の劣化を防ぐために，パーザの内部実装方法にも言及する．

2. Summary of Invention

本発明で開示する手法は，既存のアプリケーションを変更することなく，ミドルウェア内部での要点を以下にまとめる．

(1) 本開示書で提案するシステムは，本開示書で述べる標準 API の拡張版を実装したパーザ (p1) ，XML/HTML ドキュメントの断片 (df(i)) を格納する Queue (q1) ，q1 に直接データを格納したり，p1 を呼び出して実際にパーズ処理を管理する m1 という 3 つの構成要素からなる．標準 API に対する拡張とは，Queue に格納してあるデータがなくなったときに処理を中断し，また新たなデータが Queue に格納されたときに円滑に再開できるための API を追加すること，もしくはそのために必要なイベントを追加することである．

(2) m1 により呼び出された p1 は q1 から read メソドによりデータを読み込める限り，標準 API 実装と同様の振る舞いをする．q1 の内部に格納されたデータがなくなり， q1 から read メソドにより読み込んだデータ量が 0 の場合には現在のパーズ処理を中断する．このとき，XML/HTMLドキュメントの任意の場所でデータが途切れても良いものとする．q1 が次の XML/HTML ドキュメントの断片 (df(i+1)) を受け取ると，df(i+1)の先頭からパーズを円滑に再開できるように，p1 は中断するまでにパーズした際の内部状態を完全に保持する．q1 から read メソドにより読み込んだデータ量が-1 の場合には，p1 は XML/HTML ドキュメントの終わりであると判断し，パーズを終了する．

(3) p1 の内部実装において，どのような内部状態においてもパーズを再開できるように，XML/HTMLの仕様に準拠した上で，パーズに必要な工程を必要最低限に細分化し，各細分化した工程にそれぞれ状態を割り当てて，フラットなアーキテクチャを採用する．

(4) q1 は m1 から見たとき，OutputStream として見えるように，OutputStream を実装する．m1 が write メソドで df(i)を書き込んできたときは，df(i)を記憶し，データポインタを df(i)の先頭にあわせる．m1 が close メソドを呼び出したときは，XML/HTML ドキュメントの終わりであることを記憶する．q1 内部で保持できる df(i)は一つだけであるため，df(i)の全てのデータを読み出すまで，新たな df(i+1)を書き込むことは出来ない．

(5) q1 は p1 から見たとき標準 API に準拠するように，InputStream を実装する．p1 が read メソドで df(i)のデータを要求するときは，必要なデータ量だけデータポインタを移動させて，読み込んだデータとそのデータ量を返す．df(i)にデータが残っていないときは df(i)を破棄し，read メソドの戻り値として 0 を返す．ただし，XML/HTML ドキュメントの終わりである場合には read メソドの戻り値として-1 を返す．

本手法により，単一スレッドによるプロセスモデルにおいても XML/HTML のパーズを行う前に XML/HTML ドキュメント全体をバッファリングする必要がなくなるため，メモリ使用量を気にすること必要がない．このため，巨大な XML/HTML ドキュメントもパーズすることが可能となる．また，効率よくパーズの中断，再開が行えるため，巨大な XML/HTML ドキュメントでもバッファリング可能な環境下でパーズを行う場合と比較して劣化する性能を最小限で食い止めることが期待できる．

本手法はマルチコアCPU環境下以外でも，複数スレッドモデルを適用しづらい計算機資源が限られた組み込み機器上でも同様に効果を発揮できると期待できる．

3. Description

手法の説明

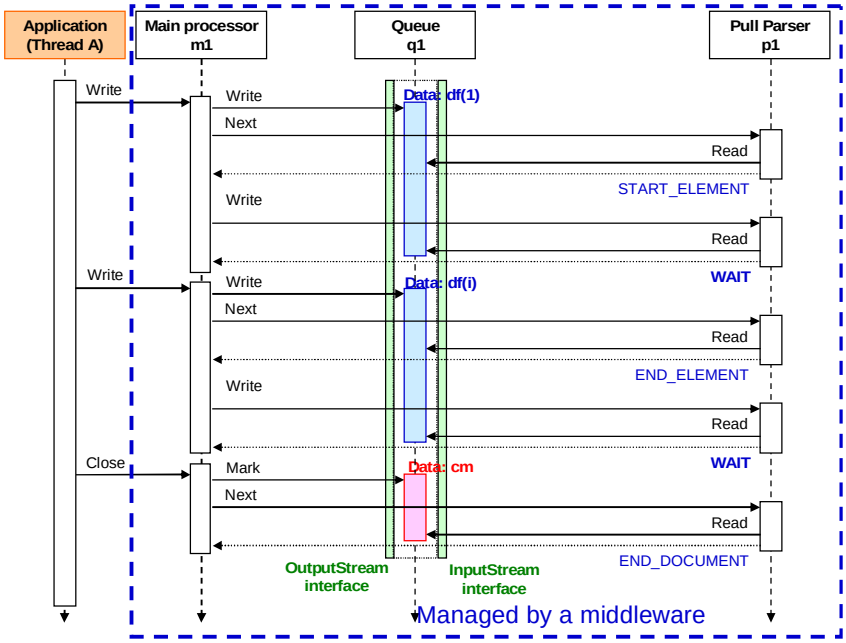
限られた計算機資源下で巨大な XML/HTML ドキュメントをパーズする場合，その全てをメモリ上にバッファリングできないため，断片を(受け取った)順に処理するのが一般的である．本手法では，既存の標準 API に XML/HTML ドキュメントのある断片から読み込めるデータがなくなったときに処理を中断し，また新たな XML/HTML ドキュメントの断片がきたときに円滑に再開できるための API を追加する，もしくはそのために必要なイベントを追加する，という非常に単純な方法でこの問題の解決を図る．

本開示書では，例として StAX API に対する拡張例を示す．StAX API とは Java6 で追加されたパーサモデルである．SAX など Push 型の XML パーザでは，パーザが主導的に XML 文書を最初から処理していき，要素などのトークンを発見したタイミングでアプリケーション側のコールバック関数を通じてイベントを通知する．一方で StAX では XML 文書の読み込み制御をアプリケーションが主導的に行うため，Pull 型パーザと呼ばれる．アプリケーションはパーザの next メソッドにより，トークンの存在を示すイベントを読み出す．

StAX API には付録 B に示すイベント群が既に用意されているが，これに WAIT イベントを追加することで，パーズ処理の任意の時点での中断および円滑な再開を可能となる．

Event Type	Description
Wait	XML/HTML ドキュメントのパーズ途中だが，Queue 中に残されたデータからはこれ以上イベントを生成できないときに返すイベント．これを受け取ったアプリケーションは，新たなデータを Queue に書き込むか，Queue を閉じて XML/HTML ドキュメントの終わりであることを知らせるか，しなければならない．

以下に，単一スレッド (Thread A) で動作するアプリケーション (a1) が XML/HTML ドキュメントの断片 (df(i) ($0 \leq i \leq n$)) を受け取りながら，順にパーズしていく場合のシステム構成および簡単な処理フローを示す．



本開示書で提案するシステムは，上記で示した StAX API 拡張版を実装したパーザ (p1) と XML/HTML ドキュメントの断片 (df(i)) を一つだけ格納可能な Queue (q1) ，q1 に直接データを格納したり，p1 を呼び出して実際にパーズ処理を管理する m1 という 3 つの構成要素からなる．各構成要素に要求される機能要件を以下に列挙する．

Main processor (OutputStream Wrapper) (m1)

- アプリケーションから，例えば write メソドにより，df(i)を受け取ると，q1 の write メソドを使って q1 に df(i)を書き出す．
- p1 を呼び出し，パーズを開始（もしくは再開）する．
- p1 から WAIT イベントを受け取ると，例えば write メソドの処理を終了し，アプリケーションに処理を戻す．
- アプリケーションから，例えば close メソドにより，XML/HTML ドキュメントの終わりを知らされると，q1 の close メソドを使って，q1 に XML/HTML ドキュメントの終わりを知らせる．

Parser (p1)

- q1 から read メソドによりデータを読み込める限り，一般的な StAX API 実装と同様の振る舞いをする．
- q1 から read メソドにより読み込んだデータ量が 0 (=read の戻り値が 0) の場合に q1 にデータが残っていないと判断する．この場合に，WAIT イベントを生成することになる．
- WAIT イベントを生成した後で，q1 が次の XML/HTML ドキュメントの断片 (df(i+1)) を受け取ると，df(i+1)の先頭からパーズを再開できるように，WAIT イベントを生成するまでにパーズした際の情報を最大限活用する．これは．パーズ性能の劣化を最低限に抑えるためである．詳細については後述するが，一般的な実装では，直前のイベントを生成したときのパーザの内部状態と df(i)内のデータポインタを記憶しておき，WAIT イベントを生成した後で，パーザの内部状態を復元し，df(i)のデータポインタを元に戻す必要がある．このため，WAIT イベントを生成するまでにパーズした際の情報が無駄になり，パーズに手戻りが発生するため，性能が劣化する．

- q1 から read メソドにより読み込んだデータ量が-1 の場合に XML/HTML ドキュメントの終わりであると判断し，EndDocument イベントを生成する．

Queue (q1)

- m1 から見たとき，OutputStream として見えるように，OutputStream を実装する．
- m1 が write メソドで df(i)を書き込んだときは，df(i)を記憶し，データポインタを df(i)の先頭にあわせる．ただし，q1 は一つの df(i)しか記憶できないため，df(i)が書き込まれる前に df(i-1)のデータを全て読み込んでいなければならない．
- m1 が close メソドを呼び出したときは，XML/HTML ドキュメントの終わりであることを記憶する．
- p1 から見たとき StAX API に準拠するように，InputStream を実装する．
- p1 が read メソドで df(i)のデータを要求するときは，必要なデータ量だけデータポインタを移動させて，読み込んだデータとそのデータ量を返す．df(i)にデータが残っていないときは df(i)を破棄し，read メソドの戻り値として 0 を返す．ただし，XML/HTML ドキュメントの終わりである場合には read メソドの戻り値として-1 を返す．

これらの機能要件を時系列にまとめると，一連の処理は以下のようになる．

(1) アプリケーションが m1 に対して write メソドにより df(1)を渡す．
(2) m1 は q1 に対して write メソドにより df(1)を渡す．
(3) q1 は df(1)がセットされると，内部データポインタを df(1)の先頭にセットする．
(4) m1 は p1 に最初のイベントを要求する．
(5) p1 はパーズを開始する．
(6) p1 は q1 に対して read メソドによりデータを要求する．
(7) q1 は df(1)にデータがある限り，読み込んだデータとそのデータ量を p1 に返す．
(8) p1 は読み込んだデータをもとにパーズを進める．データが足りなければ再度 q1 に要求するが，イベントを生成できるようになれば，そのイベントを生成して m1 に返す．
(9) m1 はイベントに応じて処理を進め，p1 に次のイベントを要求する．
(10) p1 がイベントを生成できる限りは(3)-(6)の処理を繰り返す．
(11) q1 が df(1)にデータがないことを認識して df(1)を破棄し，read メソドの戻り値として 0 を返した時点で，p1 はパーズの内部状態を保持したまま WAIT イベントを生成して，WAIT イベントを m1 に戻す．
(12) m1 は WAIT イベントを受け取ると，write メソドを終了し，アプリケーションに処理を戻す．
(13) アプリケーションは df(2)がある場合には，m1 に対して write メソドにより df(2)を渡す．そうでなければ XML/HTML ドキュメントの終わりを知らせるために m1 に対して close メソドを呼び出す．
(14) m1 は，write メソドが呼び出された場合には q1 に対して write メソドにより df(2)を渡し，close メソドが呼び出された場合には q1 に対して close メソドにより XML/HTML ドキュメントの終わりを知らせる．
(15) q1 は，write メソドが呼び出された場合には内部データポインタを df(2)の先頭にセットするが，close メソドが呼び出された場合には XML/HTML ドキュメントの終わりを知らせるフラグ (f1) を立てる．
(16) m1 は p1 に次のイベントを要求する．
(17) p1 は q1 に対して read メソドによりデータを要求する．
(18) q1 は df(2)を受け取っていれば(4)と同様に読み込んだデータとそのデータ量を p1 に返すが，f1 フラグが立っている場合には，データ量として-1 を返し，XML/HTML ドキュメントの終わりを p1 に知らせる．
(19) p1 は q1 からの read メソドの返り値として-1 を受け取ると，XML/HTML ドキュメントの終わりというこ

とで EndDocument イベントを m1 に返す.
(20)m1 は EndDocument イベントを受け取ると, パーズの処理を終了し, close メソドを抜け, アプリケーションに処理を戻す.

DOM API や SAX API を使って, パーズの中断および円滑な再開を実現するためには, StAX API の場合と異なり, 既存の parse メソドに改良を加える必要がある. 例えば, 一つの実装例としては, 以下のようにそれぞれの parse メソドを変更すればよい.

DOM API
既存メソド:

```
org.w3c.dom.Document javax.xml.parsers.DocumentBuilder.parse(
    java.io.InputStream is
) throws org.xml.sax.SAXException, java.io.IOException
```

新メソド:

```
org.w3c.dom.Document javax.xml.parsers.DocumentBuilder.parse(
    java.io.InputStream is
) throws javax.xml.InterruptException, org.xml.sax.SAXException, java.io.IOException
```

新メソドの振る舞い:
p1 は q1 の read メソドの戻り値として 0 を受け取った時点でパーズを中断し, InterruptException を投げる(StAX API において, WAIT イベントを返すのと同等の意味を持つ). これにより, m1 にパーズを中断したことを知らせる. また, p1 は q1 の read メソドの戻り値として-1 を受け取った時点でパーズを終了し, Document を返す. これにより, m1 にパーズを終了したことを知らせる.

SAX API
既存メソド:

```
void javax.xml.parsers.SAXParser.parse(
    java.io.InputStream q1, org.xml.sax.helpers.DefaultHandler dh
) throws org.xml.sax.SAXException, java.io.IOException
```

新メソド:

```
int javax.xml.parsers.SAXParser.parse(
    java.io.InputStream q1, org.xml.sax.helpers.DefaultHandler dh
) throws org.xml.sax.SAXException, java.io.IOException
```

新メソドの振る舞い:
p1 は q1 の read メソドの戻り値として 0 を受け取った時点でパーズを中断し, 1 を返す(StAX API において, WAIT イベントを返すのと同等の意味を持つ). これにより, m1 にパーズを中断したことを知らせる. また, p1 は q1 の read メソドの戻り値として-1 を受け取った時点でパーズを終了し, 0 を返す. これにより, m1 にパーズを終了したことを知らせる.

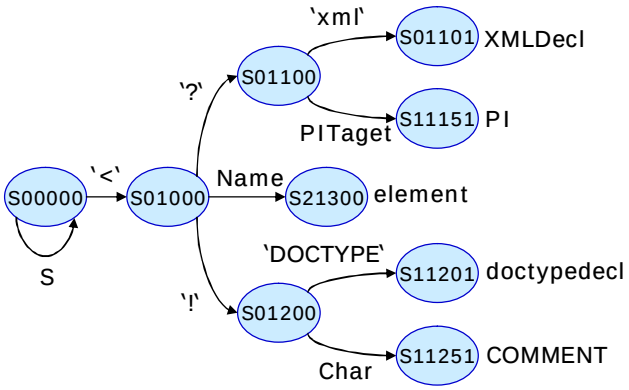
パーズの性能改善方法

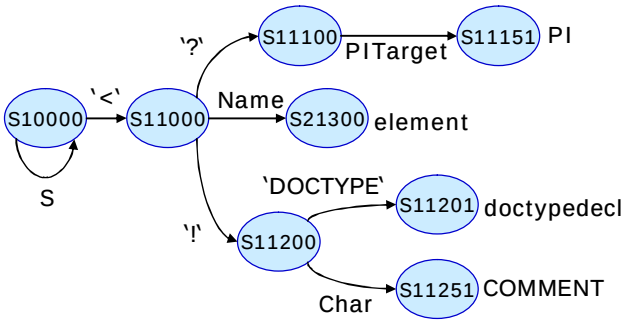
一般的なパーザ実装では, XML/HTML ドキュメントの任意の場所でパーズを中断して, ある一定時間が経過した後でパーズ再開するいった機能は必要ない. これは付録 A で示した実装のように, 処理の wait をパーザ外部で行えるためであるが, 単一スレッドによるプロセスモデルにおいては, パーザ内部でパーズ処理を明示的に中断して, 処理を一旦抜けなければならない. 一般的な実装を応用して

パーズ処理を再開させようとした場合，その実装は以下のようになるであろう．違いを明確にするために，前述した本開示書での手法と比較して，異なる部分を**太字**で示した．

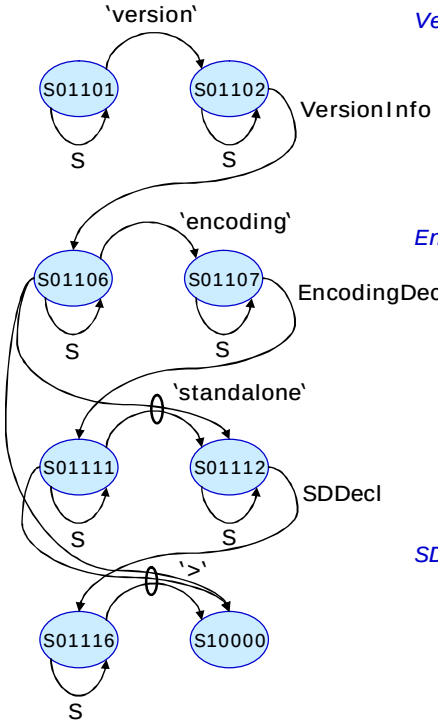
(1) アプリケーションが m1 に対して write メソドにより df(1)を渡す．
(2) m1 は q1 に対して write メソドにより df(1)を渡す．
(3) q1 は df(1)がセットされると，内部データポインタを df(1)の先頭にセットする．
(4) m1 は p1 に最初のイベントを要求する．
(5) p1 はパーズを開始する．
(6) p1 は q1 に対して read メソドによりデータを要求する．
(7) q1 は df(1)にデータがある限り，読み込んだデータとそのデータ量を p1 に返す．
(8) p1 は読み込んだデータをもとにパーズを進める．データが足りなければ再度 q1 に要求するが，イベントを生成できるようになれば，そのイベントを生成して m1 に返す．**このとき，q1 の内部データポインタの位置 (q1-pointer) と，p1 の内部処理状態 (p1-processingState) を記憶しておく．**
(9) m1 はイベントに応じて処理を進め，p1 に次のイベントを要求する．
(10)p1 がイベントを生成できる限りは(3)-(6)の処理を繰り返す．
(11)q1 が df(1)にデータがないことを認識して df(1)を破棄し，read メソドの戻り値として 0 を返した時点で，**p1 は内部状態を p1-processingState に戻し，q1 の内部ポインタも q1-pointer に戻し，** WAIT イベントを生成して，WAIT イベントを m1 に戻す．
(12)m1 は WAIT イベントを受け取ると，write メソドを終了し，アプリケーションに処理を戻す．
(13)アプリケーションは df(2)がある場合には，m1 に対して write メソドにより df(2)を渡す．そうでなければ XML/HTML ドキュメントの終わりを知らせるために m1 に対して close メソドを呼び出す．
(14)m1 は，write メソドが呼び出された場合には q1 に対して write メソドにより df(2)を渡し，close メソドが呼び出された場合には q1 に対して close メソドにより XML/HTML ドキュメントの終わりを知らせる．
(15)q1 は，write メソドが呼び出された場合には **df(2)を df(1)の後に連結させる**が，close メソドが呼び出された場合には XML/HTML ドキュメントの終わりを知らせるフラグ (f1) を立てる．
(16)m1 は p1 に次のイベントを要求する．
(17)p1 は q1 に対して read メソドによりデータを要求する．
(18)q1 は df(2)を受け取っていれば(4)と同様に読み込んだデータとそのデータ量を p1 に返すが，f1 フラグが立っている場合には，データ量として-1 を返し，XML/HTML ドキュメントの終わりを p1 に知らせる．
(19)p1 は q1 からの read メソドの返り値として-1 を受け取ると，XML/HTML ドキュメントの終わりということで EndDocument イベントを m1 に返す．
(20)m1 は EndDocument イベントを受け取ると，パーズの処理を終了し，close メソドを抜け，アプリケーションに処理を戻す．

この場合，あるイベントを生成してから，WAIT イベントを生成するまでに行ったパーズの情報を破棄することになり，パーズに手戻りが発生してしまう．これが性能の劣化を引き起こすことになる．この性能劣化を最小限にするために，本開示書では XML/HTML の仕様に準拠した上で，パーズに必要な工程を必要最低限に細分化し，各細分化した工程にそれぞれ状態を割り当てる．例えば，XML のパーズに必要な工程のうち，DTD を読み込むまでの工程の一部を細分化した際の一例を以下に示す．

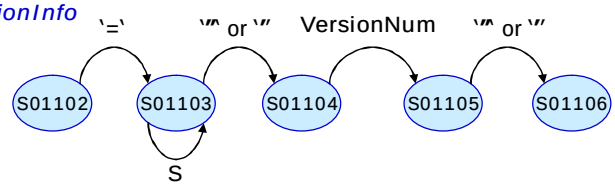




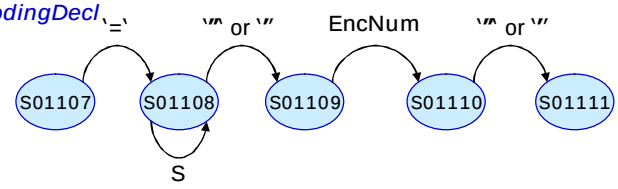
XMLDecl



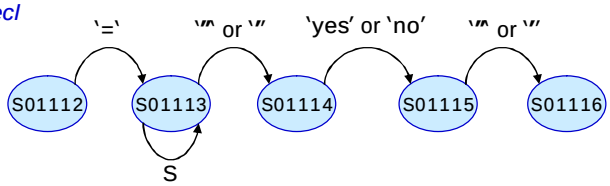
VersionInfo

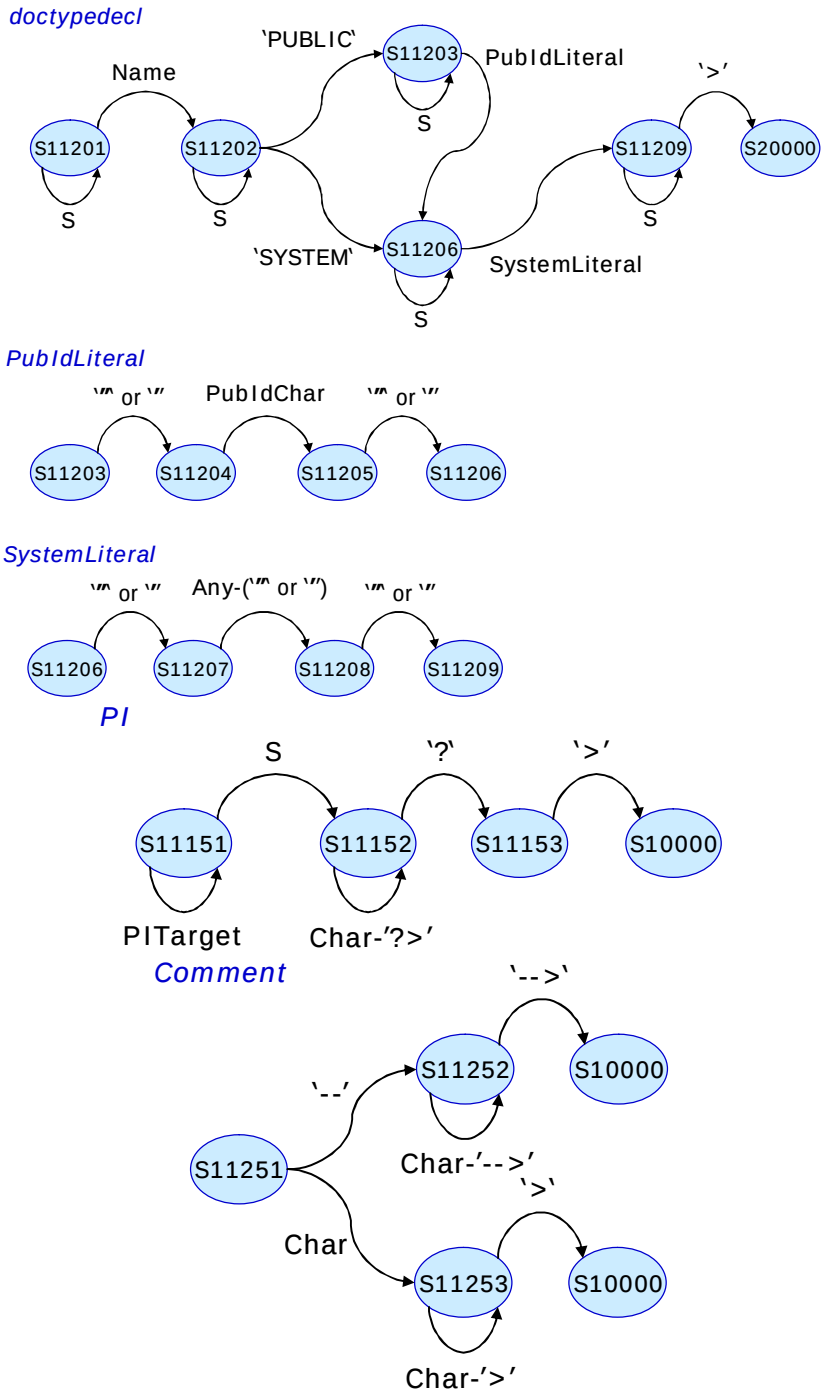


EncodingDecl



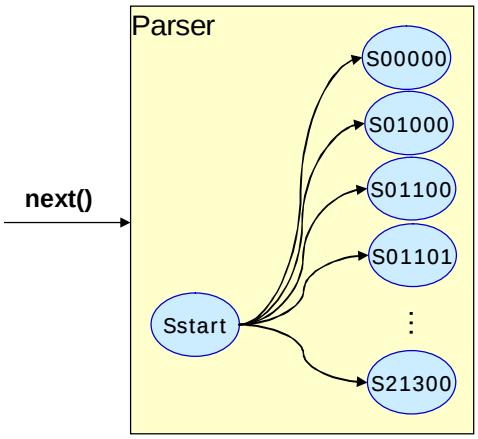
SDDecl





これらの状態間の遷移をフラットなアーキテクチャで実現するためには，m1 が next()メソドにより p1 にイベントを要求した時の処理の開始地点 S_{start} から全ての状態へ遷移できるようにしておき，状態を遷移するときには常に S_{start} を経由するようにすればよい．こうすることで，どの状態で WAIT イベントを発生させることになっても，処理の中断および再開が非常に容易になる．例えば，状態 S01100 において，

p1 が’xm’という文字列を読んだ状態 (まだ ‘xml’ という状態遷移を確定させる文字列は読み終わっていない状態) で q1 に読み込めるデータなくなると, p1 は処理を継続させることができなくなるため, m1 に WAIT イベントを返して next()メソドの処理を終了することになる.



ただし, 後にイベントを生成するために必要な情報は必ずメモリ上に保持しておくようにする. 例えば, 上記の場合だと, 現在の状態 (S01100) と既に読み込んだ’xm’という文字列はメモリ上に残しておかなくてはならない.

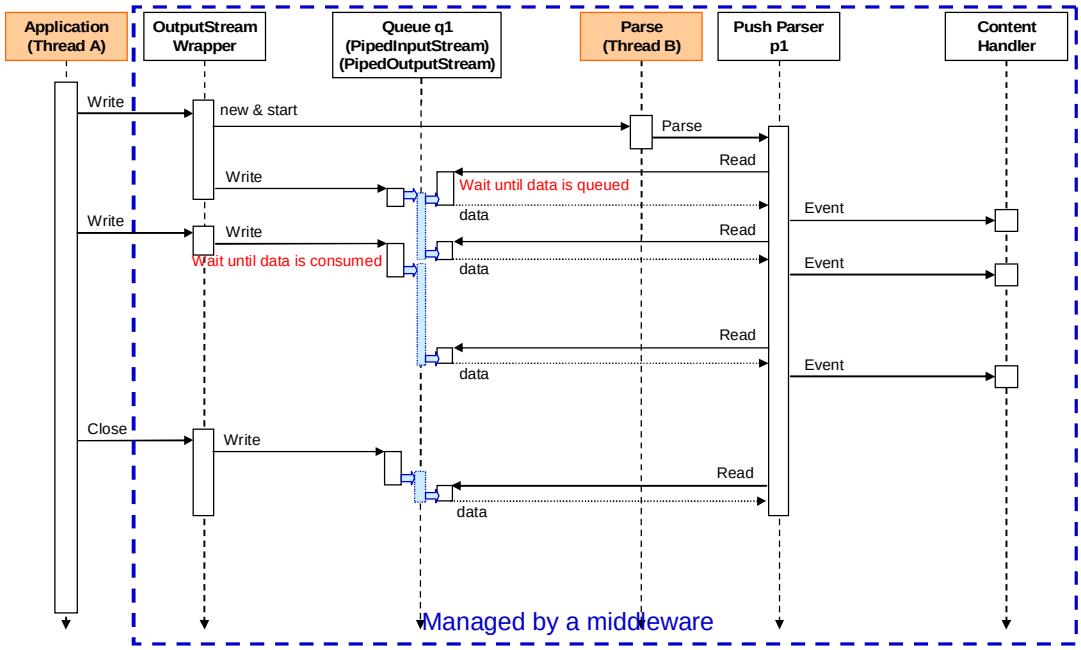
上記は単純な一例を示すものであるが, もう少し複雑な場合でも同様の仕組みでパーズを再開することが可能となる. 例えば, とある要素<p1:elem1 xmlns:p1=”http://xmln.sp1.value” attr1=”attr1value”>を読み込む場合を考えてみる. この場合, p1 は最終的には startElement (要素の開始タグを表す) イベントを生成することになるのだが, df(i)=” <p1:elem1 xmlns:p1=”http://xmln”とすると, df(i)の最後まで読んだだけでは startElement イベントを生成することはできないため, パーズを一旦中断する必要がある. この場合には, 以下に示すデータをメモリ上に残しておかなければならない.

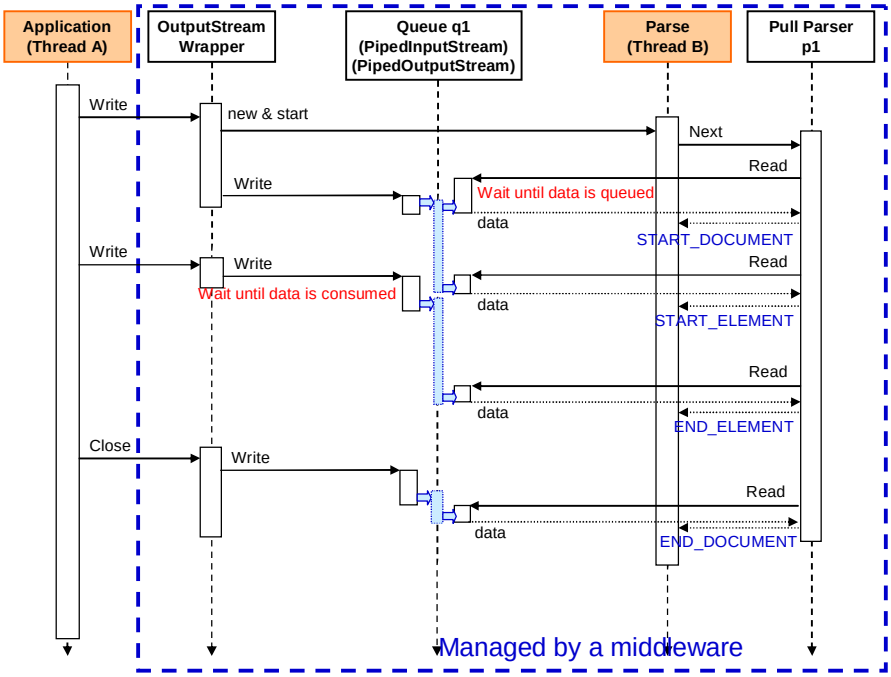
- 現在の処理状態: 名前空間宣言の URI を読み込んでいることを示す状態
- 要素の接頭辞: p1
- 要素の局所名: elem1
- 名前空間宣言に使われている接頭辞: p1
- 名前空間宣言の URI 断片: http://xmln

これらの情報は, 現在の状態, 読み込んだデータ量により動的に変化するが, 重要なポイントは, イベントを生成するために必要で, かつそれまでに読み込んだデータは全てメモリ上に残しておく必要があるということである.

付録 A.
複数スレッドモデルにおいて，DOM/SAX/StAX による XML/HTML ドキュメントのパーズのメモリ使用量を減らすための一般的な実装方法

java.io.PipedInputStream と java.io.PipedOutputStream を利用した方法であり，2 つのスレッドを用いて，1 つのスレッドが Queue への XML/HTML ドキュメントの断片の書き出し (write) を担い，もう 1 つのスレッドが Queue からの断片の読み出し (read) とパーズ (parse) を担う方法である．基本的に各スレッドはもう片方の read or write が終わるまで，次の read or write を行わない．





上記例においては Thread A が write の役割を担い，Thread B が read & parse の役割を担っている．この場合，Thread A が XML/HTML ドキュメントの最初の断片(df1) を受け取ると，Queue (q1) に df1 を格納し，Thread B を立ち上げ，XML/HTML ドキュメントのパーズを開始する．Thread B は q1 を使って，Parser (p1) を呼び出す．p1 は q1 からデータを読み出しながら，df1 のパーズを行う．df1 を全て読みきり q1 が空になった時点で次のデータが来るまで Thread B は wait する．Thread A は次の断片 df2 を受け取ると q1 に以前に書き込んだデータ (この場合 df1) を Thread B が読み込むまで wait し，読み込んだことを確認できれば df2 を q1 に書き込む．この作業を XML/HTML ドキュメントの終了まで繰り返す．

付録 B.
StAX API で規定されているイベント群.
(参照: http://java.sun.com/webservices/docs/1.6/tutorial/doc/SJSXP3.html)

Event Type	Description
StartDocument	Reports the beginning of a set of XML events, including encoding, XML version, and standalone properties.
StartElement	Reports the start of an element, including any attributes and namespace declarations; also provides access to the prefix, namespace URI, and local name of the start tag.
EndElement	Reports the end tag of an element. Namespaces that have gone out of scope can be recalled here if they have been explicitly set on their corresponding StartElement.
Characters	Corresponds to XML CData sections and CharacterData entities. Note that ignorable whitespace and significant whitespace are also reported as Character events.
EntityReference	Character entities can be reported as discrete events, which an application developer can then choose to resolve or pass through unresolved. By default, entities are resolved. Alternatively, if you do not want to report the entity as an event, replacement text can be substituted and reported as Characters.
ProcessingInstruction	Reports the target and data for an underlying processing instruction.
Comment	Returns the text of a comment
EndDocument	Reports the end of a set of XML events.
DTD	Reports as java.lang.String information about the DTD, if any, associated with the stream, and provides a method for returning custom objects found in the DTD.
Attribute	Attributes are generally reported as part of a StartElement event. However, there are times when it is desirable to return an attribute as a standalone Attribute event; for example, when a namespace is returned as the result of an XQuery or XPath expression.
Namespace	As with attributes, namespaces are usually reported as part of a StartElement, but there are times when it is desirable to report a namespace as a discrete Namespace event.