

D I O S A / X T P V1.1

利用の手引

輸出する際の注意事項

本製品(ソフトウェア)は、外国為替及び外国貿易法で規制される規制貨物(または役務)に該当することがあります。

その場合、日本国外へ輸出する場合には日本政府の輸出許可が必要です。

なお、輸出許可申請手続きにあたり資料等が必要な場合には、お買い上げの販売店またはお近くの当社営業拠点にご相談下さい。

はしがき

本書はD I O S A / X T Pにおいて利用者アプリケーションの制御を司るアプリケーション制御機能と通信制御の利用方法について説明したものです。

アプリケーション制御機能は、アプリケーションシステム本来の目的である業務処理部分についての開発・保守に専念できるように制御処理の部分を支援し、オンラインならびにバッチアプリケーションプログラムの開発・保守における生産性・即応性を大きく向上するものです。

本書の読者としては、業務アプリケーション開発を担当し、HP-UX、TPBASE、TAM、Oracle、その他関連 PP の使用法を一通り心得ているシステム技術者を想定しています。

2012 年 10 月 初版

2016 年 6 月 10 版

本書の関連説明書としては次のものがあります。

- D I O S A / X T P 導入の手引き
- D I O S A / X T P メモリキャッシュ 利用の手引き
- D I O S A / X T P データストア 利用の手引き
- D I O S A / X T P コマンドリファレンス
- D I O S A / X T P A P I リファレンス
- D I O S A / X T P 環境定義リファレンス
- D I O S A / X T P メッセージリファレンス

備考

- (1) Microsoft、Windows は、米国あるいはその他の国における米国 Microsoft Corporation の商標または登録商標です。
- (2) UNIX は、X/Open カンパニーリミテッドが独占的にライセンスしている米国ならびに他の国における登録商標です。
- (3) HP、HP-UX は、Hewlett-Packard 社の商標または登録商標です。
- (4) Linux は、Linus Torvalds の米国およびその他の国における商標または登録商標です。
- (5) Red Hat は、米国およびその他の国における Red Hat, Inc. の商標または登録商標です。
- (6) Oracle と Java は、Oracle Corporation およびその子会社、関連会社の米国およびその他の国における登録商標です。
- (7) This product includes software developed by the Apache Group for use in the Apache HTTP server project (<http://www.apache.org/>).
- (8) その他、記載されている会社名、製品名は、各社の登録商標または商標です。

目次

- 第 1 章 概要 1
 - 1.1 目的と特徴 1
 - 1.1.1 目的..... 1
 - 1.1.2 特徴..... 1
 - 1.2 構成 3
 - 1.2.1 位置づけ..... 3
 - 1.2.2 システム構成..... 3
 - 1.2.3 機能構成..... 4
 - 1.3 諸概念 5
- 第 2 章 機能 6
 - 2.1 C O 制御機能 6
 - 2.1.1 機能説明..... 6
 - 2.1.2 A P の呼び出し機能 9
 - 2.1.3 電文送受信機能..... 12
 - 2.1.4 C O 連携機能 13
 - 2.1.5 コミット／ロールバック機能..... 16
 - 2.1.6 アボート処理機能..... 17
 - 2.1.7 A P のループ／ストール監視機能 19
 - 2.1.8 ロールバック連鎖機能..... 20
 - 2.1.9 C O 制御への強制リターン機能 21
 - 2.1.10 ロールバックリトライ機能..... 22
 - 2.1.11 デッドロックリトライ機能..... 23
 - 2.1.12 電文保留機能..... 24
 - 2.2 バッチアプリケーション制御機能 25
 - 2.2.1 機能説明..... 25
 - 2.2.2 A P 呼び出し機能 26
 - 2.2.3 稼動統計情報収集機能..... 27
 - 2.2.4 ループ／ストール監視機能..... 28
 - 2.2.5 実行レポート出力機能..... 28
 - 2.2.6 D B 自動制御機能 29
 - 2.2.7 ロールバックリトライ機能..... 32
 - 2.3 タイマ制御機能 33
 - 2.3.1 機能説明..... 34
 - 2.4 メモリ管理機能 35
 - 2.4.1 機能説明..... 35
 - 2.4.2 環境設定..... 37
 - 2.5 ロック制御機能 38

2.5.1	機能説明.....	38
2.6	メッセージ出力機能	39
2.6.1	機能説明.....	40
2.7	アプリケーショントレース機能	41
2.7.1	機能説明.....	41
2.8	アプリケーション動的置換機能	44
2.8.1	諸概念.....	44
2.8.2	機能説明.....	45
2.8.3	API/コマンド	48
2.9	経過時間監視機能	49
2.9.1	経過時間監視機能.....	49
2.9.2	経過時間リセット機能.....	51
2.9.3	経過時間監視停止／再開機能.....	52
2.9.4	ユーザ情報登録機能.....	52
2.9.5	経過時間監視照会機能.....	52
2.10	稼動統計機能	53
2.10.1	稼動統計出力機能.....	53
2.10.2	稼動統計収集機能.....	57
2.10.3	運用方法.....	59
2.11	Tパス管理機能.....	61
2.11.1	Tパスの管理方法について.....	61
2.11.2	Tパスのオープン・クローズ検出方法.....	61
2.11.3	ヘルスチェック間隔の変更.....	64
2.11.4	マルチ TPBASE 対応.....	64
2.11.5	閉塞制御連携機能.....	65
2.11.6	定義の動的変更機能.....	67
2.12	流量制御機能	68
2.12.1	負荷情報収集.....	68
2.13	データベース管理機能	69
2.13.1	DBマルチコネクション制御.....	69
2.13.2	DBインスタンス振り分け機能.....	70
2.13.3	DBヘルスチェック機能.....	70
2.13.4	DB関連API.....	72
2.14	閉塞管理機能	76
2.14.1	機能説明.....	76
2.15	コマンド配信機能	78
2.15.1	コマンド配信ユーザインタフェース.....	78
2.15.2	コマンド配信宛先.....	78
2.15.3	コマンドルーティング.....	81
2.15.4	コマンド実行権限.....	82

2.15.5	コマンド配信結果.....	84
2.15.6	入力ファイル転送.....	85
2.15.7	タイムアウト監視.....	85
2.15.8	リトライ処理.....	87
2.15.9	環境変数設定.....	88
2.15.10	コマンド配信履歴	88
第3章	アプリケーション開発	90
3.1	COプログラミング	90
3.1.1	プログラムの構造.....	90
3.1.2	CO制御とのインタフェース	91
3.1.3	通信プログラムの開発.....	93
3.1.4	電文保留.....	95
3.1.5	CO制御利用者出口の開発	113
3.1.6	メモリ管理機能とのインタフェース.....	115
3.1.7	ロック制御機能とのインタフェース.....	117
3.1.8	アプリケーショントレース機能とのインタフェース.....	118
3.1.9	アプリケーション動的置換機能とのインタフェース.....	120
3.1.10	コマンド配信機能とのインタフェース.....	121
3.1.11	タイマ制御機能とのインタフェース.....	124
3.2	ユーザアプリケーションプログラム	127
3.2.1	プログラムの構造.....	127
3.3	通信制御プログラミング	131
3.3.1	電文種別決定出口.....	131
3.3.2	データベース接続出口.....	132
3.4	アプリケーションの生成	133
3.4.1	COプログラム	133
3.4.2	ユーザアプリケーションプログラム.....	134
3.4.3	利用者出口.....	135

第1章 概要

1.1 目的と特徴

1.1.1 目的

DIOSA/XTP は、大規模アプリケーションシステム構築の汎用基盤を提供するソフトウェアです。

社会インフラの重要性が増す中、大規模アプリケーションシステムにおいても、高信頼、高性能、高運用・高稼働性、そしてアプリケーション開発の高生産性・即応性への要求が益々高まっています。

DIOSA/XTP はこれらの要件を満足すべく以下の機能を実現します。

- メモリDBを利用して高速なデータアクセスを実現しつつ、障害時の高速な復旧による業務継続を可能とします。
- システム運用の自動化・省力化、24 時間運用システムを実現するための機能により、高運用・高稼働性を実現します。

1.1.2 特徴

DIOSA/XTP は大きく分けて以下の特徴を備えています。

- アプリケーションプログラムの開發生産性向上
- 24 時間運転システムの実現
- 拡張性の高い大規模・高信頼性・高可用性システムの構築支援

(1) アプリケーションシステムの開発を容易にします。

- オンライントランザクションプログラムの基本処理構造を規定することにより、アプリケーションプログラムの独立性を高め、標準化された開発が行えます。
- 大規模・分散システムを達成するための所在管理や電文のルーティング、電文送受信の制御作業から解放されます。
- アプリケーション開発のためのトレース、性能解析といったプログラム開発の下流工程を支援する機能群により、アプリケーション開発および本番時の運用作業が軽減されます。
- メモリDBを複数のアプリケーションから同時利用可能にし、高速大量処理を可能とします。
- データを分散してメモリDBに配置し、アクセスのための所在管理とルーティング制御を行い、全データへの透過的なアクセスを可能とします。

(2) 24 時間運転システムの稼働を支援します。

- オンライン業務の稼働中に、動作中プログラムの置換を瞬時に行うことが可能であり、業務の追加・変更、プログラム障害時の緊急対応を容易に行うことができます。
- ノードの所在を意識することなく、任意のノードに対し運用指示や状態照会のコマンドを投入することができます。
- オンライン業務の稼働中に、ノード追加などシステム構成の変更や動作環境の変更を行うことが可能であ

り、柔軟なシステムの保守作業を実現します。

(3) **拡張性の高い大規模・高信頼性・高可用性システムを容易に構築することが可能です。**

- OLTP レベルあるいはノードレベルでの分散形態を多様に構成することにより、アプリケーションプログラムへの影響無く、大規模かつ高信頼なシステムを構築することができます。
- メモリ DB の稼動状態を管理し、障害時はマスタ／スレーブ切り替えを自動的に行うことを可能とします。
- Oracle Real Application Clusters を利用したクラスタ構成に対応しており、アプリケーションプログラムは切り替えを意識せずにアクセスするサーバを変更することが可能です。

1.2 構成

1.2.1 位置づけ

DIOSA/XTP は、UNIX オペレーティングシステム、および OLTP や DB などミドルウェアとアプリケーションプログラムの上に立ち、分散オンライントランザクション処理システムのミッションクリティカル性を向上させるアプリケーション実行環境として位置付けられます。

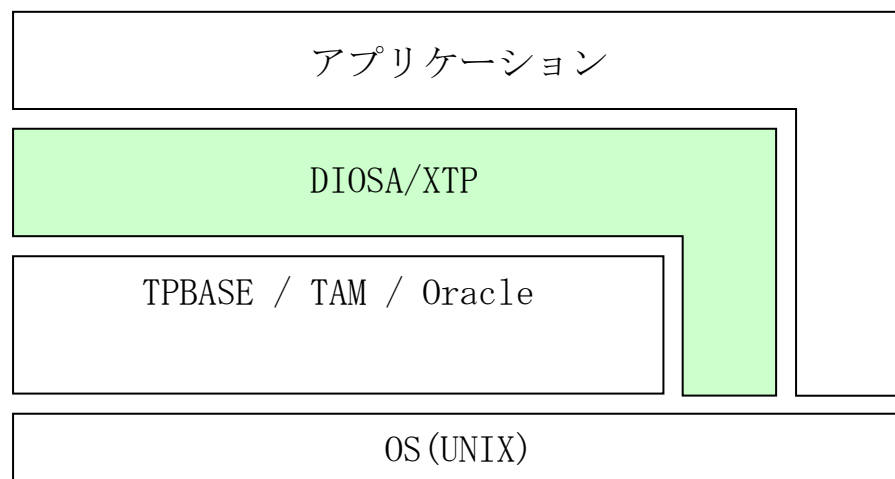
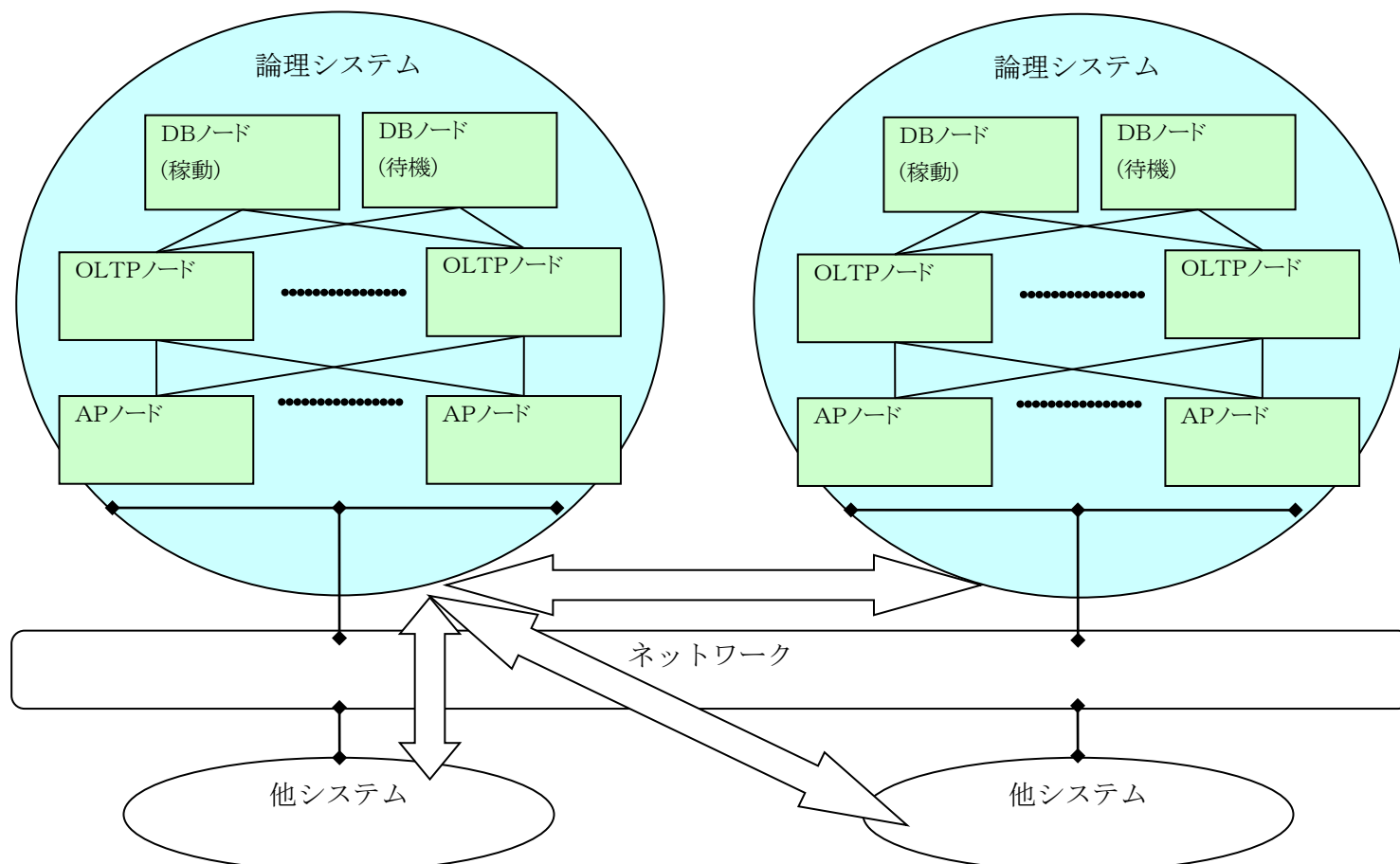


図 1-1 DIOSA/XTP の位置付け

1.2.2 システム構成

DIOSA/XTP の適用可能なシステム構成の例を以下に示します。



1.2.3 機能構成

DIOSA/XTP は大きく以下の有償プログラムプロダクト (PP) 群に分けて構成されています。

ライセンス	機能	関連 PP
アプリケーション実行制御	CO 制御機能 バッチアプリケーション制御機能 タイマ制御機能 メモリ管理機能 ロック制御機能 メッセージ出力機能 アプリケーショントレース機能 アプリケーション動的置換機能 経過時間監視機能 稼働統計機能 閉塞管理機能 コマンド配信機能	TPBASE TAM Oracle
通信制御	Tパス管理機能 流量制御機能 データベース管理機能	TPBASE Oracle
メモリキャッシュ	インメモリサーバ インメモリサーバ所在管理機能	TAM
データストア	ディレード転送機能	TAM Oracle

1.3 諸概念

DIOSA/XTP を理解するために、予め理解しておくべき概念・用語について説明します。

なお、メモリキャッシュ、データストアに関しては、以下の利用の手引きの記述を参照してください。

- ・ メモリキャッシュ 利用の手引
- ・ データストア 利用の手引き

(1) 論理システム

巨大なシステムを構成する独立したシステムの単位であり、運用の単位、通信の単位です。

論理システムは、複数の論理ノード(AP ノード、OLTP ノード、DB ノード)から構成されます。

(2) 論理ノード

論理システムを構成するサーバ上で動作する特定の機能群です。

1つの物理サーバ上で、複数の論理ノードを動作させることが可能です。

ノード種別により動作可能な機能群が変わります。

● DB ノード

Oracle Database が動作するノードです。

主に DB の状態を管理する機能が動作します。

● OLTP ノード

アプリケーションが動作するノードです。

メモリ DB を使った高速処理のアプリケーションを実現できます。

● AP ノード

OLTP ノードと外部の通信を中継するノードです。

このノードで外部／内部のプロトコル変換などを実現します。

(3) Tパス

同一論理システム内の論理ノード間で確立する論理的な通信パスの事を指します。

(4) CO(制御)／出口

CO (Control Object) と呼ばれる業務プログラムをイベント (メッセージ) により連結して処理することを実現します。

COはCO制御と呼ばれるフレームワークで制御されます。

CO制御はCO (業務) の呼び出し制御の他、トランザクションとして定型的な処理をするための処理を出口として呼び出すことができます。

第2章 機能

2.1 CO制御機能

CO制御とは、CO (Control Object) と呼ばれる利用者プログラムを使って業務をおこなうためのフレームワークであり、TPBASEのTPPとして動作します。

CO制御はイベント（メッセージ）駆動型プログラミングを採用しています。イベントの送受信はCO制御が提供するAPIを使うことで可能になります。共通関数呼び出しで業務APを構築しようとするよりも、COをイベントで動作させることで、障害の局所化やプログラムを簡素にすることが可能となります。

データアクセスはTAMを利用します。TAMへのアクセスは、インメモリサーバと呼ばれるサーバを介しておこなわれます。利用者がOracle等データベースを使用する場合はCO、利用者出口で接続、コミット、ロールバック等トランザクション制御をおこなう必要があります。

CO制御は以下の機能を提供します。

2.1.1 機能説明

(1) APの呼び出し機能

電文の発信元、受信電文解析出口または環境定義で指示されたCOを呼び出します。COのように業務に対応しないAP処理として、プロセス、トランザクション、コミット、ロールバック等トランザクション処理の切り替わり毎に利用者出口処理を呼び出すことができます、出口は環境定義で定義することができます。必要とされる出口をAPは作成してください。COと利用者出口は、アプリケーション動的置換機能を通じて呼び出しますので動的に置換することができます。

(2) 電文送受信機能

イベント（以降、断りがない限りメッセージ、電文は同じ意味です）の受信API (diosarecvtx)と送信API (diosarecvtx)を提供します。本APIを使うことにより論理システム間（外部論理システム宛）、論理システム内（内部論理ノード宛）等の電文を区別することなく送受信することができます。

(3) CO間連携機能

論理システム内で複数のCOが連携して業務処理をおこなうことが可能となります。

CO間連携機能には、連鎖、派生があります。

(4) コミット／ロールバック機能

APの終了状態に対応して、自動的にコミットまたはロールバックを行います。

(5) アボート処理機能

APからアボート処理が要求された時は、ロールバックとアボート出口の呼び出しを行います。

プログラム例外として扱うシグナルが発生した時も、アボート出口の呼び出しを行います。ロールバックはインメモリサーバにまかせます。

(6) APのループ／ストール監視機能

CPU消費時間と経過時間を監視します。

制限値を超えたCO制御（プロセス）を警告メッセージとして出力するか、または停止します。

(7) **ロールバック連鎖機能**

A P が論理的に異常を検出した時などに使用する機能で、ロールバックを行って指示された（アボート処理用の）C O を呼び出します。

(8) **C O 制御への強制リターン機能**

A P が論理的に異常を検出した時などに使用する機能でC O B O L のG O B A C K M A I N 相当の機能です。下位層のプログラムから上位層のプログラムを跳び越してC O 制御に直接リターンすることができます。

(9) **ロールバックリトライ機能**

A P が論理的に異常を検出した時などに使用する機能で、ロールバックを行って再処理します。

(10) **デッドロックリトライ機能**

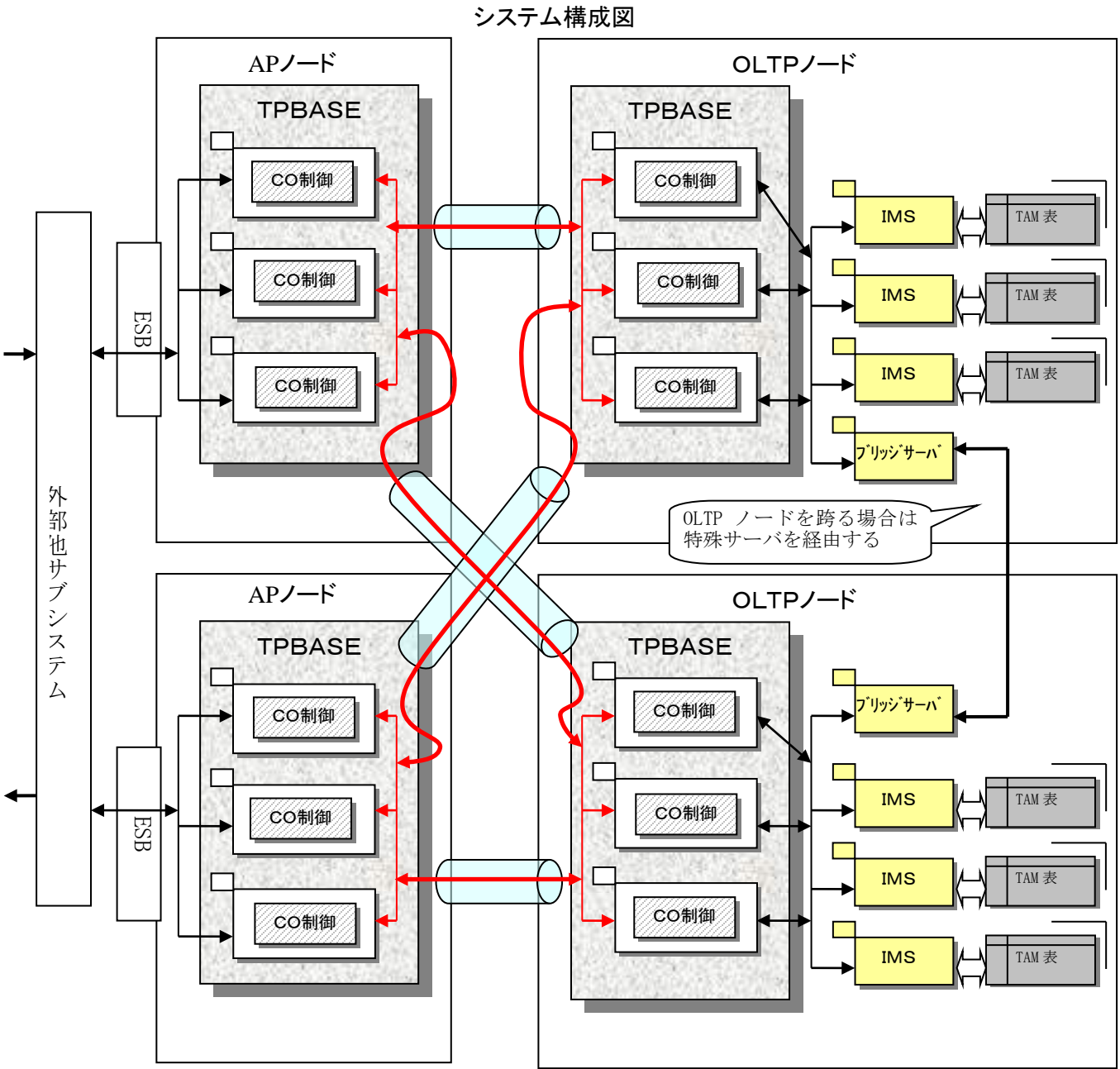
デッドロック発生時、ロールバックを行って再処理します。

(11) **電文保留機能**

受信電文を一時的に退避しておき、再処理する機能を提供します。

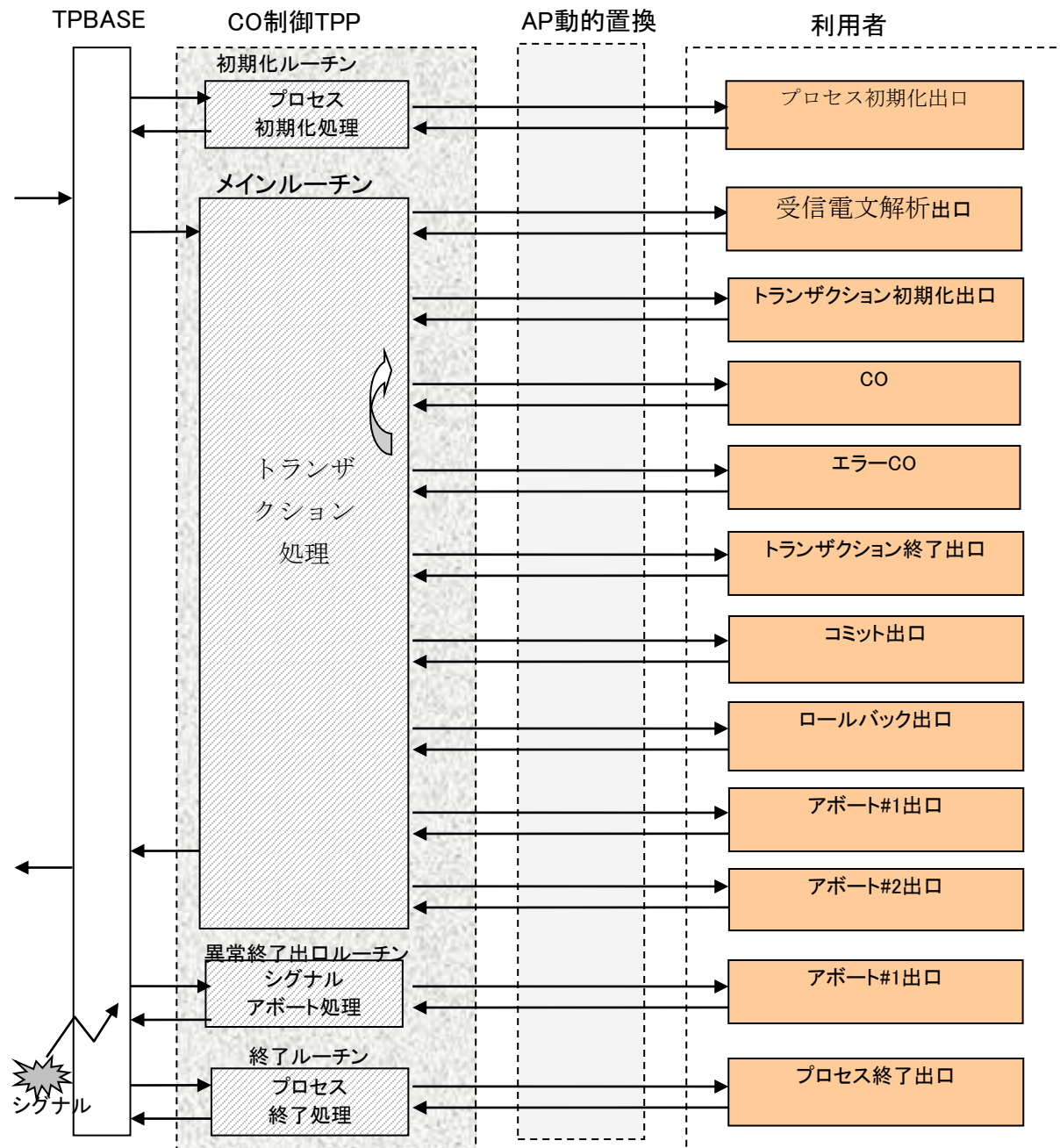
ロールバック、デッドロックリトライは同一トランザクションで即時に実行されますが、電文保留機能の再処理は別トランザクション、または別ノードで遅延して再実行します。

システム内のCO制御とインメモリサーバの配置関係を以下に載せる。



2.1.2 APの呼び出し機能

COを呼び出します。また、環境定義で定義された各種の利用者出口の呼び出しも行います。COと利用者出口は、アプリケーション動的置換機能を通じて呼び出しますので動的に置換することができます。CO制御が呼び出すCOと利用者出口を以下の図に示します。



(1) **ＣＯと利用者出口の呼び出し契機**

ＣＯと利用者出口の呼び出し契機を以下に示します。

呼び出しプロセス	ＣＯ／利用者出口	呼び出し契機
ＣＯ制御サーバ	プロセス初期化出口	プロセス開始時
	トランザクション初期化出口	トランザクション開始時（電文受信時）
	受信電文解析出口	電文受信時に呼び出します。
	ＣＯ	電文を受信した時、受信電文解析出口、トランザクション初期化出口の呼び出し後に呼び出します。 ＣＯから連鎖要求があった場合、連鎖ＣＯを呼び出します。
	エラーＣＯ	ＣＯの呼び出し失敗時、呼び出すＣＯが決定できていない時 またはＣＯが閉塞されている時呼び出します。
	トランザクション終了出口	トランザクション終了時(受信電文の処理が正常に終わった時)。ただしアボート時には呼び出しません。
	アボート＃１出口	ＡＰからのアボート要求時、およびシグナル例外発生時に、 ロールバック前に呼び出します。 アボート処理から呼ばれるアボート＃１出口とシグナルアボート処理から呼ばれるアボート出口＃１は同じものです。
	アボート＃２出口	ＡＰからのアボート要求時、ロールバック後に呼び出します。
	コミット出口	・トランザクション正常終了時にトランザクション終了出口に続いてよびだされます。 ・アボート処理時、アボート＃２出口に続いて呼び出されます。 ・コミット API (diosacommit) の延長で呼び出されます
	ロールバック出口	・C0 がロールバック連鎖要求したとき、連鎖 C0 呼び出し前に呼び出されます。 ・C0 がロールバックリトライ要求したとき、リトライ処理前に呼び出されます。 ・C0 とトランザクション終了出口がデッドロックリトライ要求したとき、リトライ処理前に呼び出されます。 ・アボート処理時、アボート＃１出口に続いて呼び出されます。 ・ロールバック API (diosarollback) の延長で呼び出されます
	プロセス終了出口	プロセス終了時

(2) **ＣＯと利用者出口の呼び出しインタフェース**

ＣＯ制御とＣＯ、利用者出口間の情報交換にD I O S A U C A（Diosa User Communication Ares）と呼ばれる領域を使います。

ＡＰはD I O S A U C Aから実行環境の情報（ノード名、ＴＰモニタ名、トランザクションＩＤ等）、トランザクション実行状況（リトライ回数、コミット回数）、アボート理由等を得ることができます。ＡＰの

出力情報は、正常、異常、リトライ等をC O制御に返却することができます。

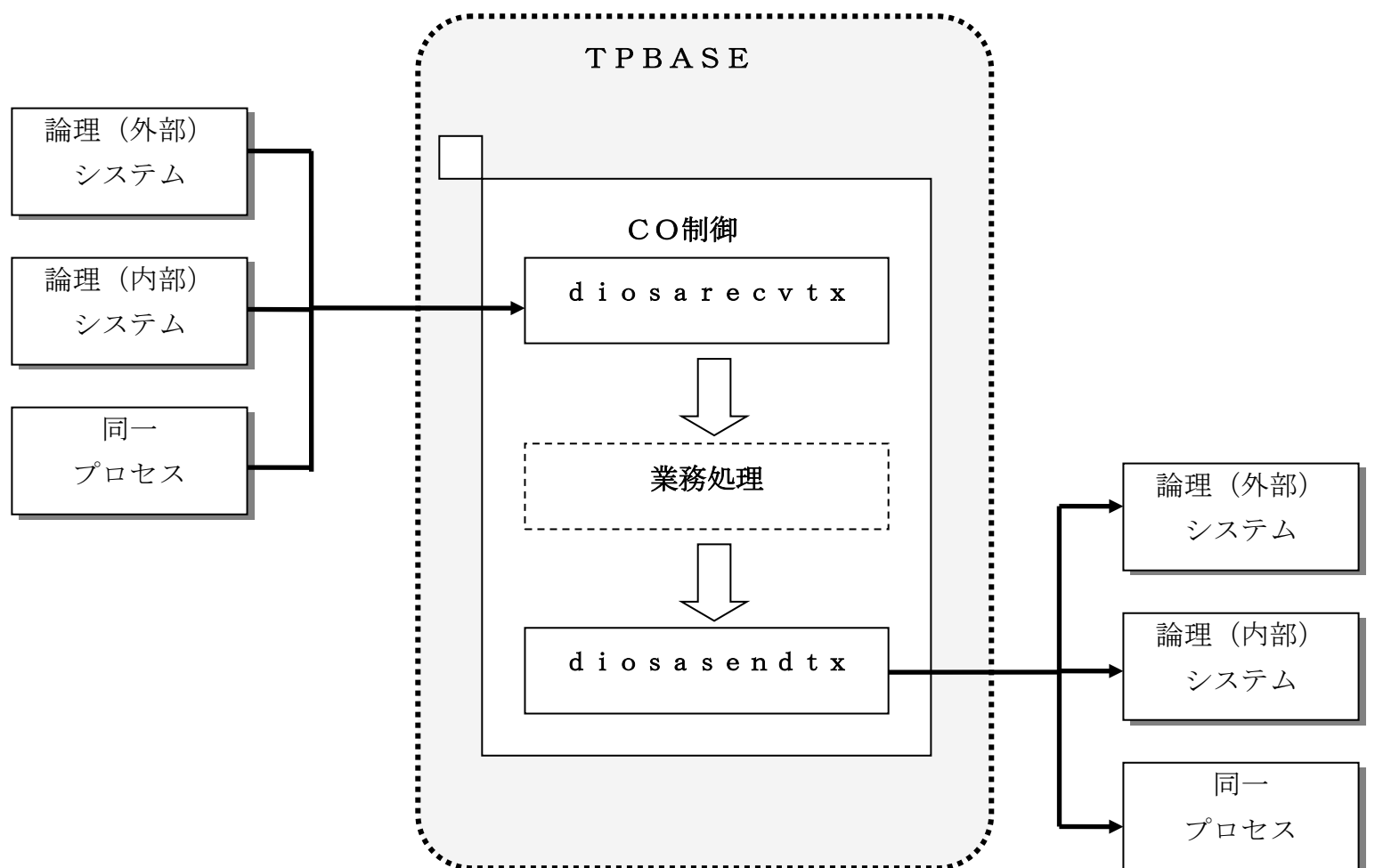
2.1.3 電文送受信機能

CO制御が提供するAPIを使い、論理システム間（外部論理システム宛）、論理システム内（内部論理ノード宛）等の電文を区別することなく送受信することができます。

コミットと同期をとって電文を送信する機能と即時に電文を送信する機能を選択することができます。コミットと同期をとって送信するモードを通常送信といい、即時に電文を送信するモードを強制送信と呼びます。通常送信はトランザクションを正常終了する、または後述するコミットAPI発行すると送信が実行され、アボート要求、または後述するロールバックAPIを発行するとキャンセルされます。強制送信はトランザクション状況、API発行有無に関係なく即時送信されます。そのため、アボート終了時に応答を送りたい場合等は強制送信を使うことができます。

電文受信には、diosarecvtxを電文送信には、diosasendtxと呼ばれるAPIを使います。

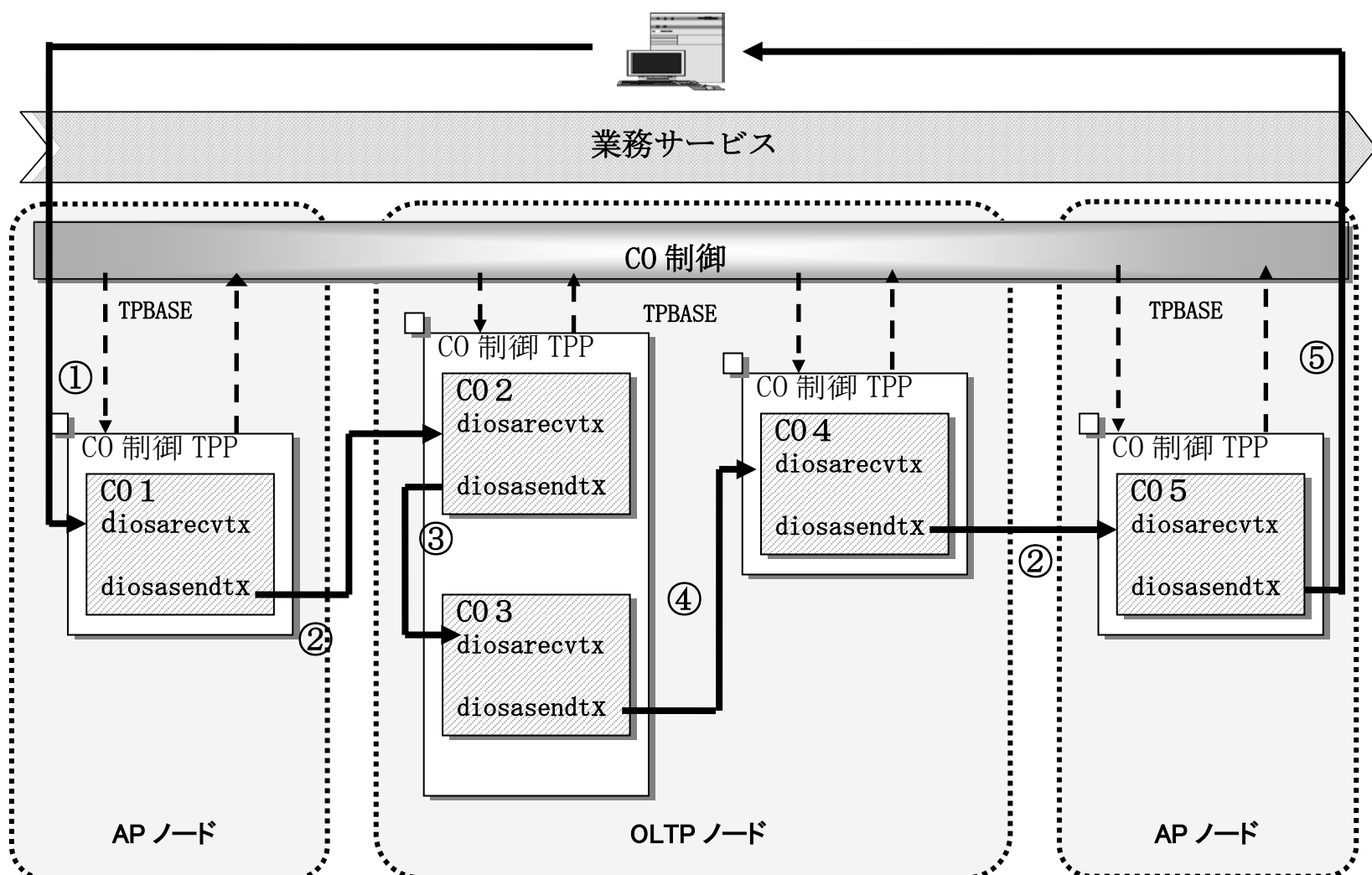
本APIはCO制御TPP配下でのみ使用することが可能です。



2.1.4 CO連携機能

論理（内部）システム内では、複数のCO群が連携して処理する方式で実装することができます。CO制御は3種類の連携方式（ノード間派生／連鎖／TPP間派生）を提供します。

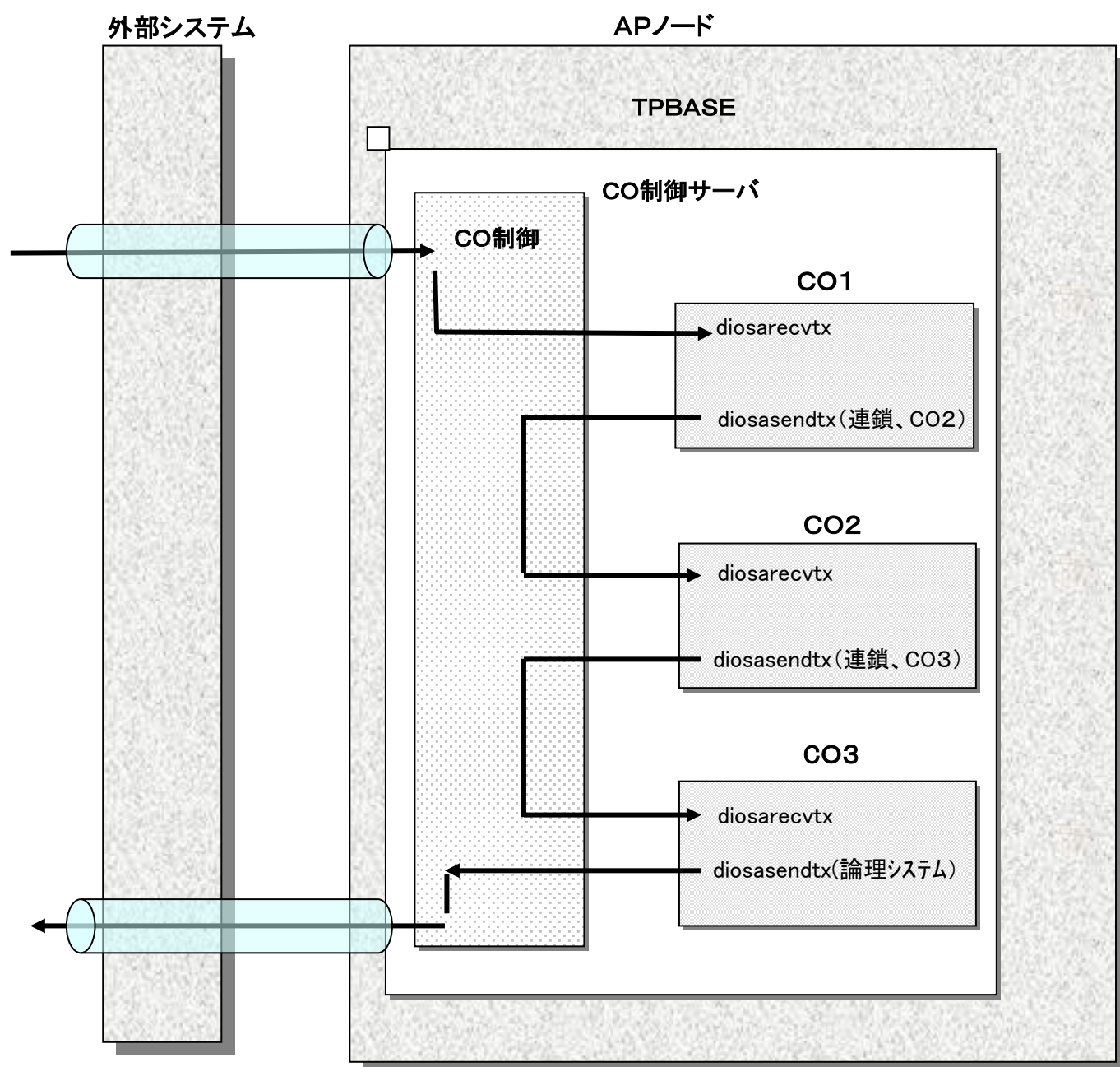
外部システムから要求を受信して、CO間連携で業務処理をおこない、外部システムに応答を返却するイメージ図を以下に記載します。



- ① 論理システム間（要求／応答）
- ② ノード間派生要求
- ③ （CO制御TPPプロセス内）連鎖要求
- ④ CO制御TPP間派生要求
- ⑤ 論理システム間（要求／応答）

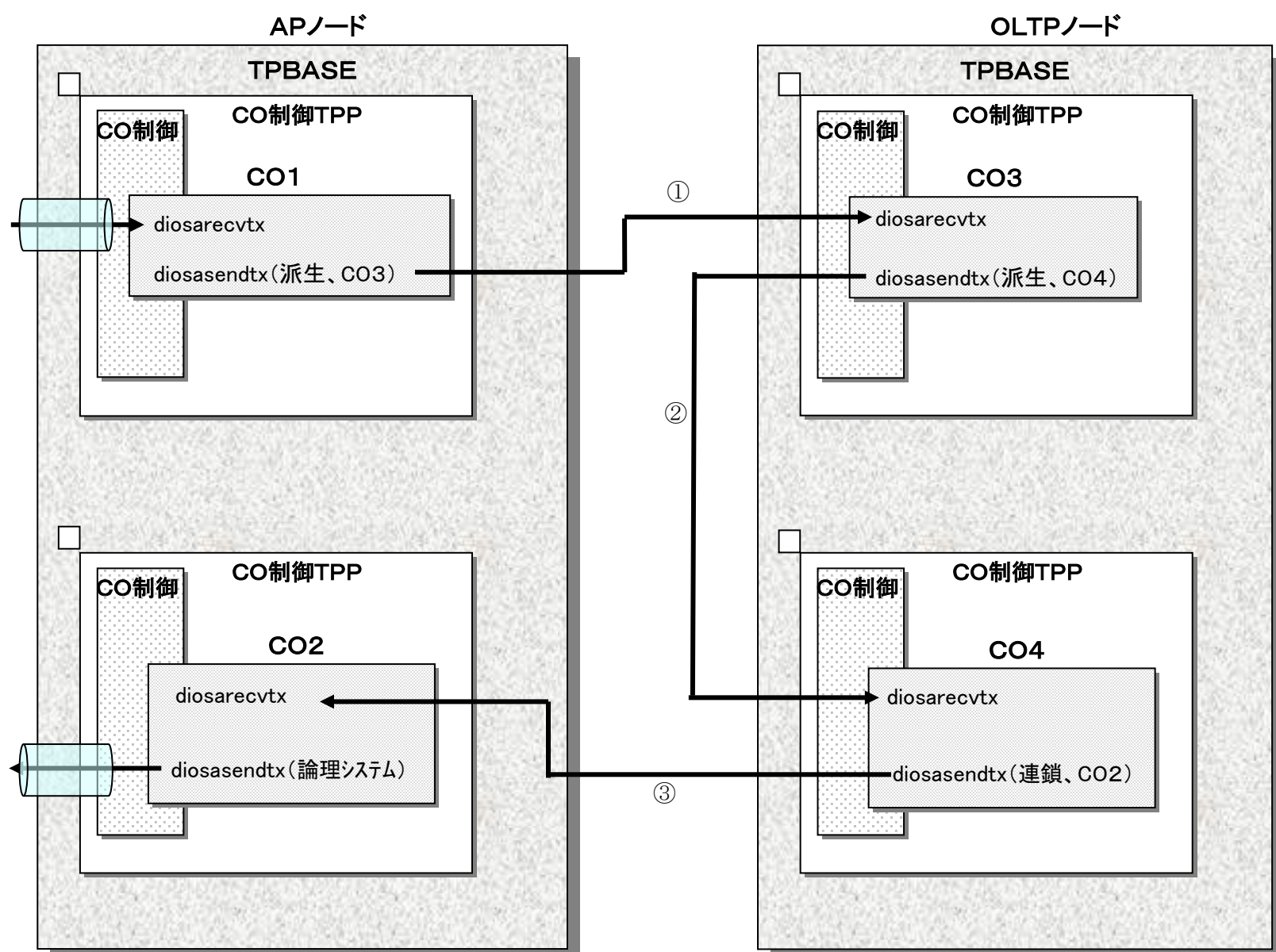
(1) **連鎖機能**

特定業務をおこなうCOと共通処理をおこなうCOを用意してある場合、特定業務COから共通処理をするCOを呼び出す必要があります。このような場合、連鎖機能を使うことで、処理元（特定業務）COと同一プロセス、同一トランザクション上でCOの呼び出しを行います。連鎖は全ての連鎖COの処理を1つのトランザクションとしてコミットすることも、現在処理中のCOの処理をロールバックしてから連鎖CO処理をおこなうこともできます。



(2) ノード間派生機能

業務処理の結果、他ノード（APノード→OLTPノード、OLTPノード→APノード）の別業務を起動したい場合、ノード間派生処理としてCO制御を動作させることができる。（下記図①、③）



(3) TPP間派生機能

業務処理の結果、同一TPBASE内の他CO制御を起動したい場合、TPP間派生処理としてCO制御を動作させることができる。（上図②）

2.1.5 コミット／ロールバック機能

CO制御はTAMに対するコミット、ロールバック処理を含むトランザクション処理をAPから隠蔽する。APはインメモリサーバが提供するTAMアクセスAPIを使い業務処理を作成するだけで、トランザクション開始、コミット、ロールバック等はCO制御が自動的にこなします。

コミット、ロールバックをAPが実行したい場合は、コミット出口とロールバック出口を使うことができます。（ただし、呼び出しはCO制御が制御します）

また、APがコミット、ロールバックを自由にできるようにAPIを使うこともできます。コミット出口、ロールバック出口が定義されている場合は、APIの延長で出口が呼び出されます。

(1) コミット／ロールバックの暗示的呼び出し機能

APからコミット、ロールバック処理を隠蔽します。APはトランザクション（CO連携）処理を正常終了することでコミットがおこなわれます。異常終了を要求するとロールバック処理がおこなわれ、異常終了の処理フェーズに自動的に遷移します。これを暗示的コミット、ロールバックと呼びます。コミット、ロールバックはインメモリサーバのコミット、ロールバックを呼び出します。

(2) コミット出口／ロールバック出口の呼び出し機能

APがコミット、ロールバック処理を実行したい場合、コミット出口、ロールバック出口を使うことができます。CO制御はインメモリサーバのコミット、ロールバックの代わりに、出口を呼び出します。

出口の呼び出しはCO制御が制御します、暗示的呼び出しと同じ呼び出しタイミングとなります。

(3) コミット／ロールバックの明示的呼び出し機能

APがコミット、ロールバックを自由にできるように、コミットAPI (diosacommit)、ロールバックAPI (diosarollback)を提供します。これを明示的コミット、ロールバックと呼びます。

ロールバック（明示）により戻される処理は最後にコミットAPIを行ったところまでとなります。APがロールバック後に再開すべき処理を決定できるように、CO制御はコミット時にコミット回数をカウントしており、COはdiosaucaによりコミット回数を参照することができます。

本APIは、後述のAPのループ／ストール監視機能におけるCPU時間と経過時間の監視をリセットするオプションを持ちます。本機能を使うことにより、長時間トランザクションを実現することができます。

2.1.6 アボート処理機能

A Pからアボート処理が要求された時は、ロールバックとアボート出口の呼び出しを行います。プログラム例外として扱うシグナルが発生した時も、アボート出口の呼び出しを行います。ロールバックはインメモリサーバに任せます。

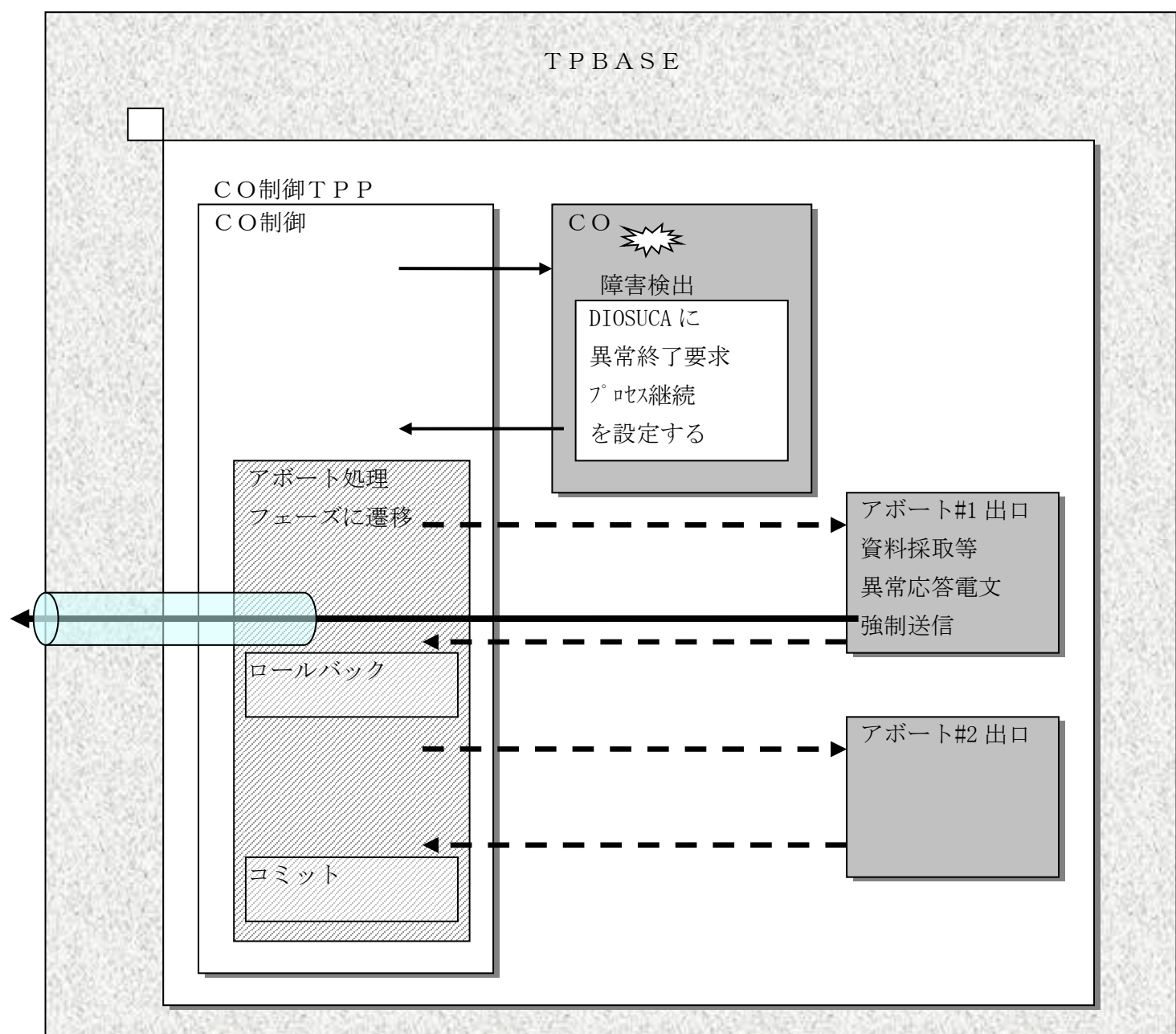
アボート出口には、アボート# 1 出口とアボート# 2 出口の2種類があります。

アボート# 1 出口はTPBASEのトランザクションキャンセル実行後に呼び出され、障害解析用の資料採取を目的としたものです。なお、アボート# 1 出口呼び出し後にはDBロールバックが行われます。異常応答を送信する場合は、アボート# 1 出口でA P要求時のアボートでも例外発生時のアボートでも共通的に行うことができます。この際の電文送信は強制送信のみ可能となります。

アボート# 2 出口はDBロールバック後に呼び出しますので、DBアクセスなどを行うことができます。

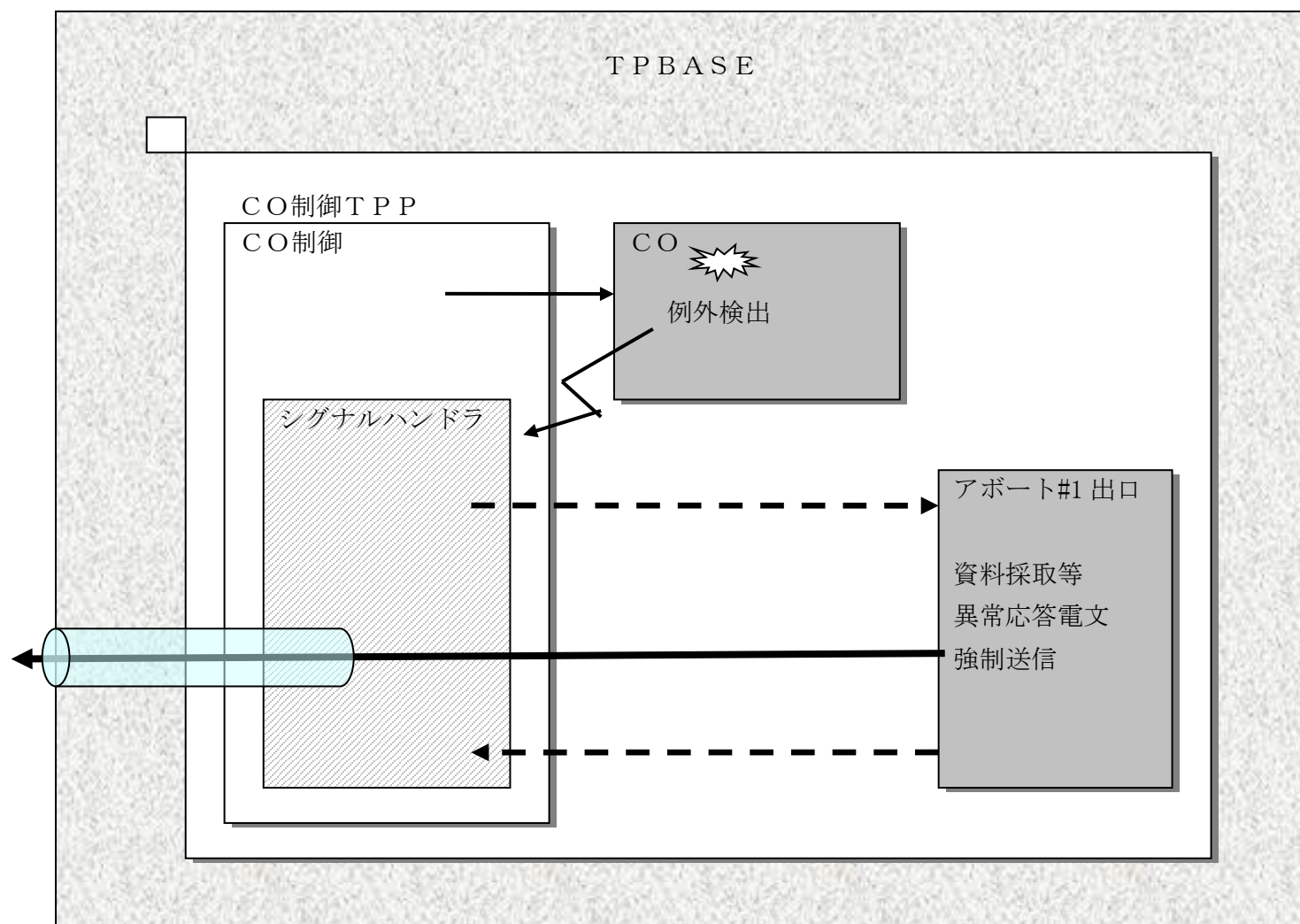
(1) A P要求時のアボート処理

A Pから「異常終了要求」が返された時、C O制御はアボート処理を行います。アボート処理後にプロセスの停止を行うか否かは、A Pから返される「異常終了要求」のコードにより決まります。



(2) 例外発生時のアボート処理

プログラム例外として扱うシグナルが発生した時、アボート# 1 出口はシグナルの発生したプロセス上で呼び出します。

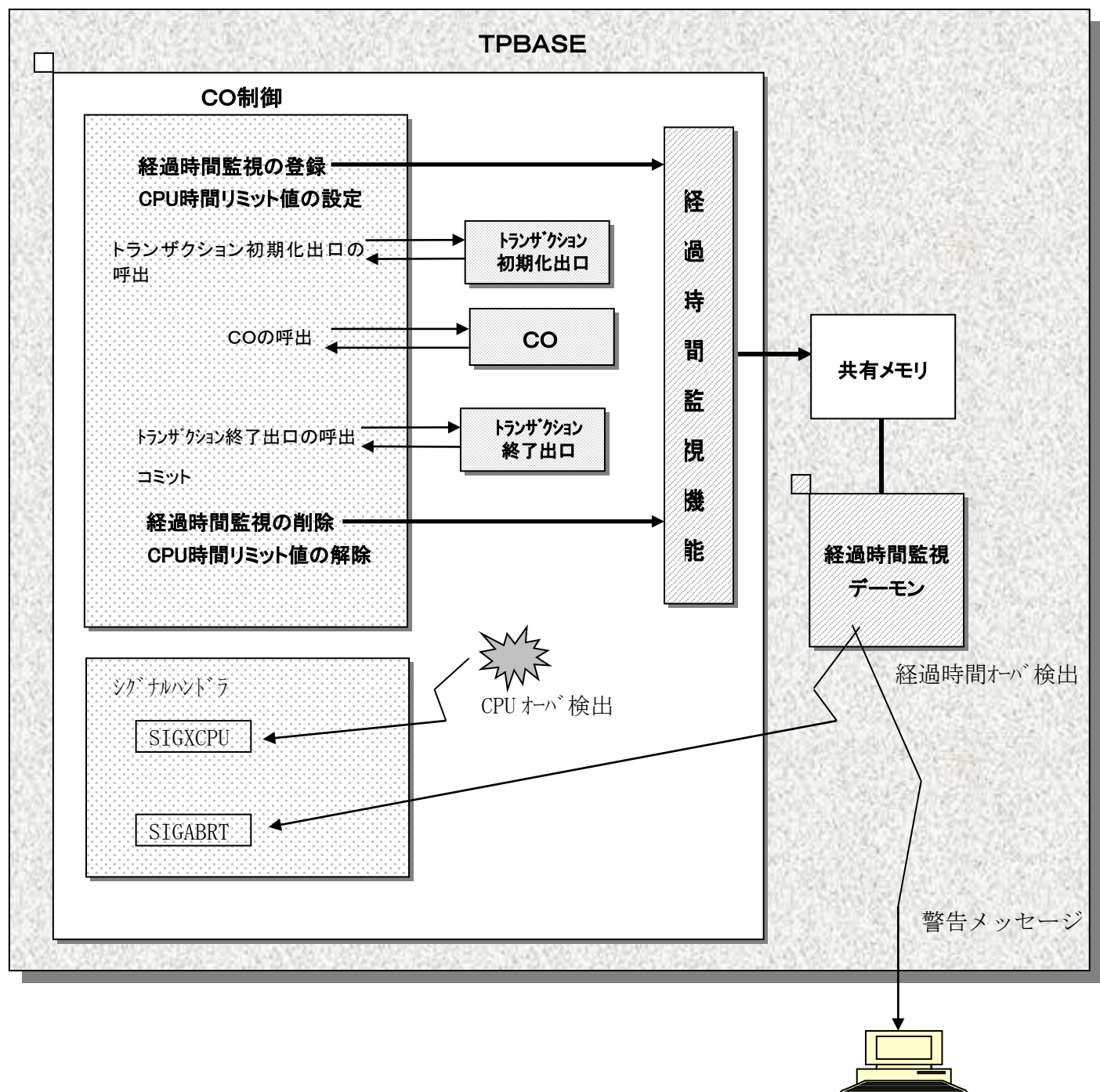


2.1.7 APのループ/ストール監視機能

CPU消費時間と経過時間を監視します。

経過時間超過時は、警告メッセージ出力か、シグナル例外を通知してプロセス終了を選択することができます。CPU時間超過時はシグナル例外を通知します。監視時間等は環境定義で定義することができます。また、コマンドを使い制限時間値等の変更をおこなうことができます。

経過時間の監視は、経過時間監視機能を利用して行います。

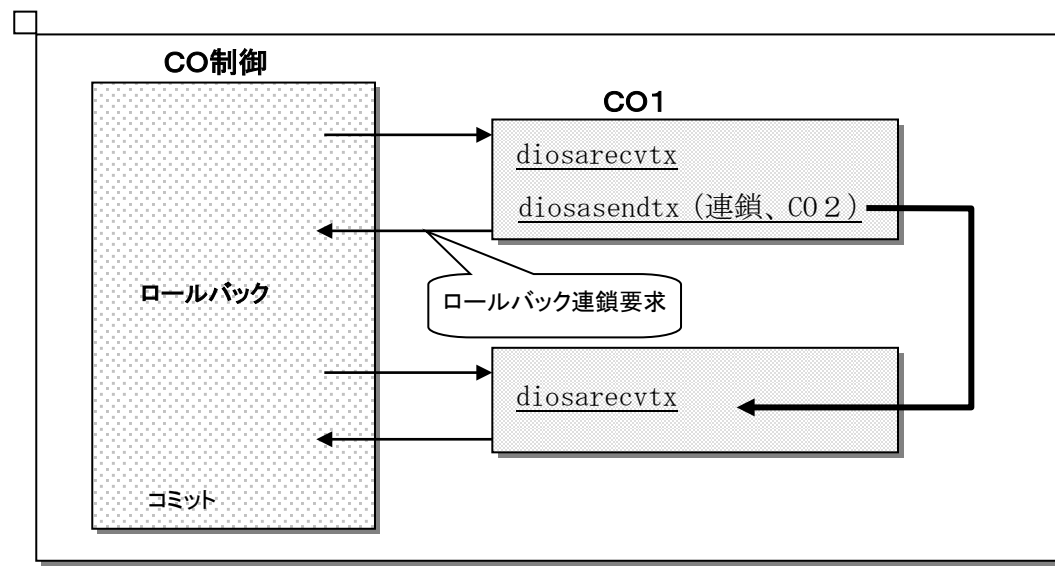


2.1.8 ロールバック連鎖機能

A Pから「ロールバック連鎖要求」が返された時、C O制御はロールバックを行ってから連鎖C Oを呼び出します。
A PはC O制御に戻る前に、送信A P Iにより連鎖電文を登録しておく必要があります。

本機能は、今までの処理をキャンセルして新たにT A Mアクセスを含む業務処理をプロセス、トランザクションを切り替えずに実行します。

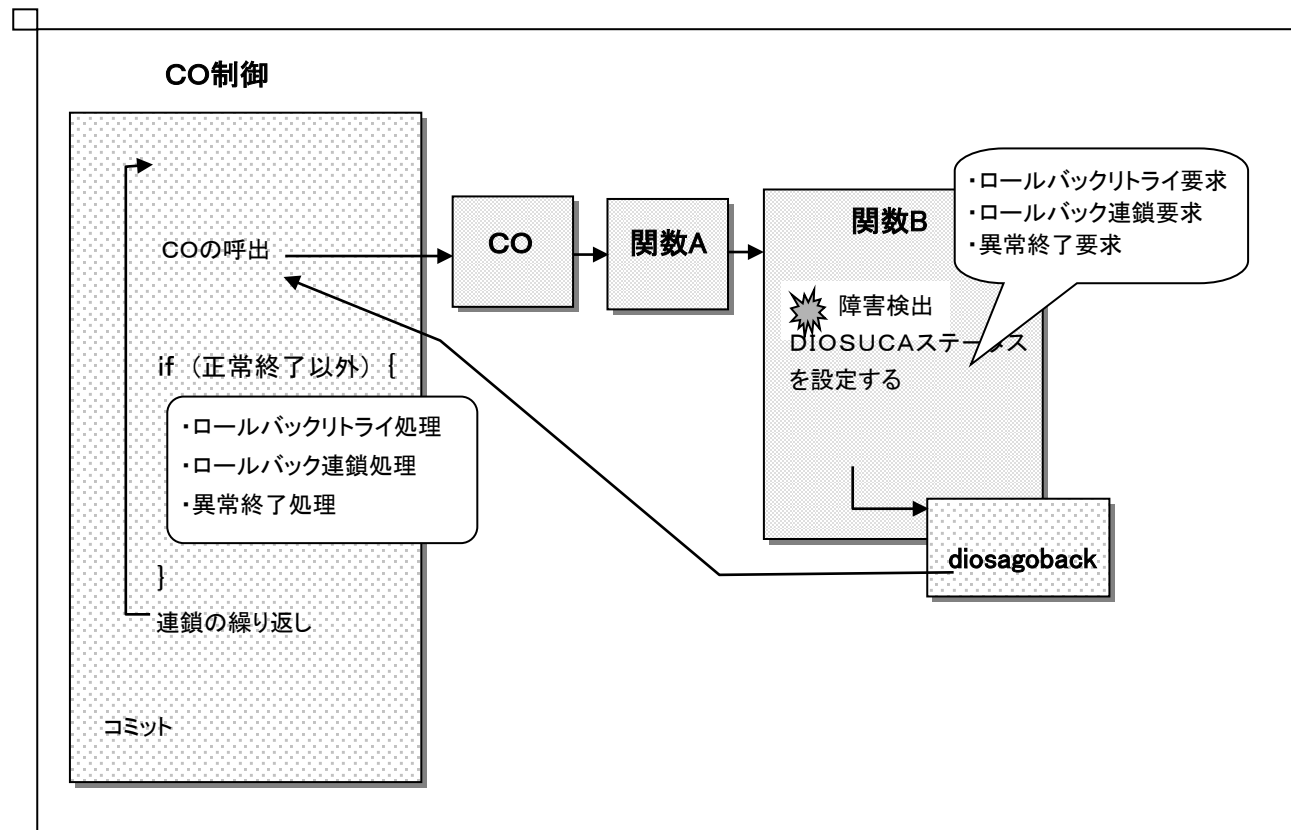
備考 ロールバック連鎖はアボート処理とは異なり、通常の連鎖処理と同様に正常終了扱いとなります。



2.1.9 CO制御への強制リターン機能

COBOLのGOBACK MAIN相当の機能です。下位層のプログラムから上位層のプログラムを跳び越してCO制御に直接リターンすることができます。強制リターン機能はdiosagobackというAPIで提供されます。

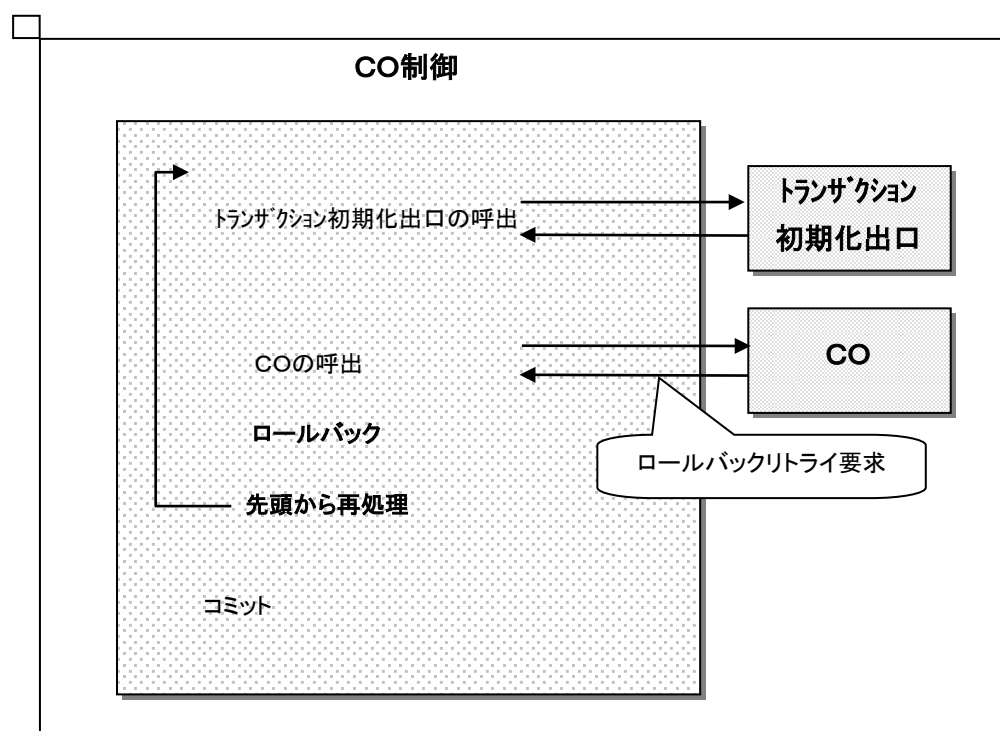
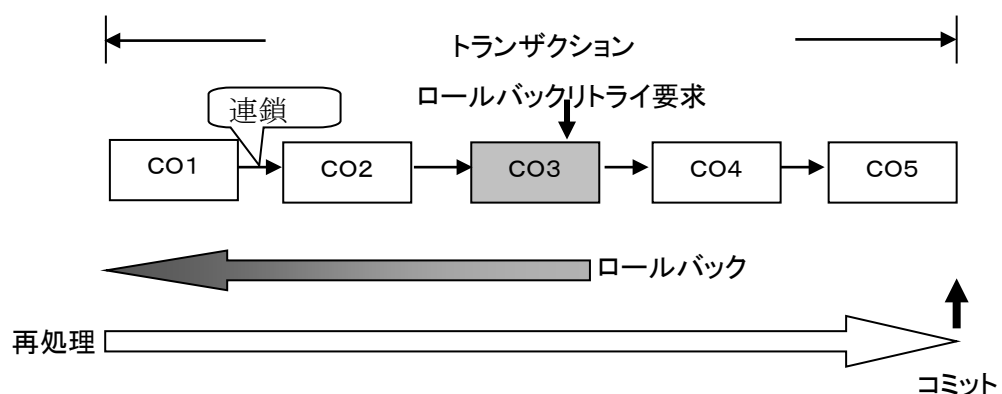
APは障害検出した関数の情報を上位関数に返却することなく、CO制御に直接制御を移行できます。(強制リターン前にDIOSAUCAに終了状態を決定しておきます)



2.1.10 ロールバックリトライ機能

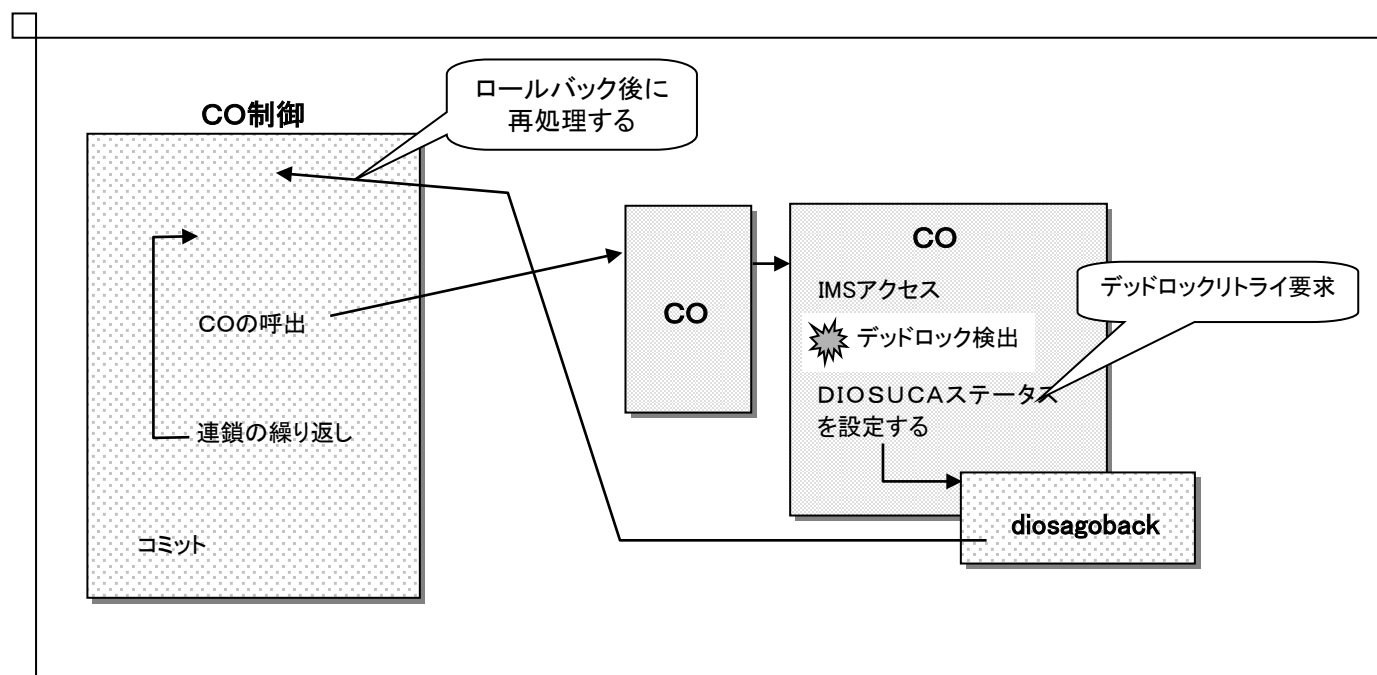
A Pから「ロールバックリトライ要求」が返された時、C O制御はロールバックを行い再処理します。

後述するデッドロックリトライもC O制御のリトライ処理対象となります。リトライ回数は環境定義で定義することができます。リトライは最低1回おこないます。また、ロールバックリトライ要求、デッドロックリトライ要求はそれぞれ1回のリトライ回数とカウントされます。



2.1.11 デッドロックリトライ機能

デッドロック発生時、ロールバック後に再処理します。APがTAMアクセスの結果デッドロックを検出した場合、DIOSUCAステータスに「デッドロックリトライ要求」を設定しCO制御へリターンします。DIOSA側でデッドロックを検出した時には、自動的にデッドロックリトライを行います。



2.1.12 電文保留機能

受信電文を一時的に退避しておき、後で処理する機能を提供します。

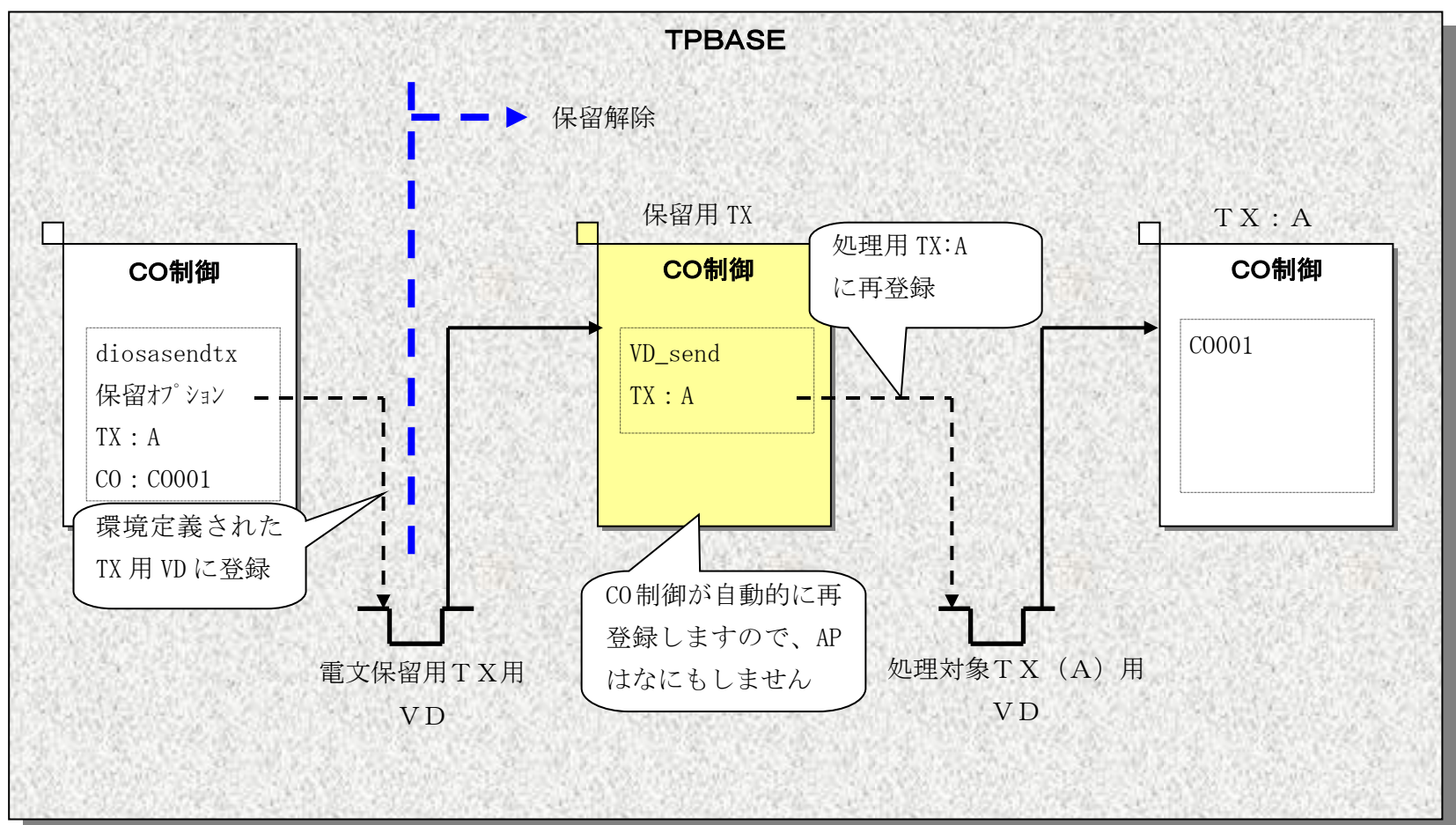
デッドロック、ロールバックリトライは直ちにリトライをしたい場合使う即時リトライ機能となり、電文保留機能はリトライ実行まで時間を要する場合に使う遅延リトライ機能となります。（以降の閉塞、閉塞解除という文言はTPBASEのトランザクション閉塞、閉塞解除の意味です）

退避する電文（保留電文と呼ぶ）は、電文保留用のトランザクションIDに対応したVDに送られます。保留する場合も diosasendtx を使用します。（保留用のトランザクションIDは、利用者が事前に閉塞しておく必要があります）

保留用トランザクションIDは環境定義で定義されますので、APが意識する必要はありません。APは保留が解除された後に処理すべきトランザクションIDを指定して diosasendtx をおこないます。

保留用トランザクションIDの閉塞を解除すると、TPBASEによってCO制御TPPが呼び出されます。CO制御は保留電文を受信すると、処理対象のトランザクションIDに対して保留電文を（通常の電文として）送信します。

CO制御は、電文保留用トランザクションIDの状態の制御のため、閉塞、閉塞解除、照会等をおこなう電文保留制御コマンドを提供します。なお、トランザクションの閉塞、閉塞解除は自動的におこなわれます。



2.2 バッチアプリケーション制御機能

バッチアプリケーション制御機能（バッチA P制御機能）はバッチジョブ環境において利用者プログラムの実行を支援することを目的とし、以下の機能を提供しています。

- オンラインで用いる利用者プログラムの呼び出し。
- D Bへの接続、コミット制御。
- 利用者による初期化处理および後処理を行うための利用者出口。

2.2.1 機能説明

(1) A P呼び出し機能

diosauca 領域をインタフェースとし、アプリケーション動的置換機能を通してC O／利用者出口を呼び出します。

(2) 稼動統計収集機能

稼動統計機能と連携して、C Oの稼動統計情報を収集します。

(3) ループ／ストール監視機能

C P U時間と経過時間を監視します。

(4) 実行レポート出力機能

バッチ処理終了時に実行レポートを出力します。

(5) D B自動制御機能

バッチ処理開始にD B（インメモリサーバまたはO r a c l e）コネクト、正常終了時にコミットとディスコネクト、異常終了時にロールバックとディスコネクトを行います。

(6) ロールバックリトライ機能

ロールバック後にC Oを再度呼び出します。

2.2.2 A P呼び出し機能

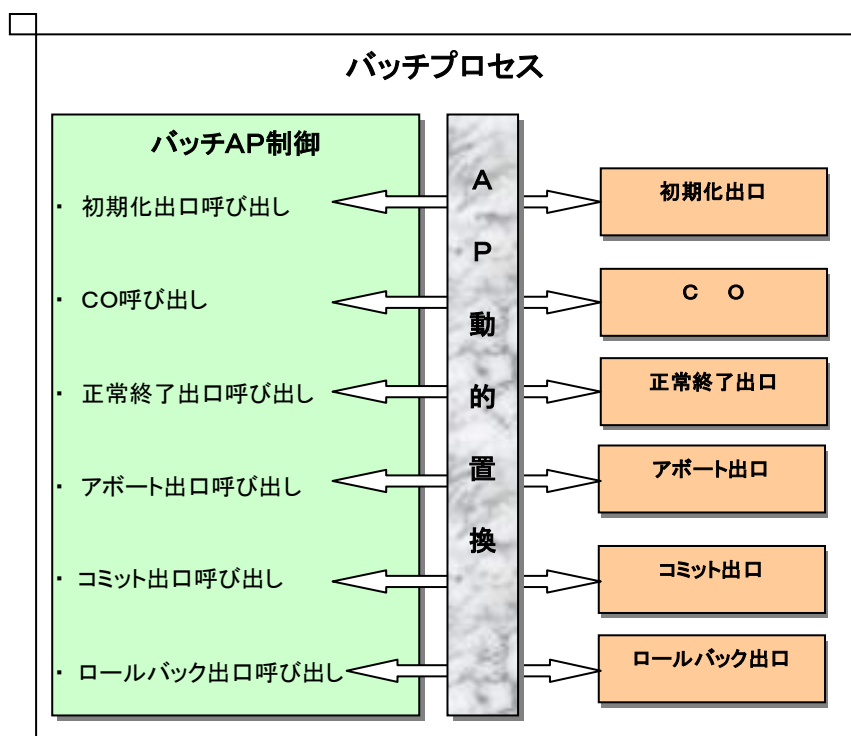
アプリケーション動的置換機能では関数名を解決してC O／利用者出口を動的に呼び出します。よって利用者はアプリケーションの物理的な配置を、アプリケーション動的置換機能の環境を設定する際に意識するだけでよく、バッチA P制御実行時にはパラメータに関数名を指定するだけで目的のC O／利用者出口を呼び出すことができます。

正常な流れでは「初期化出口→C O→正常終了出口」の順に呼び出しますが、C O／利用者出口から異常終了が返却された場合やバッチA P制御自身が異常を検出した時は、アボート出口を呼び出します。

また、シグナル発生時、コミット／ロールバック障害時にも、アボート出口を呼び出します。シグナル発生時にはD Bのロールバックは、バッチA P制御実行終了後にO r a c l eまたはインメモリサーバにより自動的にロールバックされることを期待し、バッチA P制御自身が明示的にロールバックを実行することはありません。S I G K I L Lなど捕捉不可能なシグナルが発生した場合は、アボート出口を呼び出しません。

C O／利用者出口呼び出しのインタフェースには、diosauca 領域を使用します。

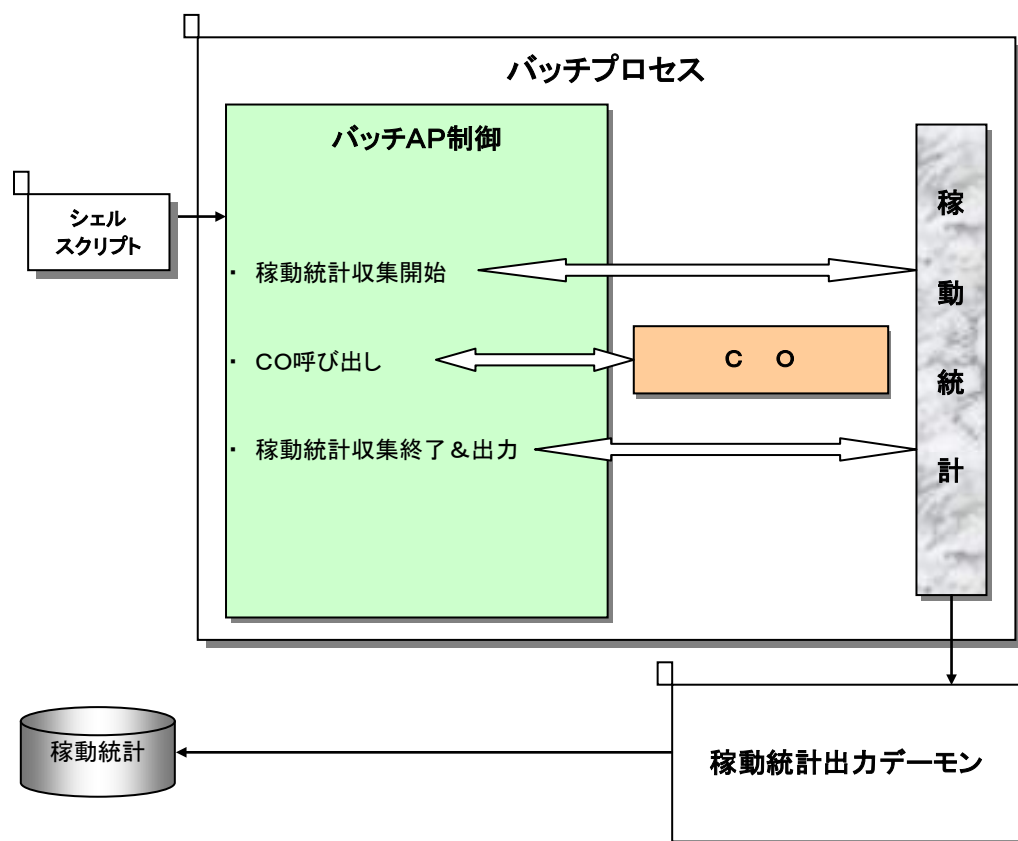
A P呼び出し機能の概要図を以下に示します。



2.2.3 稼動統計情報収集機能

稼動統計機能と連携し、C Oの稼動統計情報を収集します。当機能により本番環境における性能解析を支援します。

稼動統計情報の採取有無は、起動パラメータで指定できます。出力先は、稼動統計機能の環境定義で指定します。



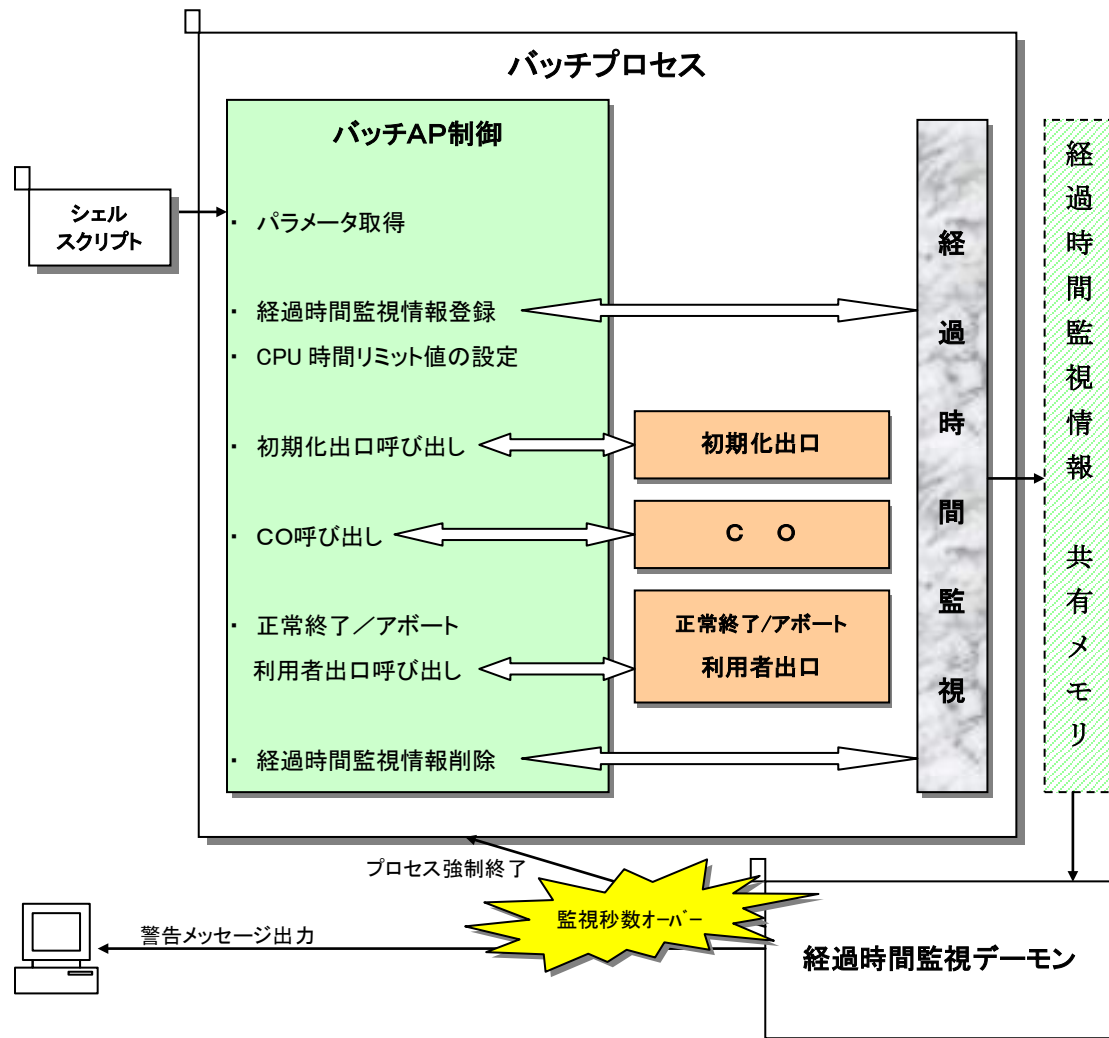
2.2.4 ループ/ストール監視機能

CPU時間と経過時間を監視します。

経過時間については、経過時間監視機能により監視を行います。

CPU時間が超過した時は、プロセスを強制終了します。また、経過時間が超過した時は、警告メッセージの出力またはプロセスの強制終了を行います。

起動パラメータでCPU時間制限値、経過時間制限値、経過時間リセット最大回数、経過時間超過時の動作を指定できます。



2.2.5 実行レポート出力機能

バッチ処理実行終了時に実行レポートを出力します。正常終了時に加え、利用者プログラムエラー時やシグナル発生時にも出力します。

実行レポートの出力内容は経過時間・CO関数名・呼び出し結果等です。

デフォルトでは出力なしであり、起動パラメータにより出力先ファイルを指定できます。既存ファイルを指定した場合は上書きします。標準出力/標準エラー出力への出力も可能です。

SIGKILLなど捕捉不可能なシグナルが発生した場合には、実行レポートは出力されません。

2.2.6 DB自動制御機能

DB自動制御機能には、TAM自動制御とOracle自動制御の2種類を提供しています。

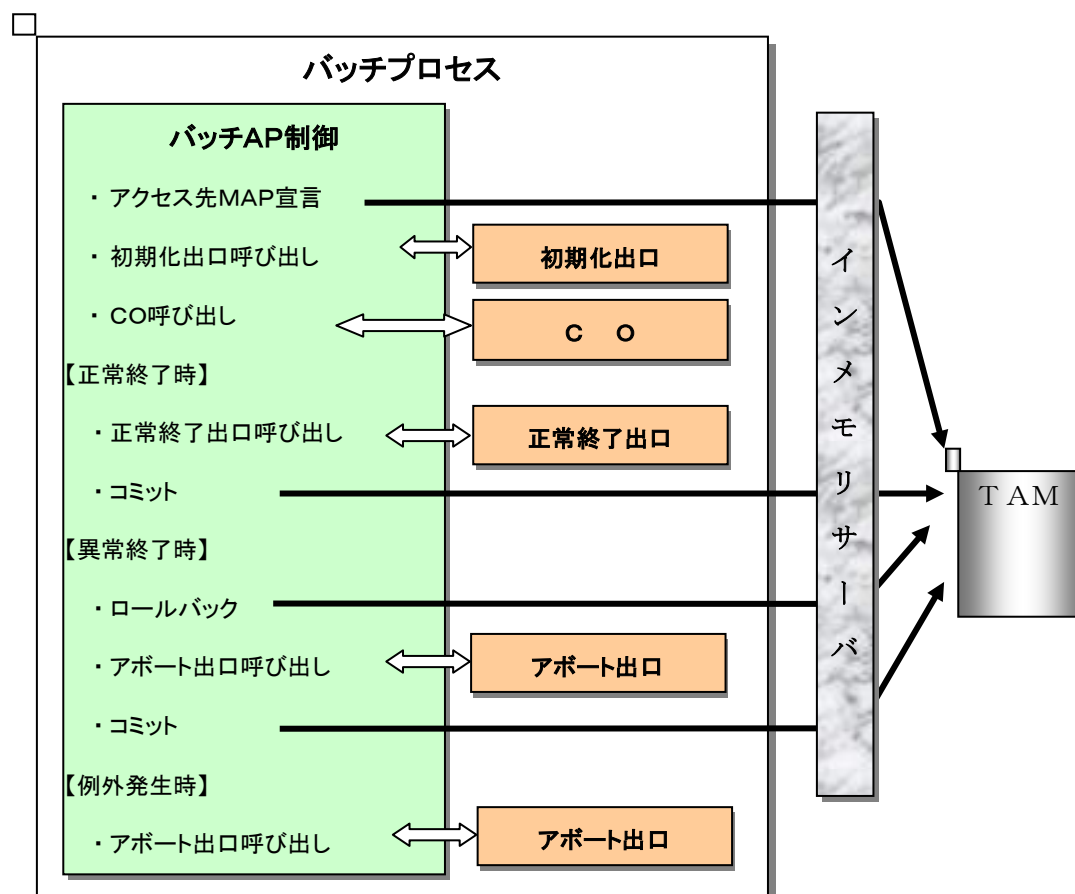
両方を同時に指定することはできません。指定した場合はパラメータエラーになります。

また、コミット出口／ロールバック出口を呼び出す場合は、バッチAP制御はコミット／ロールバックを行いません。

(1) TAM自動制御

起動パラメータに、`-m` パラメータ (MAP ID) を指定した場合に動作します。(MAP、MAP ID については、D IOSA/XTP メモリキャッシュ 利用の手引きを参照してください。)

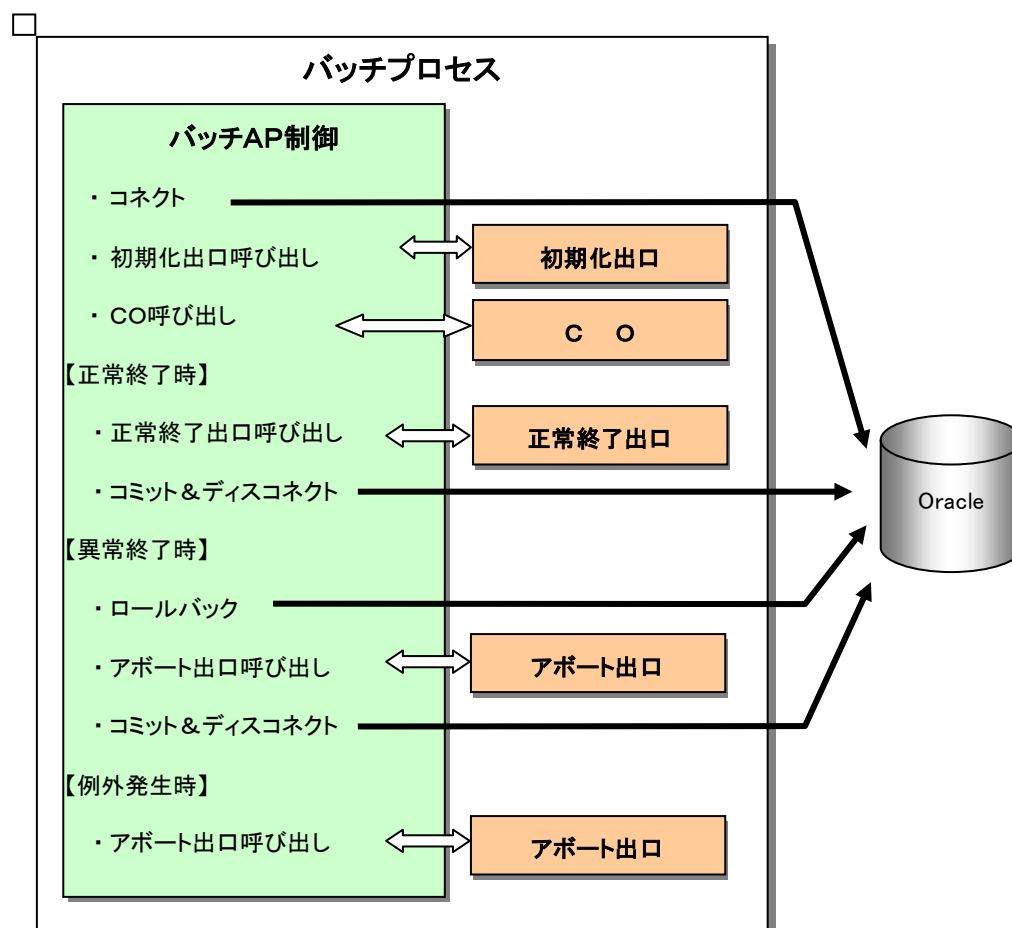
- プロセス起動時にインメモリサーバのオープンおよびアクセス先MAP宣言を行います。
- COが正常終了時、コミットを行います。
- 正常終了出口の呼出し後にコミットを行います。
- CO／利用者出口からの要求による異常終了時には、アボート出口の呼出し前にロールバックを行い、アボート出口の呼出し後にコミットを行います。(アボート出口でもTAMアクセスが可能です。)
- シグナル発生時は例外処理としてメッセージ出力やアボート出口呼出しを行います。明示的なロールバックを行いません。(バッチAP制御実行終了後にインメモリサーバにより自動的にロールバックされます。アボート出口で実行したTAM更新もロールバックされます。)
- コミット／ロールバックエラー時には、シグナル発生時と同じように例外処理を行います。



(2) Oracle自動制御

起動パラメータに、-d パラメータを指定した場合に動作します。

- プロセス起動時にOracleへコネクトします。
- COが正常終了時、コミットを行います。
- 正常終了出口の呼び出し後にコミット&ディスコネクトを行います。
- CO／利用者出口からの要求による異常終了時には、アボート出口の呼び出し前にロールバックを行い、アボート出口の呼び出し後にコミット&ディスコネクトを行います。（アボート出口でもOracleアクセスが可能です。）
- シグナル発生時は例外処理としてメッセージ出力やアボート出口呼び出しを行います。明示的なロールバックおよびディスコネクトを行いません。（バッチAP制御実行終了後にOracleにより自動的にロールバックされます。アボート出口で実行したOracle更新もロールバックされます。）
- コミット／ロールバックエラー時には、シグナル発生時と同じように例外処理を行います。



(3) ユーザによるDB制御

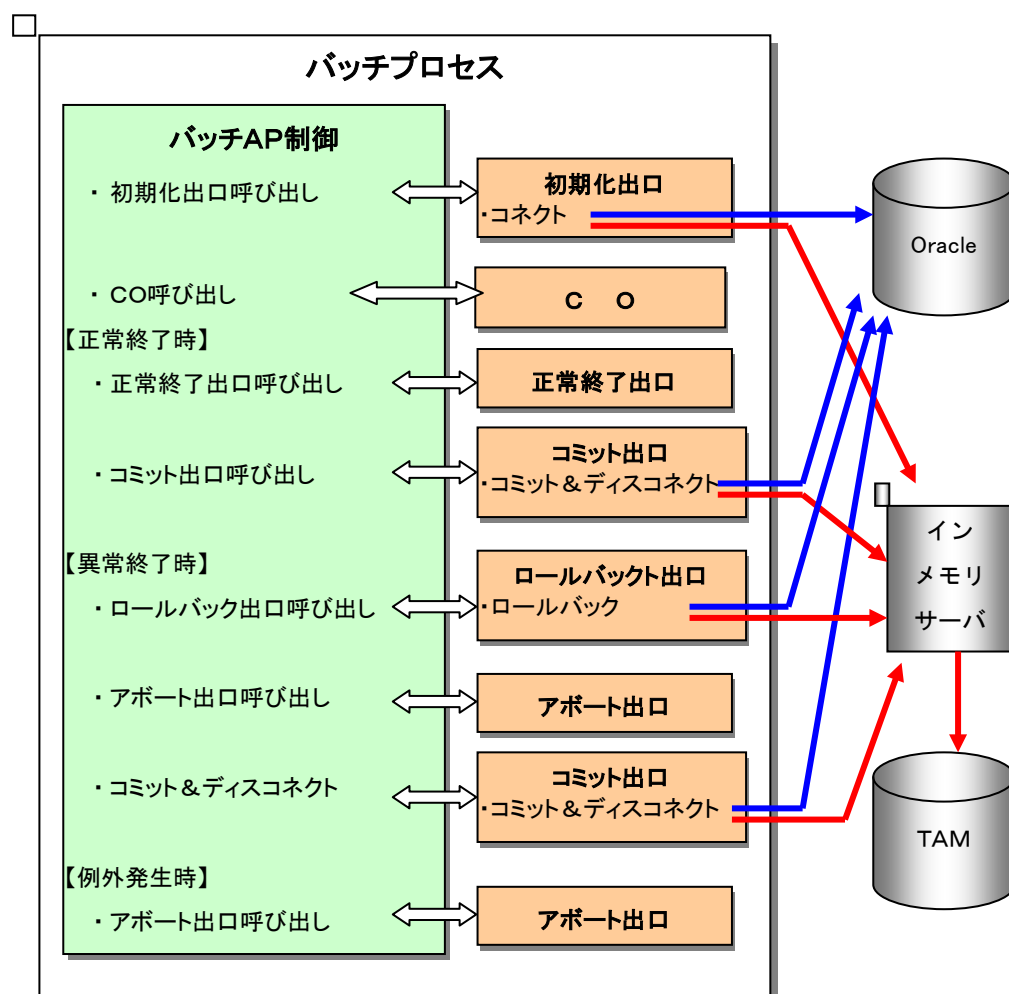
起動パラメータに、-C パラメータ（コミット出口）と-R パラメータ（ロールバック出口）を指定することにより、ユーザAPによるDB制御を行うことが可能です。

ユーザAPは、以下のように作成します。

- 初期化出口（またはCOの先頭）で、接続先を決定し、Oracleへのコネクトまたはインメモリサーバへアクセス先MAP宣言を行います。
- コミット出口は、インメモリサーバのコミット、またはOracleのコミット&ディスコネクトを行います。
- ロールバック出口は、インメモリサーバまたはOracleのロールバックを行います。

ユーザAPによりDB制御を行う場合、バッチAP制御は以下のように実行されます。

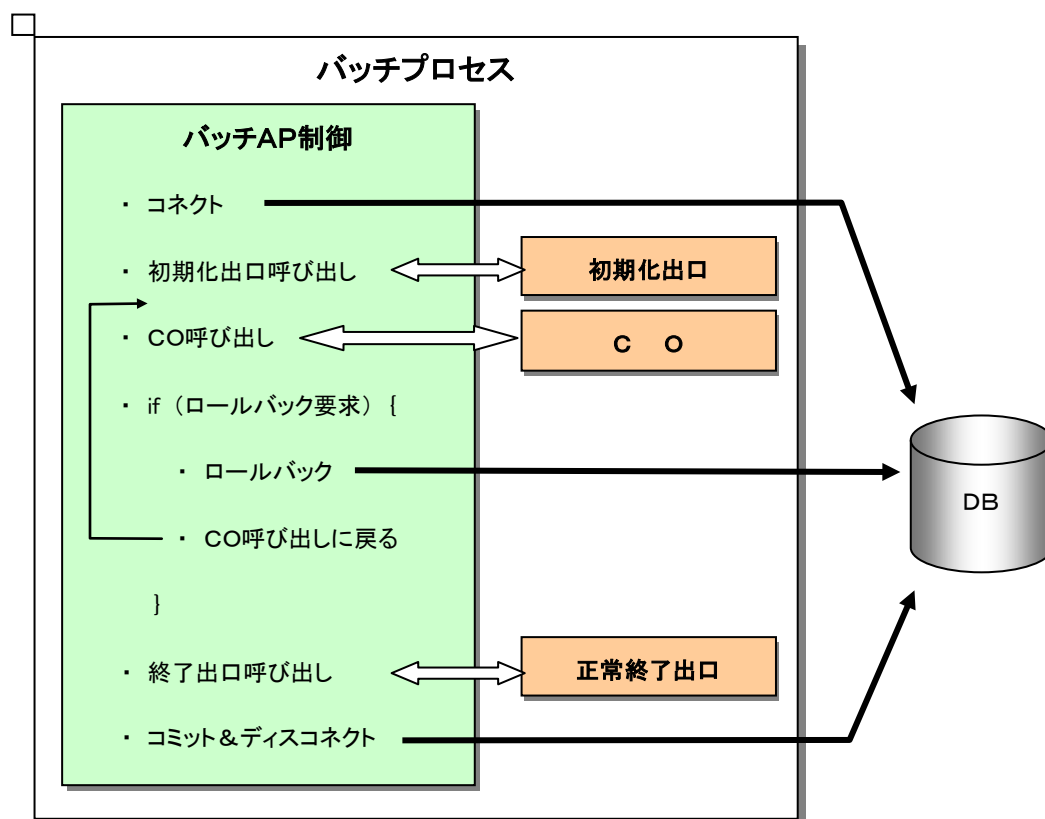
- プロセス起動時にインメモリサーバのオープンを行った後、初期化出口を呼び出します。
- COが正常終了時、正常終了出口の呼出し後にコミット出口呼び出しを行います。
- CO／利用者出口からの要求による異常終了時は、アボート出口の呼び出し前にロールバック出口を呼び出し、アボート出口の呼び出し後にコミット出口を呼び出します。（アボート出口でもDBアクセスが可能です。）
- シグナル発生時は例外処理としてメッセージ出力やアボート出口呼び出しを行います。ロールバック出口の呼び出しを行いません。（バッチAP制御実行終了後にインメモリサーバまたはOracleにより自動的にロールバックされます。アボート出口でのDB更新もロールバックされます。）
- コミット／ロールバックエラー時には、シグナル発生時と同じように例外処理を行います。



2.2.7 ロールバックリトライ機能

COから「ロールバックリトライ要求」が返された場合に、ロールバックを行い、再度COを呼び出します。

ロールバックリトライ機能を利用するためには、起動時に、-b パラメータでロールバックリトライの最大回数を指定する必要があります。

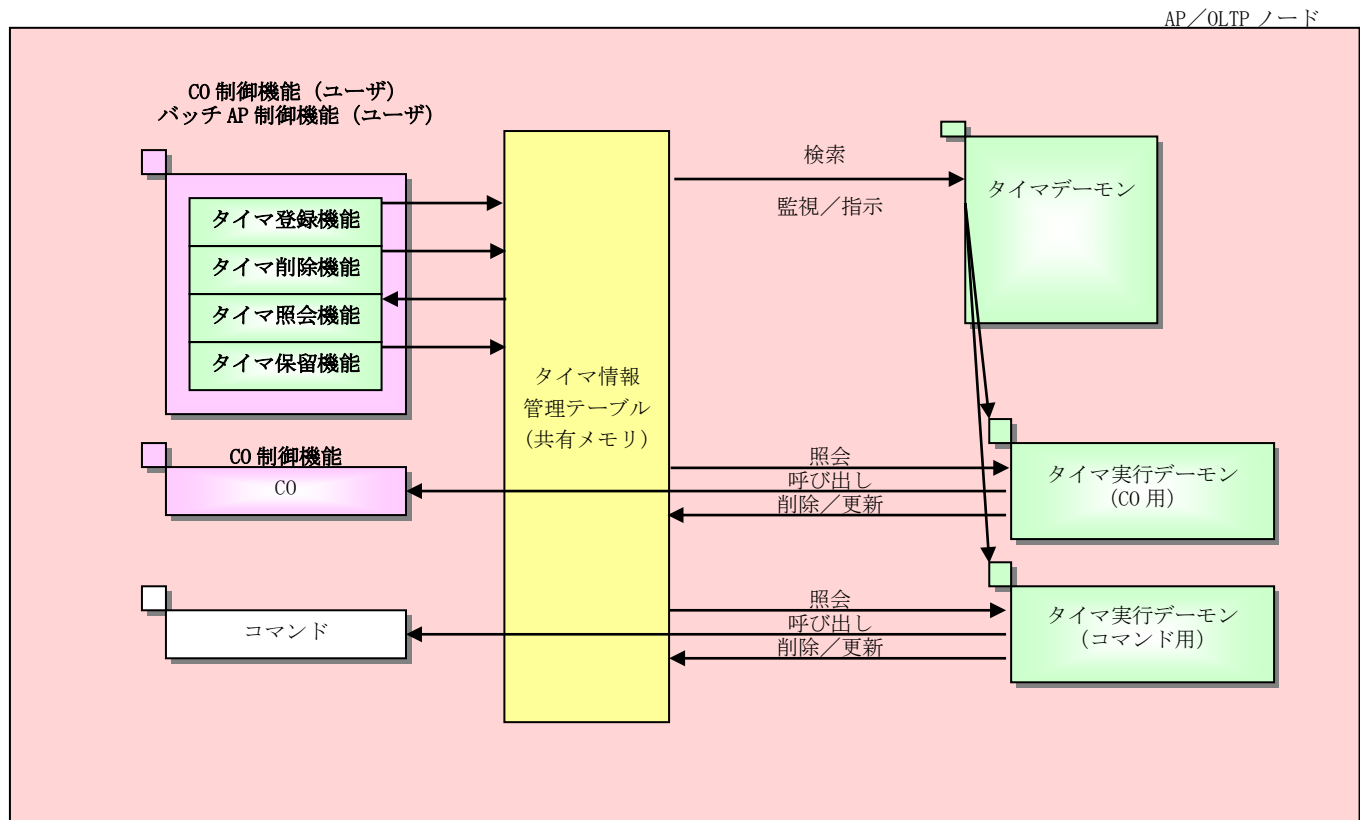


2.3 タイマ制御機能

タイマ制御機能では利用者任意のタイミングでC O呼び出しや実行コマンドを行う機能を提供します。

実行の形式としては、時刻指定、インターバル指定、即時指定の3種類があります。タイマ制御機能は共有メモリでタイマの情報を管理し、実行されるノードはタイマ登録ノード固定となります。

タイマ情報は利用者任意の文字列（16バイト）をタイマIDとして一意に管理し、以下に示す登録や削除、照会等を行うことができます。



- タイマ登録、削除、照会、保留機能により任意のタイマ情報をタイマ情報管理テーブルに登録、削除、照会、更新を行います。
- タイマデーモンは、タイマ情報管理テーブルから、タイマ実行時刻を過ぎており保留されていないタイマ情報を検索し、タイマ実行デーモンにタイマ実行の指示を通知します。
- 指示を受けたタイマ実行デーモンは、タイマ情報管理テーブルからタイマ情報を取得し、C O呼び出し、またはコマンド実行を行います。呼び出し後、タイマ情報管理テーブルのタイマ情報を削除、または更新を行います。
- タイマ実行デーモンは一定時間待機し、次イベントがなければ停止します。

2.3.1 機能説明

(1) タイマ登録機能

- A P I、コマンド、S Gからタイマ登録を行います。
- 登録時に指定するタイマ I Dをキーにして管理します。
- タイマの最大登録数は環境変数にて指定します。
- 同一のタイマ I Dの登録は、既存のタイマ情報を更新します。A P Iまたはコマンドでタイマ I Dの重複登録のチェック有りを指定することにより、重複している場合の戻り値を警告終了とすることができます。（重複登録のチェック無しの指定では、重複している場合でも戻り値は正常終了です。）
- 同一タイマ I Dによるタイマ要求の登録をした場合は、既存のタイマ要求が取り消され新しいタイマ要求のみ登録となります。つまり、既存のタイマ要求の取り消しを行ったあとに、タイマ要求の登録を行う場合と同じ結果になります。

(2) タイマ削除機能

- コマンドまたはA P Iからタイマ情報の削除を行います。
- 削除対象の指定方法には、正規表現が使用できます。

(3) タイマ照会機能

- コマンドまたはA P Iからタイマ情報の照会を行います。
- コマンドによる照会対象はタイマ I Dで指定し、正規表現が使用できます。
- A P Iによる照会は、先頭からの順次検索とタイマ I D指定のタイマエントリ照会が可能です。
- A P Iによる照会は、正規表現は使用できません。

(4) タイマ保留・保留解除機能

- コマンドまたはA P Iからタイマ情報の保留、保留解除を行います。
- 保留機能により任意のタイマ情報の実行を、タイマ情報を削除することなく一時停止することができます。
- 保留解除機能により保留されたタイマ情報を解除し、再びタイマを実行することができます。
- 保留状態でタイマの指定時刻または指定時間後を迎えた場合、保留解除後にタイマが実行されます。

(5) タイマ実行機能

- 登録されたタイマ情報を基に、指定時刻または指定時間後にタイマを実行します。
- 実行プロセスは個々に起動するため、同一時刻に複数のタイマ実行を行えます。
- 同時実行プロセス数の上限は環境変数にて指定します。

(6) タイマ監視機能

- 実行したタイマについて一定時間以上応答が無い場合、要求中のタイマをエラーとして削除します。
- 応答を待つ時間については環境変数にて指定します。

2.4 メモリ管理機能

2.4.1 機能説明

メモリ管理機能はユーザアプリケーションで使用するメモリを一括で管理し、以下の機能を提供します。

- 予めメモリ使用上限値を設定して、メモリの使い過ぎを防止することができます。
- C0 制御のトランザクション初期化時に前回のトランザクションで確保（解放漏れ）したプロセスメモリを自動解放することができます。
- 保護属性共有メモリを確保して、不正なメモリ更新はシグナルにより防止することができます。

C0 制御機能管理下およびバッチアプリケーション制御機能管理下のアプリケーションプログラムにおいて、動的にデータ展開を行う場合には、メモリ管理機能が提供する共有メモリおよびプロセスメモリの動的な領域割り当て/解放機能等を利用する事ができます。

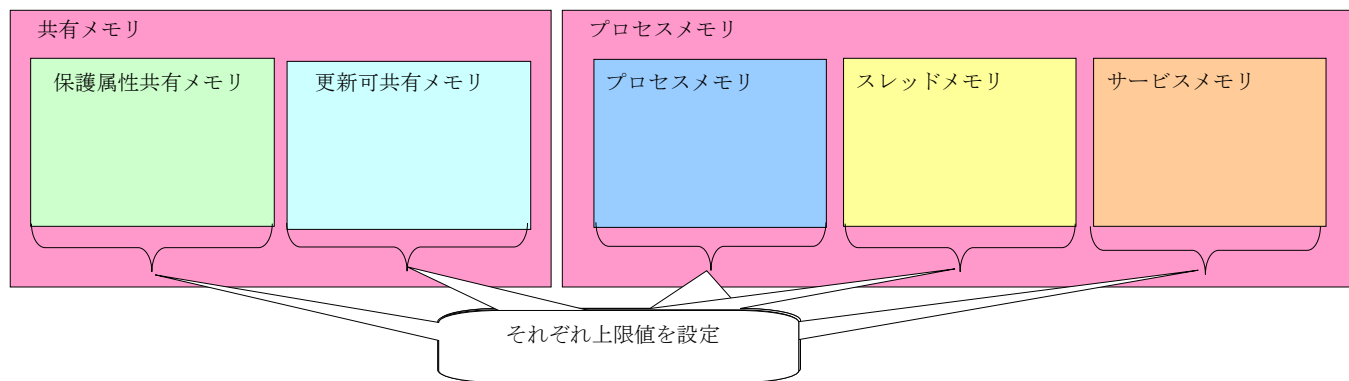
メモリ管理機能は、共有メモリおよびプロセスメモリの割り当て/解放、プロセスが異常終了した場合にメモリ情報をダンプファイルに出力、メモリ情報/ダンプファイル内容照会を行う機能等を提供します。

メモリ管理機能ではアプリケーションが使用するメモリとして保護属性共有メモリ、更新可共有メモリ、プロセスメモリを生成し、管理します。

ユーザがメモリ管理機能を利用して、上記メモリの割り当て等を行う時、以下のメモリ種別を指定することができます。保護属性共有メモリは、利用者が直接更新できない共有メモリで、保護属性領域書き込み機能(API 提供)を利用し、共有メモリにデータをコピーすることで不正なメモリ更新を防止します。

種別			説明
共有メモリ	ノード内	保護属性	利用者がメモリ割り当てで確保した領域毎に解放する保護属性（利用者が直接更新できない）共有メモリ
		更新可	利用者がメモリ割り当てで確保した領域毎に解放する更新可（利用者が直接更新できる）共有メモリ
プロセスメモリ	プロセス内	一括解放対象外	利用者がメモリ割り当てで確保した領域毎に解放するプロセスメモリ
	スレッド内	一括解放対象外	利用者がメモリ割り当てで確保した領域毎に解放するプロセスメモリ
	サービス内	一括解放対象	C0 制御のトランザクション初期化により一括解放するプロセスメモリ（利用者がメモリ割り当てで確保した領域毎に解放することも可能）

ユーザが確保する利用者用の共有メモリ(保護属性、更新可)は、同じ共有メモリ(保護属性、更新可)内で割り当てます。共有メモリサイズは上限値を設けて、共有メモリ最大サイズはその上限値の合計値となります。



メモリ管理機能はユーザが使用するメモリサイズに上限値を設定することで AP の不正割り当て要求により、共有メモリ/プロセスメモリが他の利用可能なメモリを食いつぶすことを防止できます。

提供する機能は以下の通りです。

(1) メモリ領域操作機能

- メモリ領域割り当て機能(API 提供)
ユーザ AP から呼び出され、必要なサイズのメモリを割り当てます。
- メモリ領域再割り当て機能(API 提供)
ユーザ AP から呼び出され、メモリ領域割り当て機能より割り当てたメモリ領域のサイズを変更し、メモリ領域を再度割り当てます。メモリの内容は引き継ぎます。
- メモリ領域解放機能(API 提供)
ユーザ AP から呼び出され、メモリ割り当て機能より割り当てたメモリ領域中の指定メモリを解放します。一括解放を指定した場合、C0 制御のトランザクション初期化時に一括解放対象のプロセスメモリを全て解放します。
- メモリ領域アドレス取得機能(API 提供)
ユーザ AP から呼び出され、メモリ割り当て機能で指定したメモリ識別子を指定して、そのメモリのアドレスを取得します。メモリ割り当て機能でメモリ識別子を省略した場合、本機能は利用できません。
- 保護属性領域書き込み機能(API 提供)
ユーザ AP から呼び出され、データを保護属性の共有メモリに書き込みます。
- 共有メモリアドレス変換機能(API 提供)
ユーザ AP から呼び出され、メモリ割り当て機能で割り当てた共有メモリのアドレスをオフセットへ、オフセットをアドレスへ変換します。

(2) メモリ共通領域設定・取得機能

共通領域アドレス設定/取得機能はユーザが共有メモリを割り当てる際、識別子を指定していない且つ別のプロセスへ共有メモリアドレスを渡す場合に利用します。共通領域はユーザ開始処理呼び出し機能にて共通領域(共有メモリ)を割り当て/設定し、別のプロセスで取得して参照することができます。

- 共通領域設定機能(API 提供)
ユーザ AP から呼び出され、メモリ割り当て機能より割り当てた共通領域アドレス(共有メモリ)を共有メモリ管理領域へ設定する場合に、共通領域設定機能を利用します。インタフェースとして(`diosagapptrset()`)を提供します。共有メモリ管理領域に設定できるポインタは論理ノード内で1つであり、共通領域ポインタの設定が複数回行われた場合には、最後に設定したポインタのみをメモリ管理機能

が管理します。

- 共通領域取得機能(API 提供)

ユーザ AP から呼び出され、共通領域取得機能を利用して共有メモリ管理領域から共通領域アドレスを取得します。インタフェースとして (`diosagapptrget()`) を提供します。なお、C0 制御やバッチ AP 制御から呼ばれる C0 では、C0・利用者出口パラメータ (`diosauca`) を参照することにより、共通領域アドレスを取得することができます。

(3) **コマンド機能**

- メモリ状態/ダンプファイル内容表示機能(コマンド提供)

ユーザの端末から実行され、共有メモリ情報およびアボートダンプファイルからメモリ情報を表示します。

- メモリー一括解放機能(コマンド提供)

割り当て領域の共有メモリを一括で解放します。

2.4.2 **環境設定**

アプリケーションプログラムが利用できるメモリ管理領域は、事前に環境定義に設定する必要があります。メモリ種別毎に初期サイズ、拡張サイズ、最大サイズを設定します。

初期サイズ：メモリ管理機能初期処理時に生成するサイズ

拡張サイズ：メモリ領域不足時に動的に拡張するサイズ

最大サイズ：動的にメモリ領域を拡張する場合の上限値

最大サイズを超えてセグメントの生成は行われない (`diosamalloc()` で `DIOSA_ENOBUFS(-7)` が返却される)。

2.5 ロック制御機能

2.5.1 機能説明

ロック制御機能は、ユーザアプリケーションが排他制御を行うための、ロック操作を行うインタフェースを提供する機能です。

ロックの有効範囲は、使用用途に応じて論理システム内と論理ノード内の2つのパターンから選択できます。本機能では、指定されたロックの有効範囲をもとに、ロック制御提供機能によって最適なロック操作を選びます。本機能が提供するロック操作は、システム関数 `fcntl()` を利用したファイル型ロック、およびデータベースに含まれる排他制御支援パッケージ (DBMS_LOCK) を利用した DB 型ロックの2種類です。

ロック範囲とロック操作の対応を以下に示します。

ロックの有効範囲	対応するロック操作
論理ノード内	ファイル型ロック
論理システム内	DB 型ロック

利用者は任意の時点でロックの取得／解放を行うことができます。DIOSA 提供のコミット／ロールバックに連動してロック制御は自動的に解放します。

ロック取得時の指定として、ロック操作に関わらず、以下を選択することができます。

- ・ロックモード：占有／共有ロックの選択
- ・ウェイトオプション：資源がロックされていた場合の動作として、解放されるまで待つ／エラーリターンの選択

ロック操作による機能差異について以下に示します。

(1) ファイル型ロック制御機能

- ・ロックの識別子をファイルセグメントに割り当てることでロック制御を行います。
- ・同じロック ID を使用しても論理ノードを跨ったロック制御は行えません。
- ・`fcntl` 動作中にシグナルを受信すると、エラーを返却します。
- ・ファイル型ロック同士のデッドロックは検出できますが、DB 型との相互デッドロック検出機能はありません。

(2) DB 型ロック制御機能

- ・データベースの DBMS_LOCK パッケージを用いてロック制御を行います。
- ・同じロック ID を使用することで論理ノードを跨ったロック制御が行えます。
- ・DBMS_LOCK 動作中にシグナルを受信してもエラーを返却しません。
- ・通常のデッドロック検出に加えてアプリケーションによる DB 更新との相互デッドロック検出ができます。

2.6 メッセージ出力機能

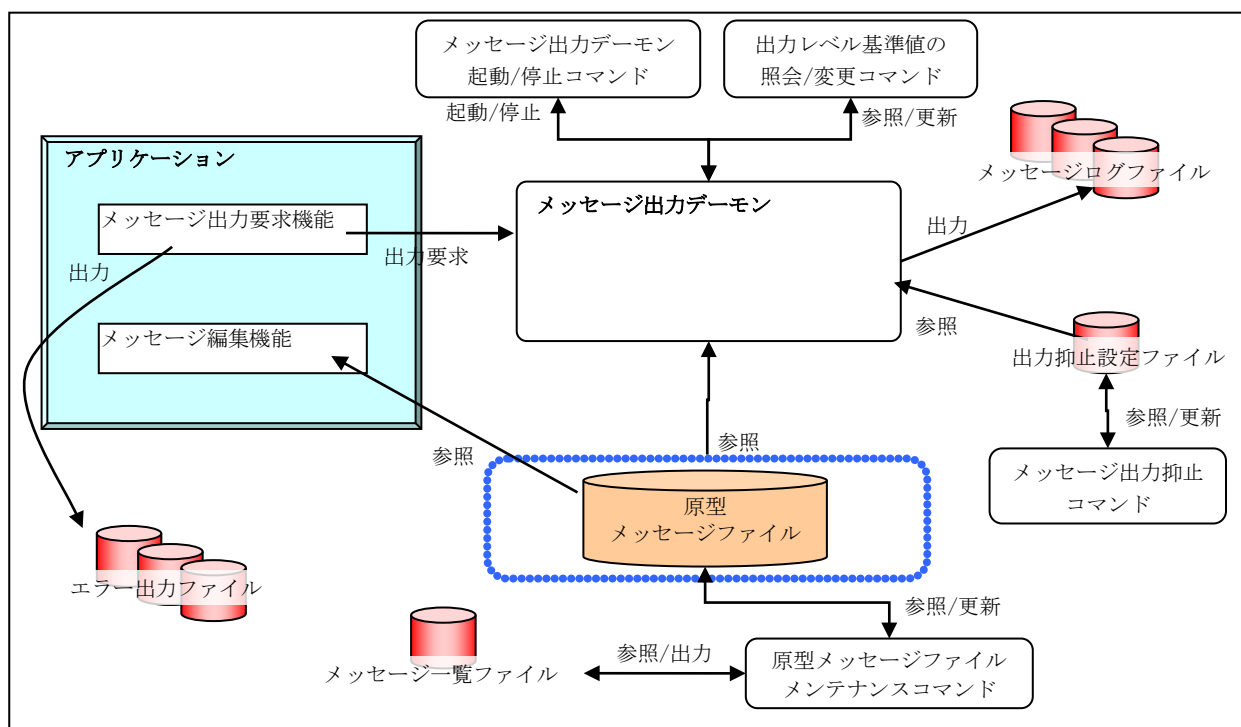
メッセージ出力機能は、ユーザアプリケーションがユーザ固有の重要メッセージをログファイルに出力するためのインタフェースを提供します。

本機能では、メッセージ出力デーモンによって、メッセージの出力処理、出力動作の制御、及びメッセージ情報の管理を行います。これにより、ユーザアプリケーションをとめることなく、設定を変更することができます。

本機能が出力するメッセージは、tail コマンド等で複数のユーザから参照可能なテキスト形式とし、WebSam 等の統合運用管理基盤と連携することにより、運用監視端末に出力できます。また、メッセージの内容によって、メッセージレベル(1(最高)～5(最低))とメッセージ種別(E(異常)/W(警告)/I(通知))を設定することができます。

なお、出力するメッセージは、メッセージ一覧ファイル、原型メッセージファイルの2つのファイルにおいて管理します。原型メッセージファイルとは、メッセージ出力機能が参照する DBM 形式のファイルを指し、メッセージ出力機能はこのファイルに記載されたメッセージを出力します。また、メッセージ一覧ファイルとは、メッセージの一覧をテキスト形式で記載したファイルを指し、利用者はこのファイルをメンテナンスします。

メンテナンスしたメッセージ一覧ファイルは、原型メッセージファイルメンテナンス機能を利用することで、原型メッセージファイルに反映することができます。原型メッセージファイルへの反映をもって、メッセージ出力機能が出力するメッセージが切り替わります。



図中の用語について：

- ・ メッセージ出力デーモン…メッセージ出力機能のデーモンプロセス
- ・ メッセージログファイル、エラー出力ファイル…DIOSA/XTP のログ情報や、利用者から出力要求のあったメッセージを出力するファイル
- ・ 出力抑止設定ファイル…メッセージ出力を抑止するためのメッセージ ID の一覧を管理するファイル
- ・ メッセージ一覧ファイル、原型メッセージファイル…出力するメッセージの一覧を管理するファイル

2.6.1 機能説明

(1) メッセージ出力要求機能 (API 提供)

ユーザアプリケーションは、メッセージ出力要求機能を使ってメッセージ出力要求をメッセージ出力デーモンに送信します。出力要求を受け取ったメッセージ出力デーモンは、指定されたメッセージ ID 及び出力情報をもとに、メッセージをログファイルに出力します。

メッセージ出力要求の送信に失敗した場合はリトライ処理を行います。メッセージ出力デーモンが起動していない場合は、メッセージをエラー出力ファイルへ出力します。また、エラー出力ファイル書き込み時にエラーが発生した場合は syslog へ出力します。

(2) メッセージ編集機能 (API 提供)

ユーザアプリケーションが独自にメッセージを出力したい場合、メッセージ編集機能を使用することにより、原型メッセージファイルに登録されているメッセージを取得することができます。

(3) メッセージ出力デーモン起動停止機能

dimsgdctrl(メッセージ出力デーモンの起動停止コマンド)により、メッセージ出力デーモンを起動/停止します。

(4) 出力レベル基準値照会/変更機能

メッセージ出力機能は、メッセージレベルが出力レベル基準値以下のメッセージを出力します。

出力レベル基準値は、dimsglvref(出力レベル基準値照会コマンド)により、照会することができます。

なお、出力レベル基準値の初期値は環境変数 DIOSA_MSG_LEVEL に設定された値であり、dimsglvmod(出力レベル基準値変更コマンド)により、出力レベル基準値を変更することができます。

(5) メッセージ出力抑止機能

dimsglimit(メッセージ出力抑止コマンド)により、指定したメッセージ ID の出力抑止/抑止解除/状態照会を行います。

(6) 原型メッセージファイルメンテナンス機能

dimsgmtn(原型メッセージファイルメンテナンスコマンド)により、メッセージ一覧ファイルから原型メッセージファイルの作成、及び原型メッセージファイルからメッセージ一覧ファイルの作成を行います。

これを利用することで、原型メッセージファイルから内容を取得し、修正後再度登録することができます。また、本機能はメッセージ出力デーモンが起動中でも起動することが可能で、メッセージ出力デーモンは更新した原型メッセージファイルの内容を動的に反映します。

これにより、出力メッセージ本文がアプリケーションと分離されるため、メッセージの共有化が図られるとともに、アプリケーションへの影響なく、メッセージ追加/削除/更新することができます。

2.7 アプリケーショントレース機能

アプリケーショントレース機能は、ユーザアプリケーションがユーザ固有のトレース情報を出力するためのインタフェース（API）を提供します。これにより、ユーザアプリケーションの任意の動作情報、障害発生時の解析情報を必要な場合にのみ、ファイルに出力することが可能となります。

出力方式は用途に応じて下記の2パターンを選択可能です。

- ファイル直接出力

トレース情報をスレッド単位にファイルに直接出力します。トレース出力APIの延長で、ファイルへのI/Oが発生します。

- メモリ経由出力

トレース情報を全てのプロセスから同じファイルに出力します。トレース出力APIの延長では、トレース情報は共有メモリ上に一旦蓄積され、共有メモリが一杯になったタイミングでトレース情報出力デーモンによりファイルへの出力が行われます。

アプリケーショントレース機能を利用して出力されるトレース情報は、バイナリ形式のファイルで出力されます。この出力ファイルを元に任意の検索条件で抽出し、テキスト形式で編集出力するためのコマンドが用意されています。

また、アプリケーショントレース機能の環境設定の一部は、動的に変更可能であり、変更のためのコマンドが用意されています。

2.7.1 機能説明

(1) トレース情報ファイル直接出力(API 提供)

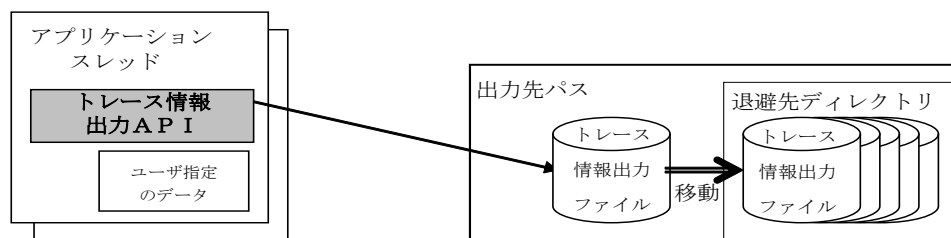
トレース情報をファイルへ直接出力するAPIを提供します。ただし、API実行時に指定する出力レベルが環境設定で指定された出力レベルより大きい場合は、トレース出力は抑止されます。

トレース出力ファイルはスレッド単位に作成されます。

また、ファイル直接出力の場合、環境設定により下記の2つのファイル出力パターンが選択可能です。

<時系列保存方式>

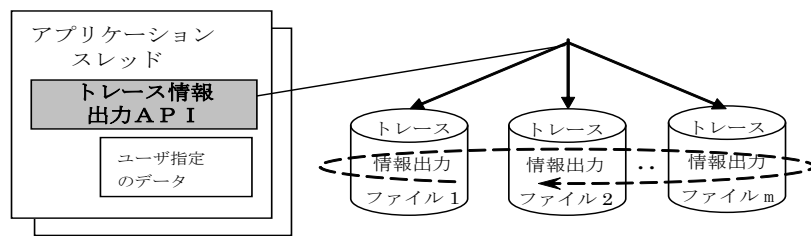
環境設定で指定されたサイズに達すると退避用ディレクトリにファイルを移動します。



<ローテーション方式>

環境設定で指定されたファイル数に応じて出力ファイルをローテーションして使用します。

なお、上書きするか、上書きせずにトレース出力をエラーとするかは選択可能です。



(2) トレース情報メモリ経由出力 (API 提供)

トレース情報をメモリ経由で出力するAPIを提供します。ただし、API実行時に指定する出力レベルが環境設定で指定された出力レベルより大きい場合は、トレース出力は抑止されます。

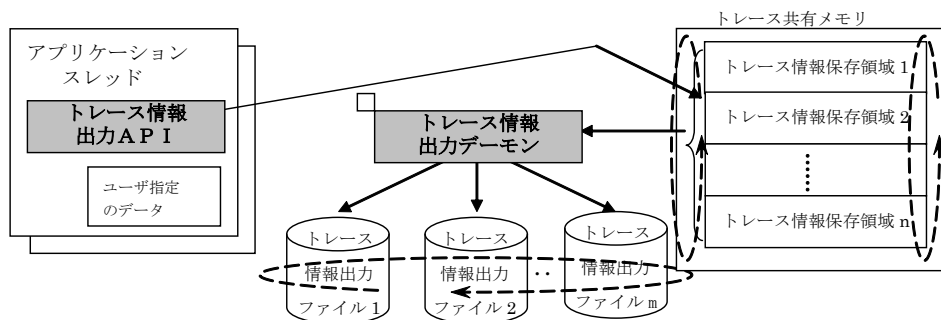
トレース情報を全てのプロセスから同じファイルに出力します。

また、メモリ経由出力の場合、ファイル出力パターンはローテーション方式となります。

<ローテーション方式>

環境設定で指定されたファイル数に応じて出力ファイルをローテーションして使用します。

なお、トレース共有メモリにトレース情報を格納する領域が無い場合、トレース共有メモリ上のトレース情報を上書きするか、上書きせずにトレース出力をエラーとするかは選択可能です。



(3) トレース情報ファイル検索編集コマンド

トレース出力ファイルから、任意の検索条件（下記）でトレース情報を検索し、テキスト形式に編集出力するコマンドを提供します。

<検索条件>

時間帯、出力レベル、ユーザデータ部の値、関数名、コメント

(4) トレース情報動作変更コマンド

下記の環境設定を動的に変更するコマンドを提供します。なお、環境設定の変更は、トレース情報動作変更コマンド実行後のトレース出力API実行時に有効となります。

- ・ トレース情報出力レベル
- ・ トレース情報出力ファイルへのユーザデータ出力フラグ
- ・ トレース情報出力時の強制書き込みフラグ ※ファイル直接出力の場合のみ有効
- ・ トレース情報出力ファイルの絶対パス ※次回ローテーション以降に有効

ユーザデータ出力フラグは、ユーザデータ(diosaapptrcf())、diosaapptrcm() でユーザが指定したバイナリデータをファイルに出力するかどうかを指定します。ユーザデータの詳細については「API リファレンス」を参照してください。

強制書き込みフラグは、トレース情報を API の呼び出しタイミングで強制的にファイル出力するか、任意のタイミング(OS 環境に依存)でファイル出力するかを指定します。

(5) **トレース情報フラッシュコマンド**

共有メモリ上に蓄積されたトレース情報をファイルに出力するコマンドです。メモリ経由のトレース出力の場合にのみ有効です。

2.8 アプリケーション動的置換機能

アプリケーション動的置換機能はオンライン処理システムにおいて、オンラインを停止させずに、アプリケーションプログラムの追加や迅速な置換を行うための機能を提供します。

2.8.1 諸概念

(1) **LM**

アプリケーション動的置換機能を利用するプロセスです。

(2) **論理ライブラリ**

バグ置換や平行稼動でライブラリ名や格納場所が変更されても変わらない、論理的に同じライブラリの集合です。

置換コマンド等で指定するのも論理ライブラリ名です。

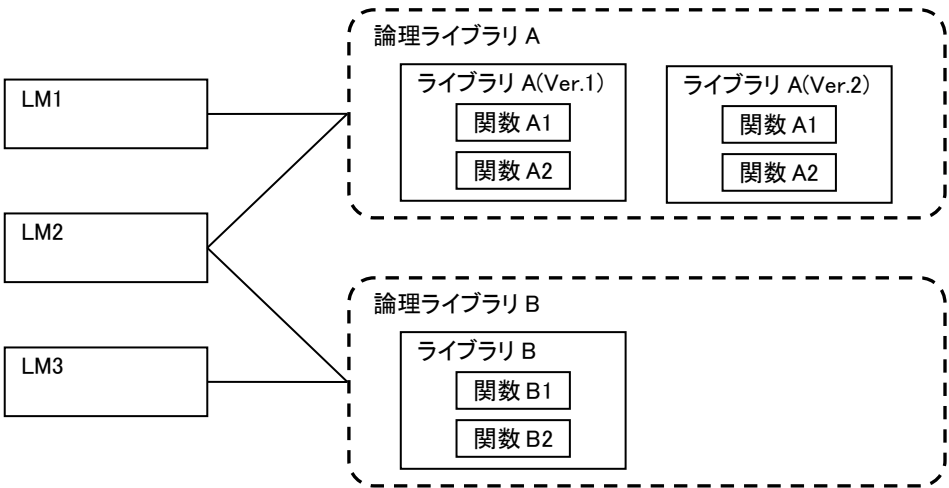
実ライブラリファイルと同じ名前で定義することも可能です。

(3) **ライブラリ**

呼び出し関数が含まれるライブラリファイルです。

(4) **関数**

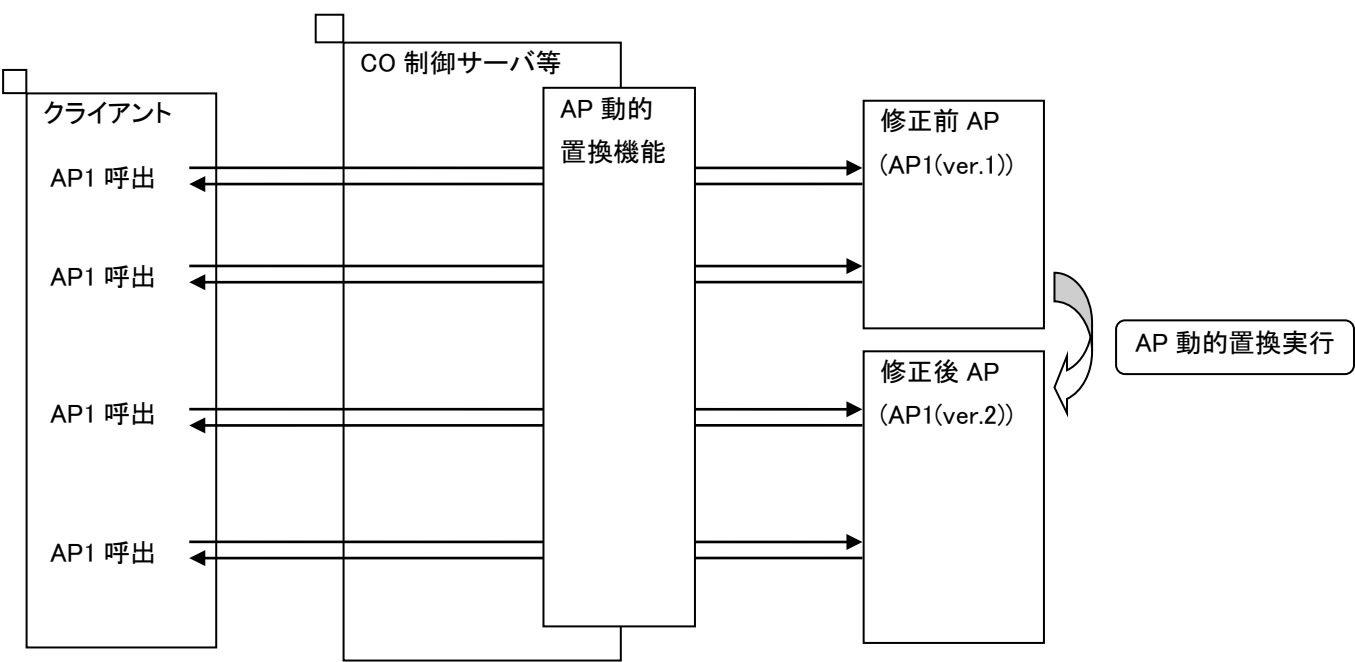
実際に呼び出される関数です。



2. 8. 2 機能説明

(1) アプリケーション動的置換機能

業務を停止せずに AP (CO、利用者出口、ユーザ関数) の追加、削除、置換が可能です。



(a) 環境定義による呼び出し

環境定義 APLIB 節にアプリケーション動的置換機能を利用するプロセス、論理ライブラリ等の情報を定義することで、アプリケーション動的置換機能を利用可能です。

諸概念で説明した項目と、環境定義の対応は以下の通りです。

概念	環境定義項目
LM	APLIB 節-LM 項
論理ライブラリ	APLIB 節-LLIB 項
ライブラリ	APLIB 節-LLIB 項-LIBRARY 項
関数	APLIB 節-LLIB 項-LIBRARY 項-FUNC 項

環境定義による関数呼び出しをおこなう場合、呼び出したい関数は全て FUNC 項に定義する必要があります。定義されていない場合、ライブラリに関数が含まれていても呼び出すことはできません。

(b) 環境変数による呼び出し

環境変数 DIOSA_LIBNAME に、アプリケーション動的置換対象関数を含むライブラリを指定することで、アプリケーション動的置換機能からの呼び出しが可能です。

(2) **その他機能**

(a) ロード、アンロードタイミング制御

アプリケーション動的置換機能では、ライブラリをロード、アンロードするタイミングを制御することが可能です。

・ロードタイミング

指定	動作
起動時	プロセス起動時にその LM に定義されたライブラリをロードします。 ロード時に定義された関数のシンボル解決まで実行するため、呼び出し時のオーバーヘッドが小さいです。 C0 制御サーバで使用するライブラリについては起動時ロードを推奨します。
呼び出し時	関数呼び出し時にその関数が含まれるライブラリをロードします。 起動時のロード処理をおこなわないため、起動時間は短縮されますが、初回の関数呼び出し時にロード処理、関数のアドレス解決をおこなうため、呼び出し時のオーバーヘッドは大きくなります。ロード済みのライブラリについてはロード処理をスキップします。 バッチアプリケーション制御等、多くのアプリケーションの一部のみを使用して動作するバッチプロセスでは呼び出し時ロードを推奨します。

・アンロードタイミング

指定	動作
なし	ロードしたライブラリは置換されるまでアンロードされません。 繰り返し呼び出す場合に、毎回ロード処理が実行されることがないため、2回目以降の呼び出し時のオーバーヘッドが小さいです。
トランザクション終了時	トランザクション終了時にロードされていたらアンロードします。 トランザクションごとにロード処理をおこなうため、呼び出し性能は悪いですが、アプリケーション実行中以外は、ライブラリファイルは常に更新可能で、置換コマンド等を実行しなくても、各トランザクションで常に最新のライブラリをロードして関数呼び出しをおこなえます。

呼び出し方式ごとのロード/アンロードするタイミングは以下の通りです。

(i) 環境定義 APLIB 節に定義されたライブラリ

APLIB 節に定義したライブラリについては、LLIB 項-LOAD パラメータ、UNLOAD パラメータを定義することで、ライブラリをロード、アンロードするタイミングを指定することができます。

パラメータ	指定	動作
LOAD	INIT	プロセス起動時にロードする(既定値)
	DEFERRED	関数呼び出し時にロードする
UNLOAD	NO	ロードしたライブラリは置換指示があるまでアンロードしない(既定値)
	TRNS	ロードしたライブラリはトランザクション終了時にアンロードする

(ii) 環境変数 DIOSA_LIBNAME に指定されたライブラリ

環境変数 DIOSA_LIBNAME に指定されたライブラリは、起動時にロードされることはなく、必ず関数呼び出し時にロードされます(既にロード済みの場合、ロード処理はおこなわれません)。

アンロードタイミングは環境変数 DIOSA_LIBUNLOAD で指定することができます。

DIOSA_LIBUNLOAD の指定	動作
NO	ロードしたライブラリは置換指示があるまでアンロードしない
TRNS	ロードしたライブラリはトランザクション終了時にアンロードする(既定値)

(b) シンボル解決について

ライブラリのロード時に参照シンボルの解決をおこなうかどうかを指定可能です。

APLIB 節-LM 項-LOADMODE パラメータで指定します。

パラメータ	指定	動作
LOADMODE	YES	ロード時にシンボル解決をおこなう
	NO	呼び出し時にシンボル解決をおこなう

シンボル解決の指定対象は関数のみです。外部変数については LOADMODE パラメータの指定によらず、ロード時にシンボル解決をおこないます。

ライブラリ間での関数の直接呼び出しをおこなった場合、LOADMODE を NO に指定する、定義順を呼び出し先、呼び出し元の順番に定義する、などの方法で関数呼び出しは可能となりますが、置換処理が正常におこなえないことがあるため、ライブラリ間での関数の直接呼び出しは推奨しません。

(c) 全プロセス共通ライブラリについて

全ての LM に対してリンクするライブラリは、DFLTLLIB 項に記述することで、全ての LM 項に定義を記述するする必要がなくなります。

DFLTLLIB 項を定義した場合、LM 項が定義されていない LM についても、DFLTLLIB 項に書かれているライブラリがロード対象となります。

2. 8. 3 API/コマンド

アプリケーション動的置換機能が提供する C 関数の一覧を示します。

API 名	説明
diosavcall	関数呼び出し API

アプリケーション動的置換機能が提供するコマンドの一覧を示す。

コマンド名	説明
didlrchg	アプリケーション動的置換コマンド
didlrinit	アプリケーション動的置換機能初期化コマンド
didlrreflib	論理ライブラリ情報参照コマンド
didlrreflm	LM 定義情報参照コマンド

2.9 経過時間監視機能

経過時間監視機能は、CO制御、バッチAP制御、およびデータストア基盤ログデータ実行制御上で動作するアプリケーションの経過時間を監視することにより、アプリケーションのストールやプロセス停止を監視する機能です。

また、アプリケーション内で経過時間をリセットする機能および経過時間を超過したプロセスを停止させる機能を提供します。

2.9.1 経過時間監視機能

経過時間監視デーモンとして起動され、一定間隔ごとにプロセスおよびアプリケーションの実行時間の監視を行う機能です。アプリケーション開始時に時刻を経過時間監視テーブルに登録し、定期的にテーブルをチェックし、警告時刻（登録時刻＋経過監視時間）が現在時刻を過ぎている経過時間情報が存在した場合には、メッセージを出力し、経過時間情報を削除します。また、監視対象プロセスが存在し、経過時間監視テーブルのシグナル送信フラグが設定されている場合は、監視対象プロセスをシグナルにより停止させます。

ただし、経過時間監視停止機能により、停止要求されているものは監視対象外とします。

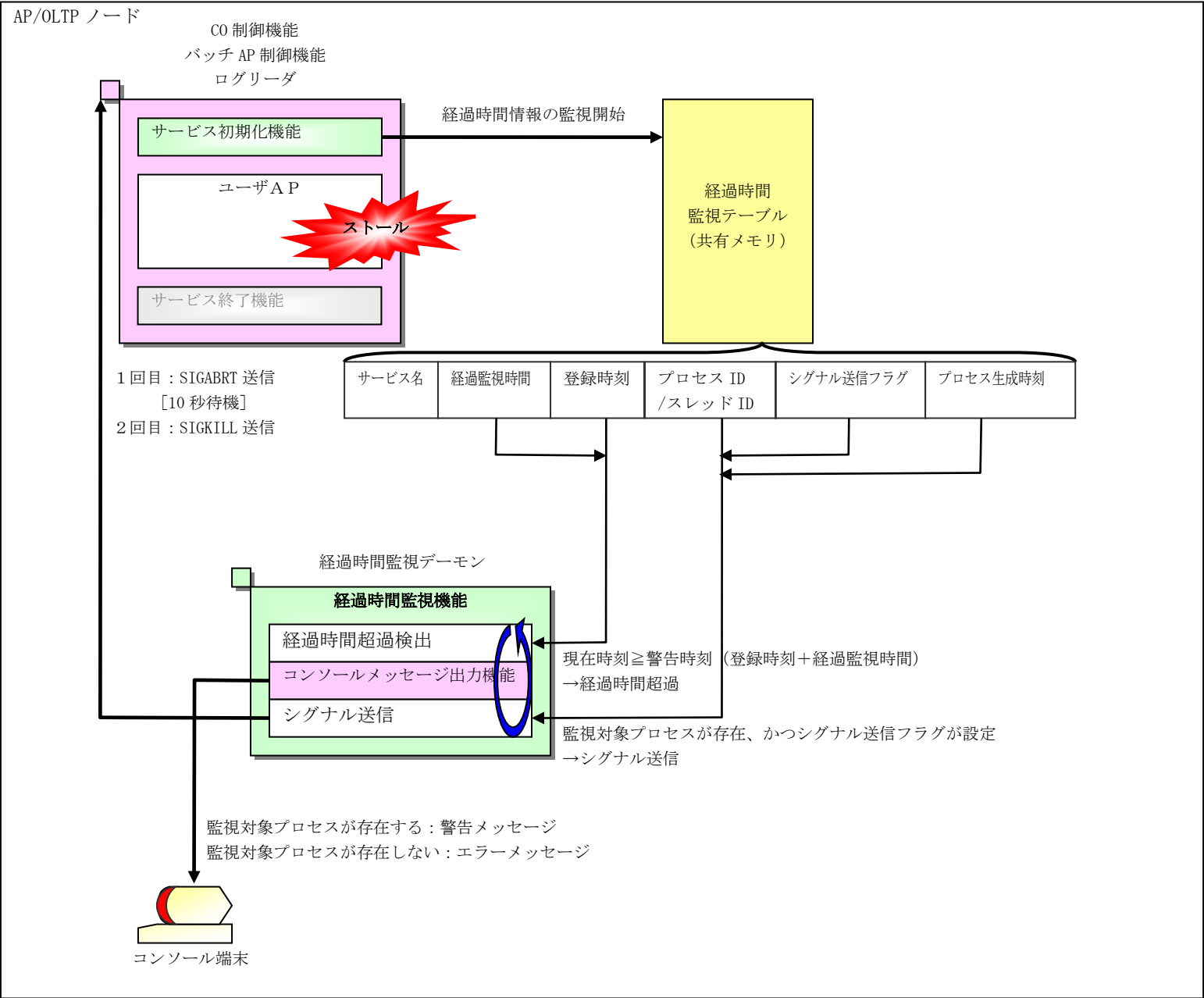
経過監視時間および検出時の動作情報については、CO制御およびログリーダは環境定義(COCENV節のELPTIMEパラメータ、DELAYED節のELPTIMEパラメータ)、バッチAP制御は起動パラメータで指定できます。

(1) メッセージ

- 監視対象プロセスが経過監視時間を超過した場合、警告メッセージを出力します。
- 経過監視時間超過の警告メッセージを出力した後、当該プロセスにシグナルを送信した場合、警告メッセージを出力します。
- 経過監視時間を超過した場合の警告メッセージにはユーザ情報も表示します。

(2) シグナル

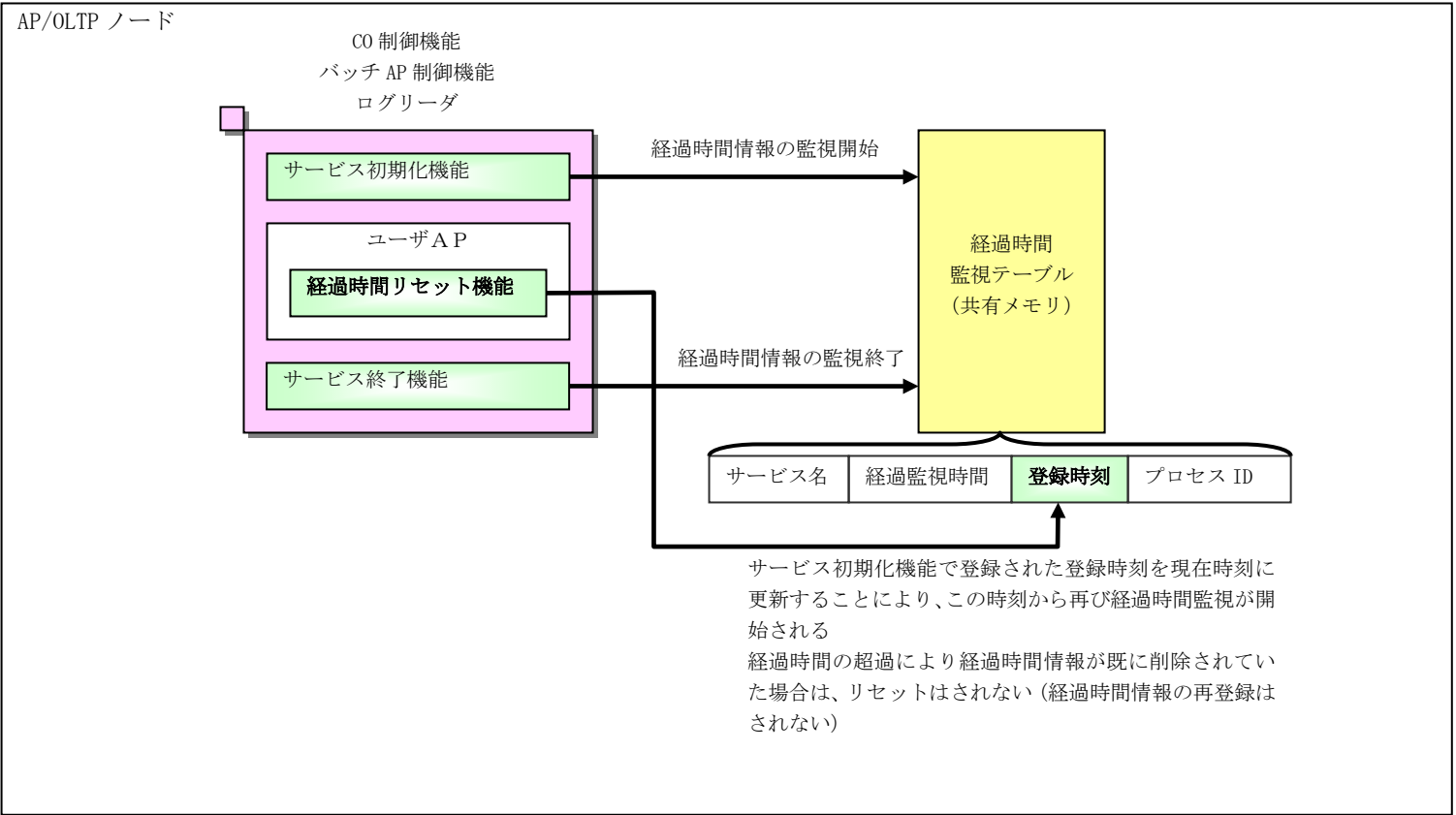
- 経過監視時間超過時に監視対象プロセスが存在し、シグナル送信フラグが設定されていた場合、監視対象プロセスを停止させます。
- 監視対象プロセスにSIGABRTシグナルを送信した後も監視対象プロセスが存在する場合、10秒後にSIGKILLシグナルを送信します。



2.9.2 経過時間リセット機能

経過時間監視テーブルに登録された経過時間情報の登録時刻を現在時刻に更新する機能です。本機能により経過時間がリセットされ、経過時間監視デモンのサービスの経過時間の監視が延長されます。

また、リセット回数に制限値を設け、最大リセット回数以上にリセットが指示された場合も監視対象とします。



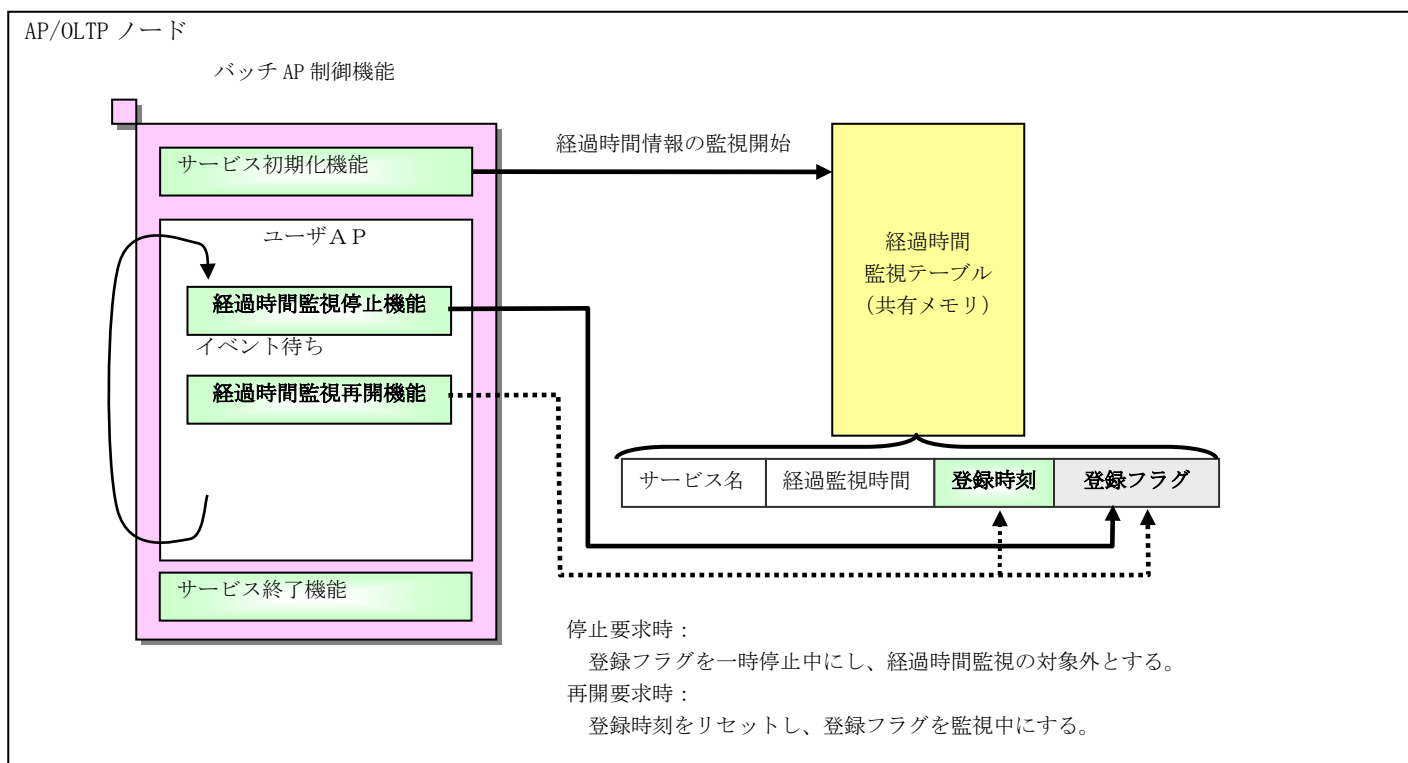
2.9.3 経過時間監視停止／再開機能

バッチAP制御上で動作するCOから呼び出され、経過時間監視の停止／再開を要求する機能です。

バッチAP制御のCOを常駐ジョブとして動作させる場合に、イベント待ちの間は経過時間の監視対象外とするために使用します。

経過時間監視停止処理は、経過時間情報の登録フラグを監視一時停止に設定し、経過時間監視の対象外とします。

経過時間監視再開処理は、経過時間情報の登録時刻、警告時刻（登録時刻＋経過監視時間）をリセットし、登録フラグを監視中に設定することで経過時間監視を再開します。



2.9.4 ユーザ情報登録機能

ユーザ情報を登録／更新するAPIを提供します。経過時間超過時に表示するコンソールメッセージにこのユーザ情報を付加します。本情報を利用することで、該当アプリケーションや処理内容等を特定することができます。

2.9.5 経過時間監視照会機能

経過時間監視テーブルに登録されている経過時間情報を出力する機能です。

本機能により経過時間を監視しているサービスを認識することができます。

2.10 稼動統計機能

稼動統計機能は、C Oの稼動情報を蓄積し、収集・編集する機能を提供します。C O実行毎に開始時刻、経過時間、C P U時間、インメモリサーバA P I 実行時間等の稼動情報を採取・蓄積するので、定常時のアプリケーションの動作状態確認だけでなく、性能問題等異常時の原因解析にも役立ちます。

出力機能は業務アプリケーションと非同期に動作し、物理ファイルへの出力を待ち合わせないことで、運用中の業務アプリケーションへの負荷を極力抑え且つ高速化を図っています。また、出力ファイルを複数セット使用して出力先を分散利用することでシステム全体のスループットを向上させることが可能です。

また、データファイルは一定容量が出力されると次ファイルへ順次出力スワップし一定数ファイルへの出力が完了したら、先頭ファイルをサイクリックに利用します。すなわち設定されたファイル容量のみを使用することで、パーティションへの不用意な容量圧迫を防止することが可能です。

また、出力された稼動情報の分析を容易に行う事を可能とするため、各ノードの稼動統計ファイルを収集する機能と、収集した稼動統計ファイルを任意のファイルへマージ・編集する機能を提供します。

2.10.1 稼動統計出力機能

(1) 稼動情報出力機能

C O制御およびバッチアプリケーション制御と連携し、C O呼び出しの前後で情報を採取し、同じノード内で起動されているファイル出力機能へソケット通信で、稼動情報を送信します。

ファイル出力機能への各種情報の送信にはU N I Xドメインを使用し、受信側（ファイル出力機能）の処理とは非同期に処理を行うことにより、業務アプリケーションへの負荷を最小限に抑えます。

本情報の採取有無は、C O制御環境定義(\$COCENV-TRANS-OPS)で定義します。

採取する情報には、次の2種類の情報があります。

(a) C Oの稼動統計情報

C O単位の稼動情報であり、1つのC O実行にかかる処理時間やC P U時間を採取するのが主な目的です。C Oの稼動統計情報では次の情報を採取します。

- オンライン／バッチ種別
- 論理ノード種別
- 論理ノード名
- 論理ノードI D
- プロセスI D
- プロセス内通番
- T P B A S E トランザクションI D
- T P B A S E モニタ名
- C Oの関数名
- 電文開始時刻
- C Oの開始時刻
- C Oの処理時間（マイクロ秒単位）
- C OのシステムC P U時間（マイクロ秒単位）

- COのユーザCPU時間（マイクロ秒単位）
- `diosaUCA`の状態コード
- `diosaUCA`の利用者コード
- ユーザ情報

(b) コミットの稼動統計情報

トランザクション終了時のコミット、アボート処理のアボート#2 出口呼出し後のコミット情報を採取します。採取する情報はCOの稼動統計情報と同じですが、COの関数名は、トランザクション終了時のコミットで"@COMMIT_EXIT_SVCTERM"、アボート処理のアボート#2 出口呼出し後のコミットで"@COMMIT_EXIT_ABORT"が表示されます。

なお、本情報を採取するには環境変数(DIOSA_COCCOMMITOPS)を指定する必要があります。

(c) トランザクション単位の稼動統計情報

CO制御・バッチAP制御上のトランザクション単位の性能情報を採取します。トランザクション単位の稼動統計情報では次の情報を採取します。

- オンライン／バッチ種別
- 論理ノード種別
- 論理ノード名
- 論理ノードID
- プロセスID
- プロセス内通番
- TPBASEトランザクションID
- TPBASEモニタ名
- CO関数名（複数実行時は最後に実行したCO）
- 電文開始時刻
- トランザクション処理開始時刻
- 処理時間
- インメモリサーバAPI情報

※ インメモリサーバAPI情報は、下記の情報を採取する。

- レコード読込（呼出回数、平均処理時間、ブリッジ回数）
- レコード更新（呼出回数、平均処理時間、ブリッジ回数）
- レコード追加（呼出回数、平均処理時間、ブリッジ回数）
- レコード削除（呼出回数、平均処理時間、ブリッジ回数）
- 全レコード削除（呼出回数、平均処理時間、ブリッジ回数）
- コミット／ロールバック（処理時間、ブリッジ回数）
- ロック待ち情報
- デッドロック情報
- ロック情報
- グループコミット情報

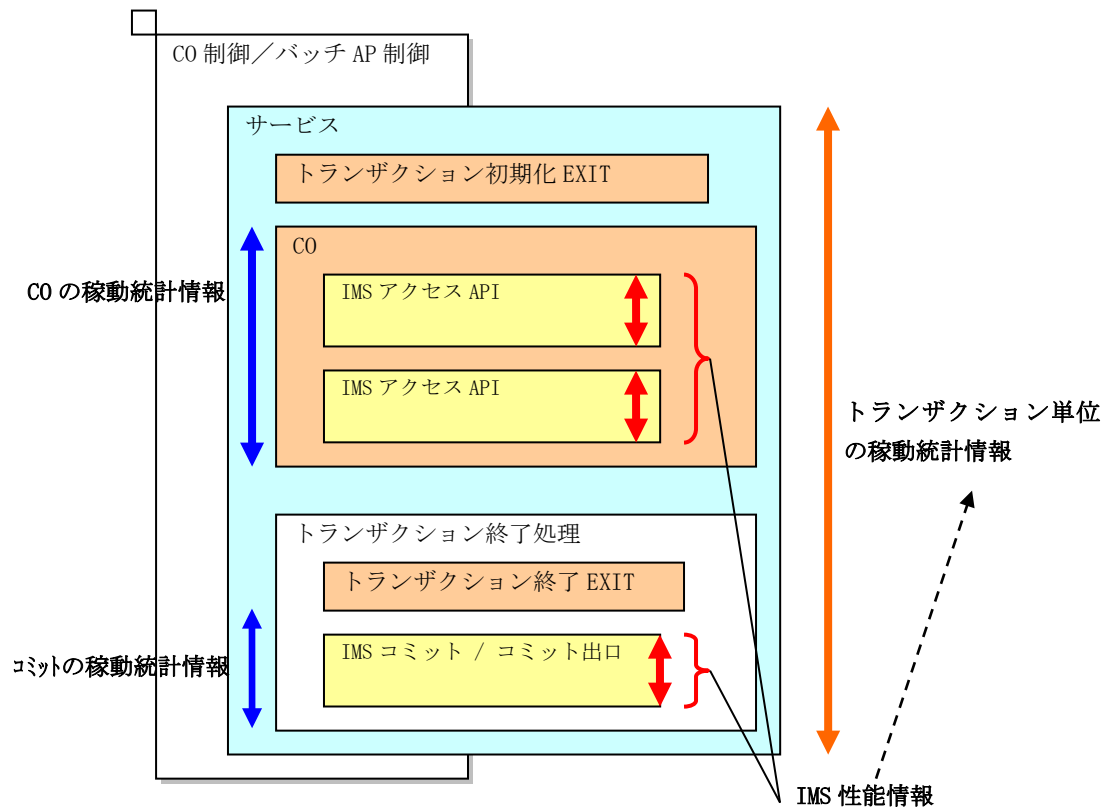
(d) 稼動統計情報の採取区間

C0 の稼動統計情報は、1 つの C0 実行に関する情報を採取します。コミットの稼動統計は、トランザクション終了時のコミットに関する情報を採取します。トランザクション単位の稼動統計情報は、サービス開始からコミット処理までの区間に含まれる情報を採取します。

• 例 1)

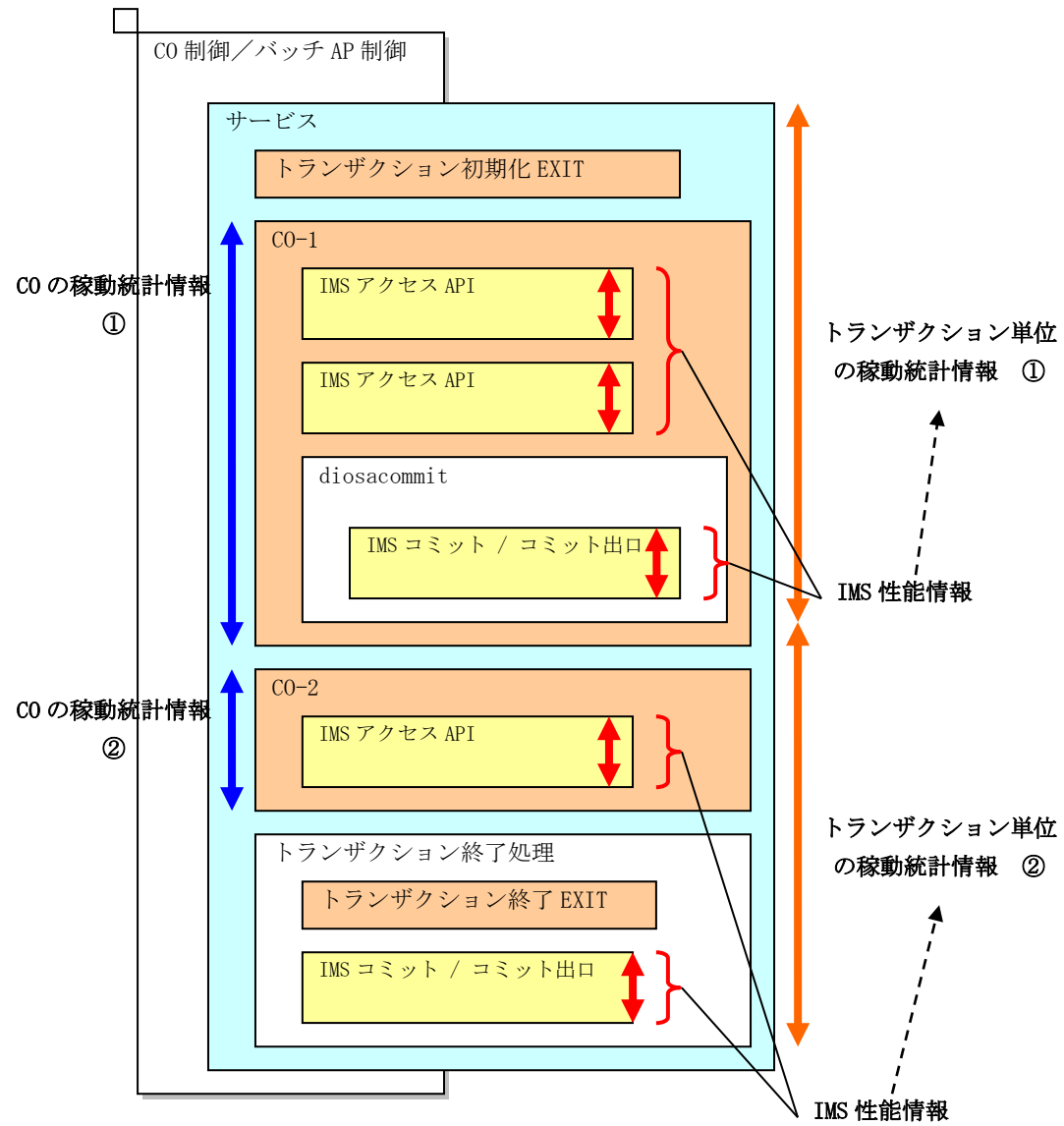
下記のような処理の場合、C0 の稼動統計情報 1 件、コミット稼動統計情報 1 件とトランザクション単位の稼動統計情報が 1 件出力されます。

コミットの稼動統計情報を採取する場合



- 例 2)

下記のような処理の場合、CO の稼動統計情報 2 件と、トランザクション単位の稼動統計情報 2 件が出力されます。



(2) ユーザ情報登録機能

CO 制御およびバッチアプリケーション制御上の CO から呼び出され、稼動統計情報として出力したいユーザ情報を登録する機能です。ユーザ情報は、バイナリ形式、または文字列形式で登録することができます。

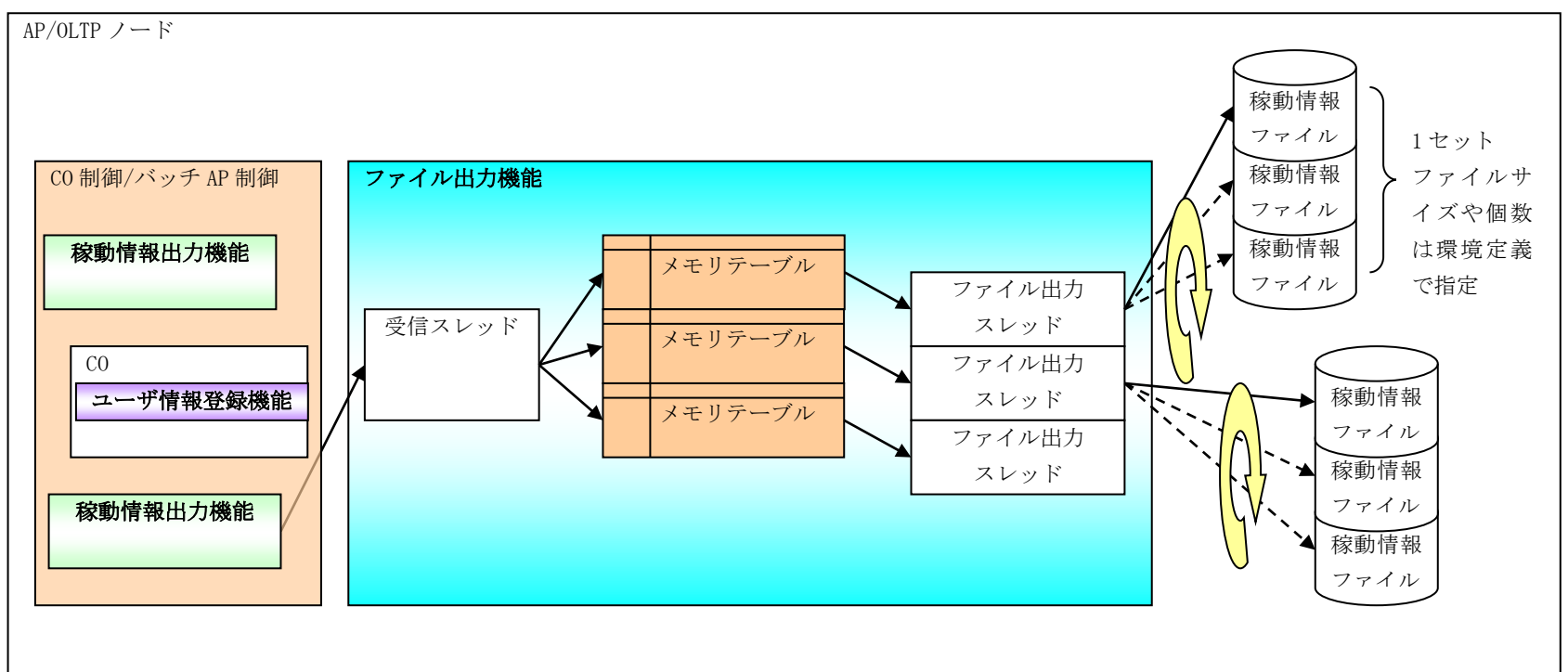
(3) ファイル出力機能

稼動情報出力機能からソケット通信で渡された稼動情報をフラットファイル（稼動情報ファイル）へ出力する機能です。

当機能は、稼動情報出力機能から各種情報を受信するスレッドとファイルへ出力するスレッド（ファイル出力スレッド）を持ち、複数のプロセスからの非同期通信を高速に処理できます。

出力ファイルは、ファイル出力スレッド単位に、環境定義にて指定された固定サイズのファイルを、同じく環境定義にて指定された個数を使用してサイクリックに出力します。ファイル出力スレッドを多重化することで、出力ファイル群を複数セット使用することも可能です。ファイル出力スレッドは、メモリテーブル内にデータが一定量蓄積されると、出力ファイルへ稼動情報を出力します。

また、メモリテーブル中に残っている稼動情報をファイルへフラッシュさせるコマンドを持ちます。



2.10.2 稼動統計収集機能

(1) 稼動統計収集機能

各ノードで出力された稼動情報ファイルを集めるための機能で、環境定義で指定されたディレクトリ、あるいは実行時に指定されたディレクトリに収集します。

また、オプション指定により収集対象の選択を行うことができます。

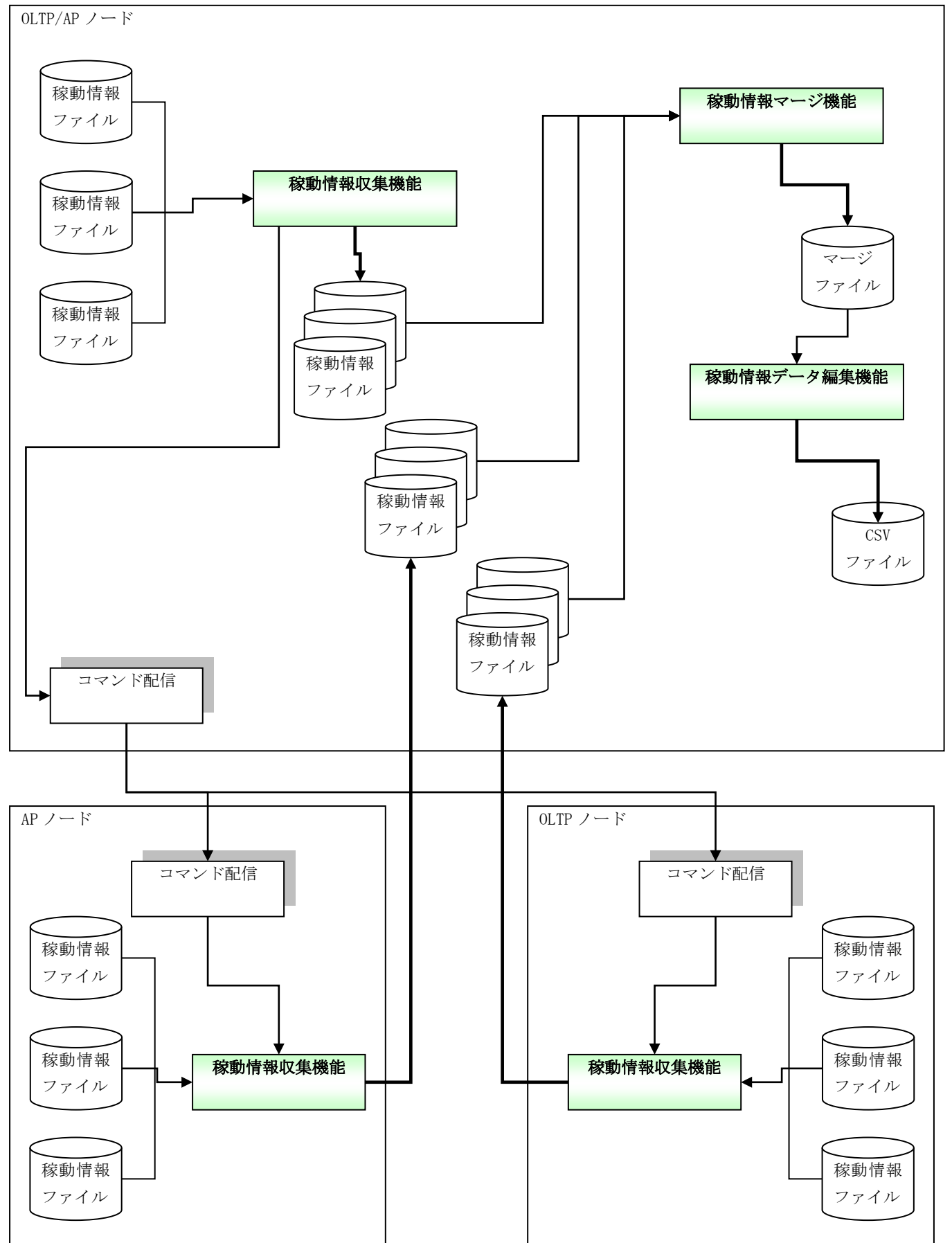
(2) 稼動統計マージ機能

稼動統計収集機能により 1 つのノード・ディレクトリに集められた稼動情報ファイルを時系列にソート・マージし、別ファイルへ出力する機能です。

(3) 稼動統計データ編集機能

稼動統計マージ機能の出力ファイルから、編集パラメータで指定された情報を抽出してCSV形式で出力する機能です。

編集パラメータには論理ノード名、CO関数名、プロセスID、時間範囲（FROM～TO）、COの終了ステータス等があります。



2.10.3 運用方法

稼動統計収集機能を運用する方法について以下に説明します。

(1) 日毎に収集～編集を実施する手順

日毎に稼動統計収集機能を実施する場合、下記①～④の手順を毎日実施し、C S V形式のデータ編集ファイルを日毎に作成していきます。

なお、F T Pが使用できる場合、手順②は不要となります。F T Pが使用できない場合、手順②から実施します。

① 稼動統計収集コマンド

実行例

```
diopsgather -m default
```

稼動統計収集コマンドが実行されると自分が属する論理システム配下の全O L T P / A P ノードで稼動統計転送コマンドが呼び出され、F T Pにて稼動統計ファイルを収集します。-m default オプションを付加することで、出力が完了し、未収集の稼動統計ファイルのみを収集対象とします。

※F T Pが使用できない場合、②から実行します。

② 稼動統計転送コマンド

実行例

```
diopsfput -m default
```

稼動統計転送コマンドは、自論理ノードの稼動統計ファイルを収集ディレクトリにコピーします。-m default オプションを付加することで、出力が完了し、未収集の稼動統計ファイルのみを収集対象とします。

稼動統計収集コマンドが使用できない場合、当コマンドを全O L T P / A P ノードで実行した後、収集した稼動統計ファイルを任意のディレクトリに移動してください。

③ 稼動統計マージコマンド

実行例

```
diopsmrg -m output -o マージファイル名(任意)
```

稼動統計マージコマンドにより、稼動統計収集コマンドもしくは稼動統計転送コマンドにて収集した複数の稼動統計ファイルを、開始時間でソートし1つのファイルにまとめます。-m output オプションを付加することで、新規にファイルを作成します。

④ 稼動統計データ編集コマンド

実行例

```
diopsedit マージファイル名 -T -o データ編集ファイル名(任意)_YYYYMMDD
```

稼動統計マージコマンドで出力したマージファイルを、稼動統計データ編集コマンドによりCSV形式に編集します。ファイル名に年月日を付加するなどして、日毎の稼動統計ファイルを作成していきます。

(2) **一つのファイルに稼動情報を随時蓄積する手順**

一つのファイルに稼動情報を蓄積する場合、一定期間毎に下記①～③の手順を繰り返します。必要に応じて、稼動統計データ編集コマンドによりデータ編集ファイルを作成することができます。

① 稼動統計収集コマンド

(1)-①と同様

② 稼動統計転送コマンド

(1)-②と同様

③ 稼動統計マージコマンド

実行例

```
diopsmrg -m append -o マージファイル名(既存)
```

稼動統計マージコマンドにより、稼動統計収集コマンドもしくは稼動統計転送コマンドにて収集した複数の稼動統計ファイルを、開始時間でソートし1つのファイルにまとめます。-m append オプションを付加することで、既存のマージファイルに追加出力されます。

④ 稼動統計データ編集コマンド

実行例

```
diopsedit マージファイル名 -T -o データ編集ファイル名(任意) [ -抽出条件 ]
```

稼動統計マージコマンドで出力したマージファイルを、稼動統計データ編集コマンドによりCSV形式に編集します。必要に応じて、開始/終了日時等の抽出条件をオプションに指定することで、特定の情報のみを取得することが可能です。

2.11 Tパス管理機能

2.11.1 Tパスの管理方法について

Tパス管理機能では、相手ノードのT P P宛に電文を送信し、正常応答が返却されるか否定応答が返却される（送信失敗、応答受信のタイムアウトも含む）かで、相手の状態を判別するヘルスチェック処理を行います。このヘルスチェック処理を行った結果の、相手ノードの状態をTパス状態と呼びます。また、このヘルスチェック処理により、以下の5つの状態異常をTパスの状態異常として検出します。

- 相手ノードのD I O S A / X T Pの起動未完了状態
- 相手ノードのT P B A S Eのダウン状態
- 相手ノードとの通信パスの障害
- 相手ノードのダウン
- 相手ノードのノード閉塞

また、Tパス管理機能では、上記の5つの状態ではない相手ノードの状態をTパスオープン状態と呼び、上記の5つの状態のいずれかの状態である相手ノードの状態をTパスクローズ状態と呼びます。

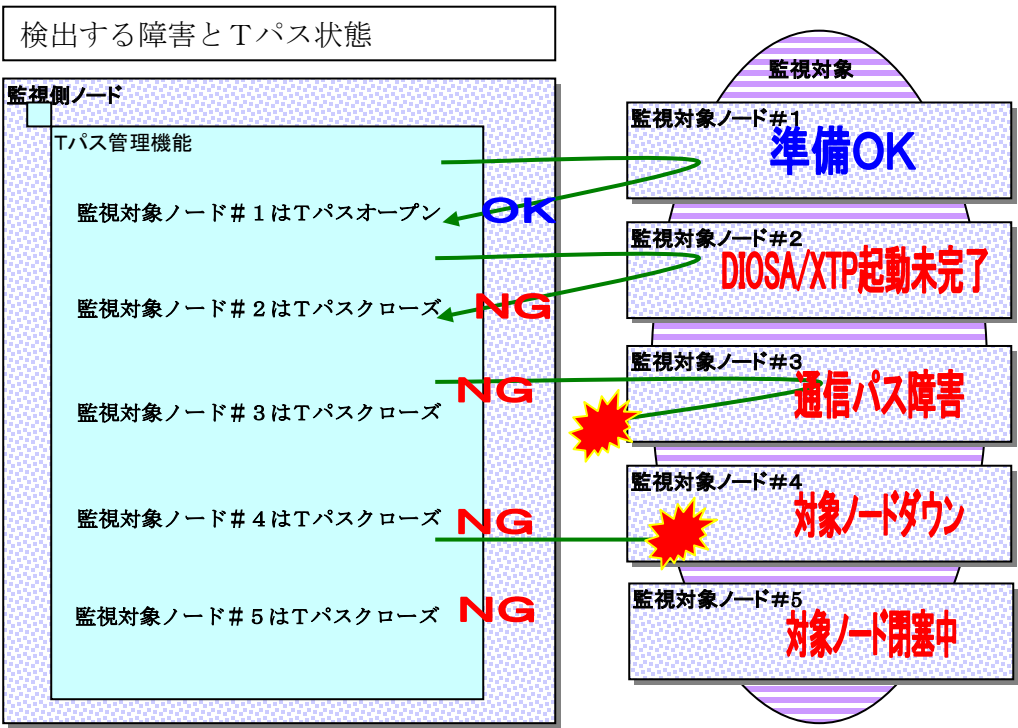


図 2-1 検出する障害と Tパス状態

2.11.2 Tパスのオープン・クローズ検出方法

Tパスのオープンやクローズの検出方法は以下のようになります。

(1) Tパスオープンの検出方法

相手ノードにヘルスチェック電文を非同期型送信し、送信した電文の正常応答を受信した場合をTパスオープン状態とします。

ネットワークが不安定な状態の場合、電文の送信が成功・失敗を繰り返す可能性があります。そのような状態をすぐにTパスオープン状態と判別しないために、複数回連続して正常応答を受信した時に初めてTパ

をオープン状態とする機能を提供します。この機能は、Tパス管理機能の環境定義にTパスオープンのチェック回数を定義することにより動作し、ヘルスチェックの間隔でTパスオープンのチェック回数分連続して正常応答を受信したときにTパスオープン状態とします。

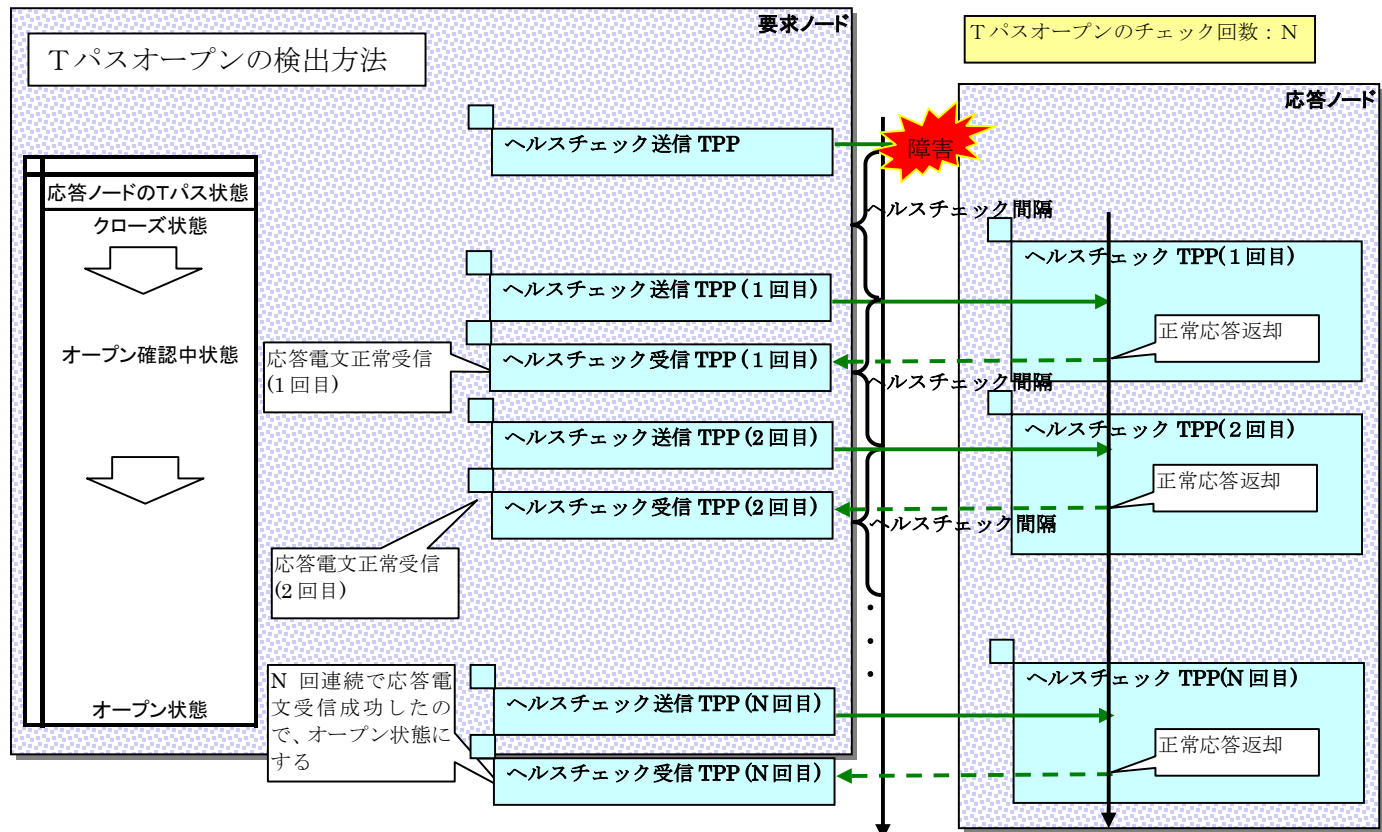


図 2-2 Tパスオープンの検出方法

(2) Tパスクローズの検出方法

相手ノードにヘルスチェック電文を非同期型送信し、送信した電文の異常応答を受信した場合やタイムアウトした（環境定義に指定したタイムアウト時間を経過しても応答を受信できない）場合または電文送信が失敗したときは、Tパスクローズ状態とします。

通信の瞬間的な切断や二重化されたネットワークの障害による切り替え時は、瞬間的に通信が切断され、その瞬間以外は正常に送信可能な状態があります。そのため、Tパス管理機能では、複数回連続して異常応答を受信した場合やタイムアウトした場合にTパスをクローズとする機能を提供します。この機能は、Tパス管理機能の環境定義にTパスクローズのチェック回数を定義することにより動作し、ヘルスチェックの間隔でTパスクローズのチェック回数分連続して異常応答受信またはタイムアウトしたときにTパスクローズ状態とします。

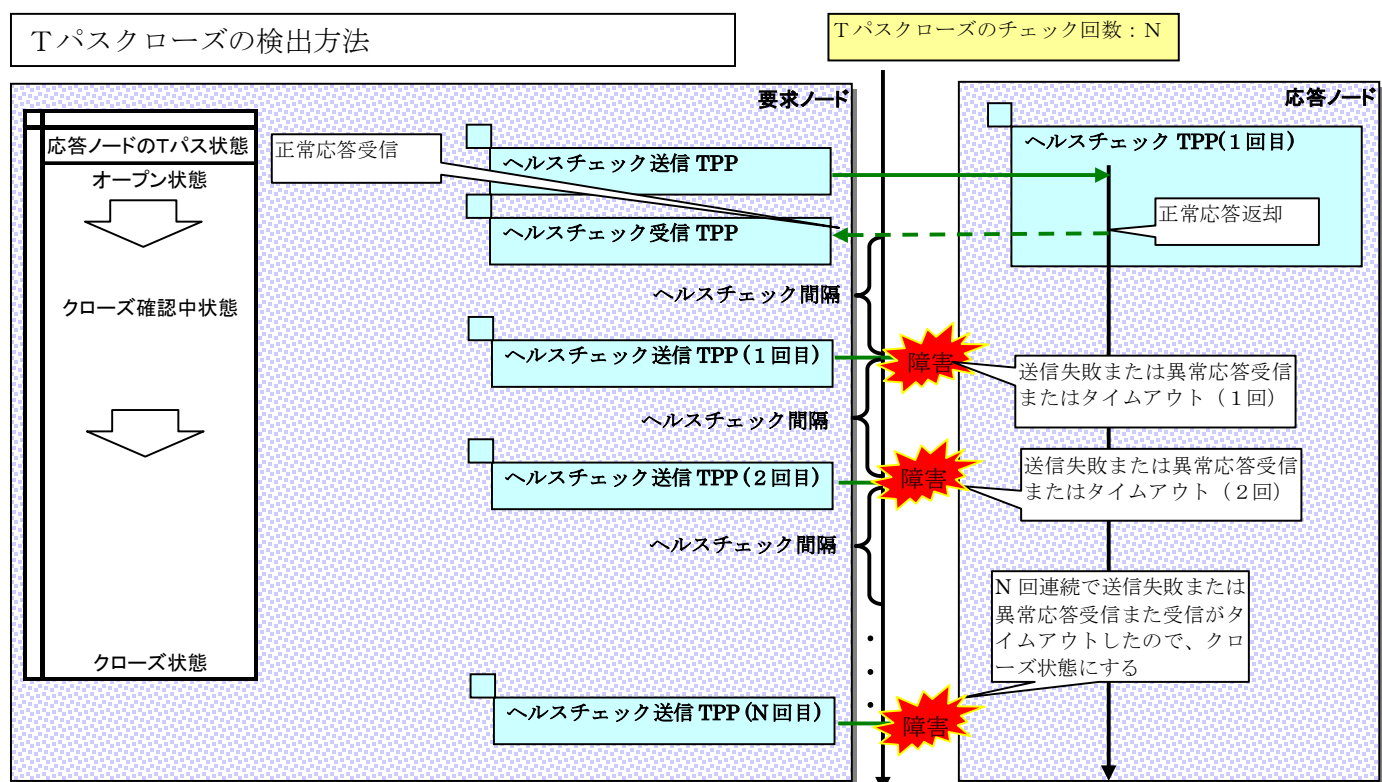


図 2-3 Tパスクローズの検出方法

(3) Tパスオープン確認中のクローズ検出

Tパスクローズ状態で相手ノードから正常応答を受信した場合、オープンの確認中状態になりますが、この時に、相手ノードからの異常応答やタイムアウトを検出した場合は、クローズ状態にします。これは、Tパスクローズ状態のときにネットワークが不安定状態になり、クローズ状態とオープン確認中を繰り返さないようにするためです。また、これにより、クローズ状態に変更されたときに出力される重要メッセージを連続して出力することを抑えることも目的とします。

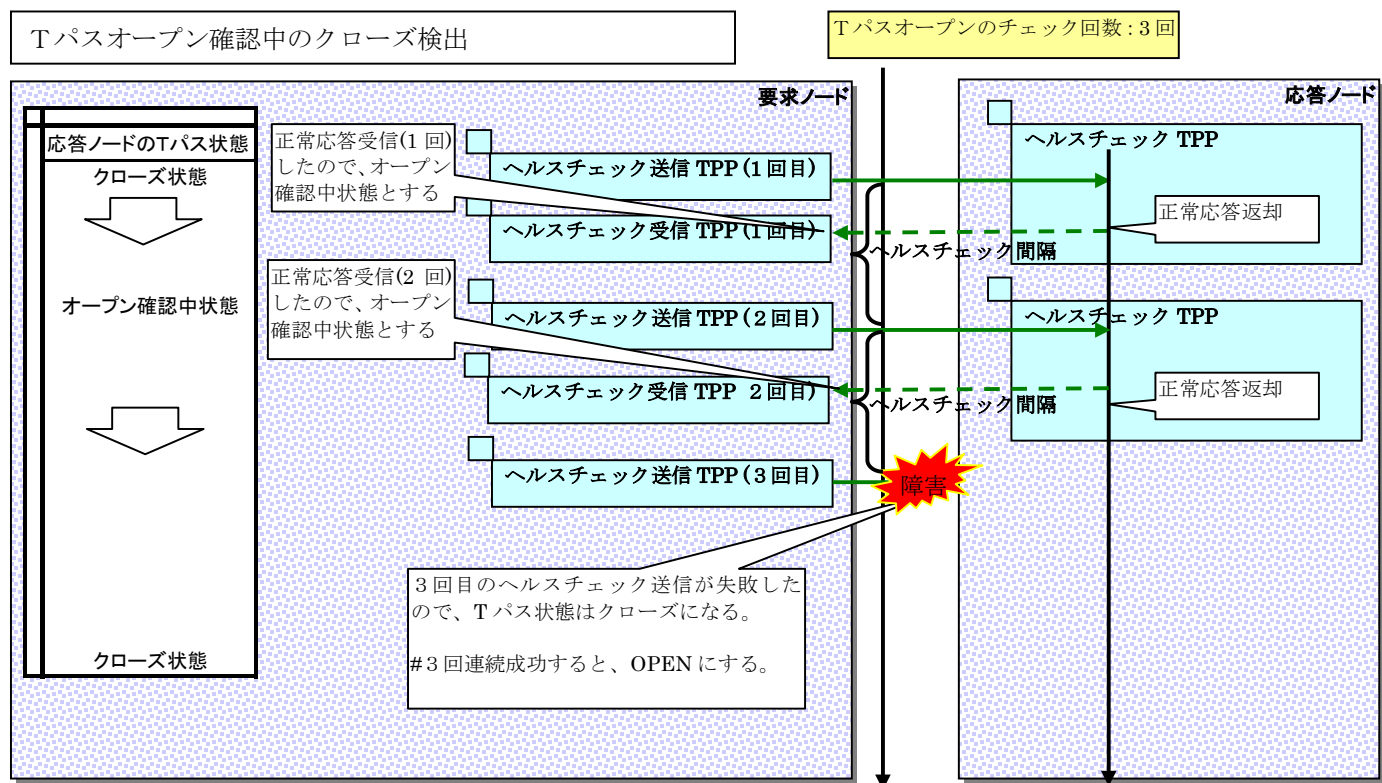


図 2-4 Tパスオープン確認中のクローズ検出

(4) Tパスクローズ確認中のオープン検出

Tパスオープン状態で相手ノードからの異常応答やタイムアウトを検出した場合、クローズの確認中状態になりますが、この時に、相手ノードから正常応答を受信した場合は、オープン状態にします。これは、Tパスオープン状態のときにネットワークが不安定状態になり、オープン状態とクローズ確認中状態を繰り返さないようにするためです。また、これにより、オープン状態と判別したときに送信した電文の欠落を抑えることも目的とします。

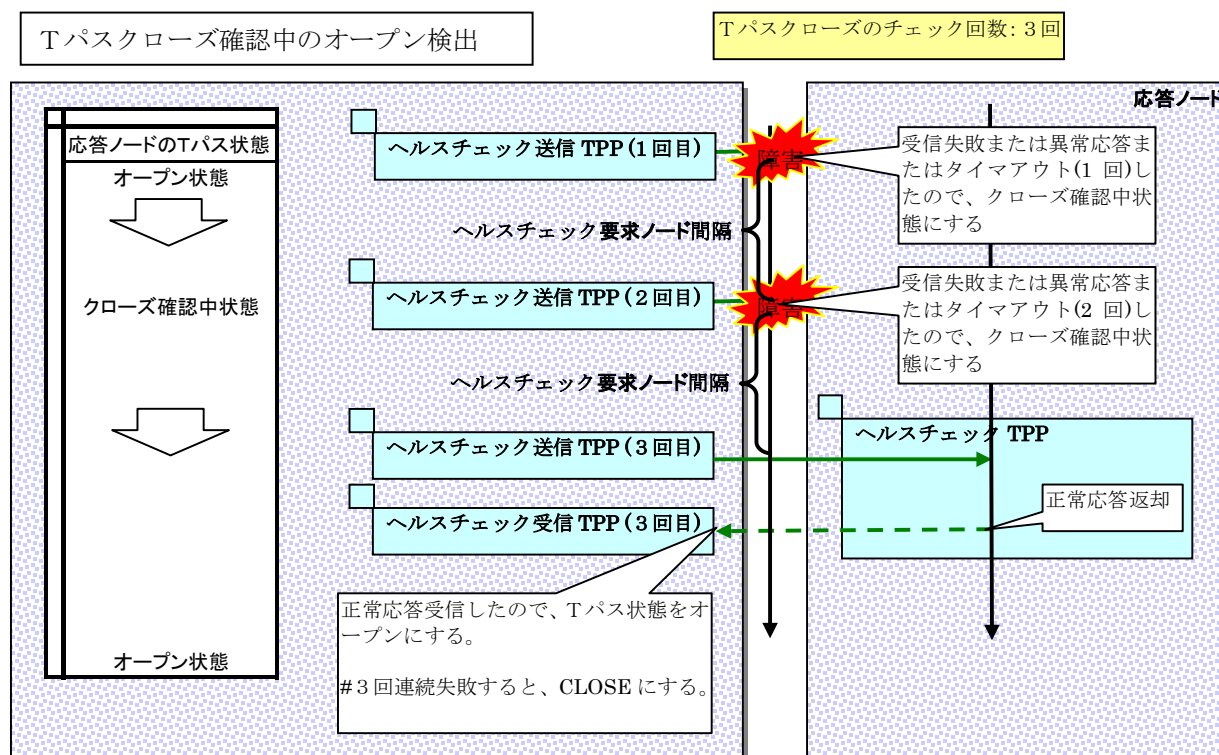


図 2-5 Tパスクローズ確認中のオープン検出

2.11.3 ヘルスチェック間隔の変更

Tパス管理機能では、障害などにより長時間停止中のノードに対しての無駄なヘルスチェック処理を軽減するために、Tパスがクローズ状態の場合のヘルスチェック間隔を通常時と別に定義することができます。

2.11.4 マルチ TPBASE 対応

性能に関するスケーラビリティとして TPBASE を 1 ノードで複数起動し、同一業務を複数の TPBASE で実行することを可能にします。

この場合、Tパスヘルスチェック要求を複数の TPBASE に対して送信し、いずれかの TPBASE から応答があれば、ノード単位にはTパスオープンとします。

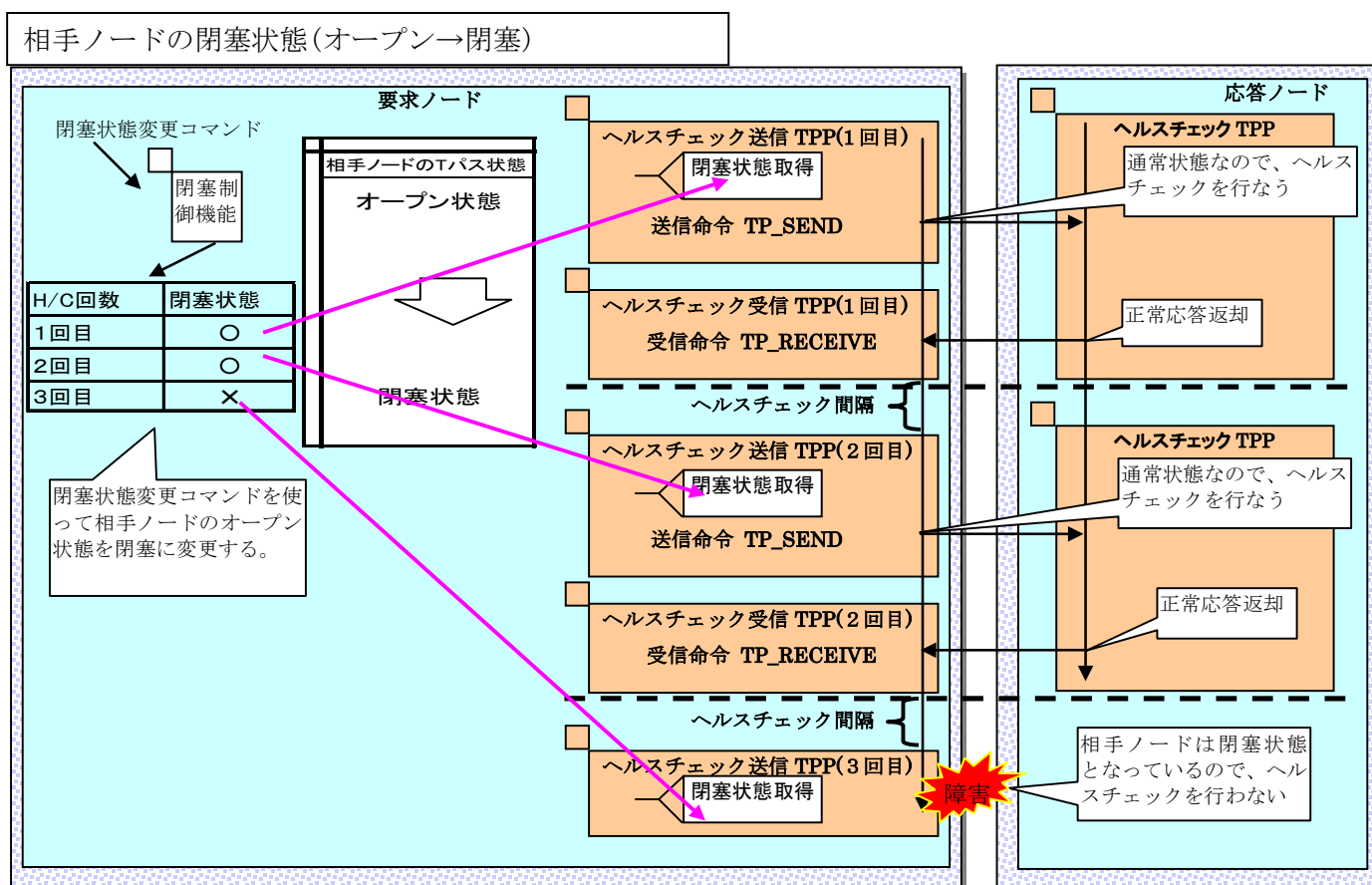


図 2-7 相手ノードの閉塞状態(オープン→閉塞)

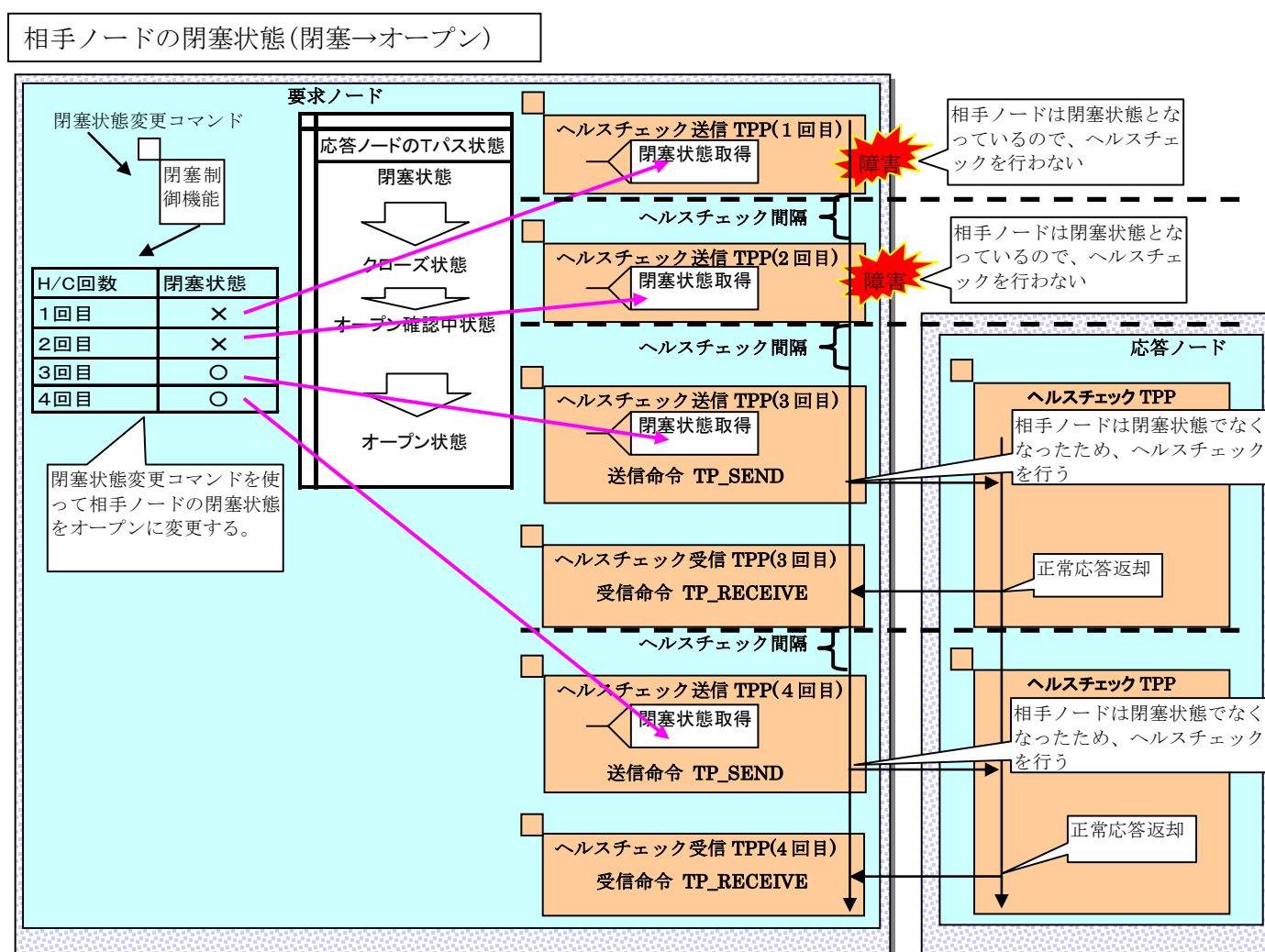


図 2-8 相手ノードの閉塞状態(閉塞→オープン)

また、T パス管理機能の起動コマンドまたは動的変更コマンド実行時に、相手ノードが閉塞中の場合はヘルスチェックを行いません。自ノードが閉塞中の場合はヘルスチェックを行います。

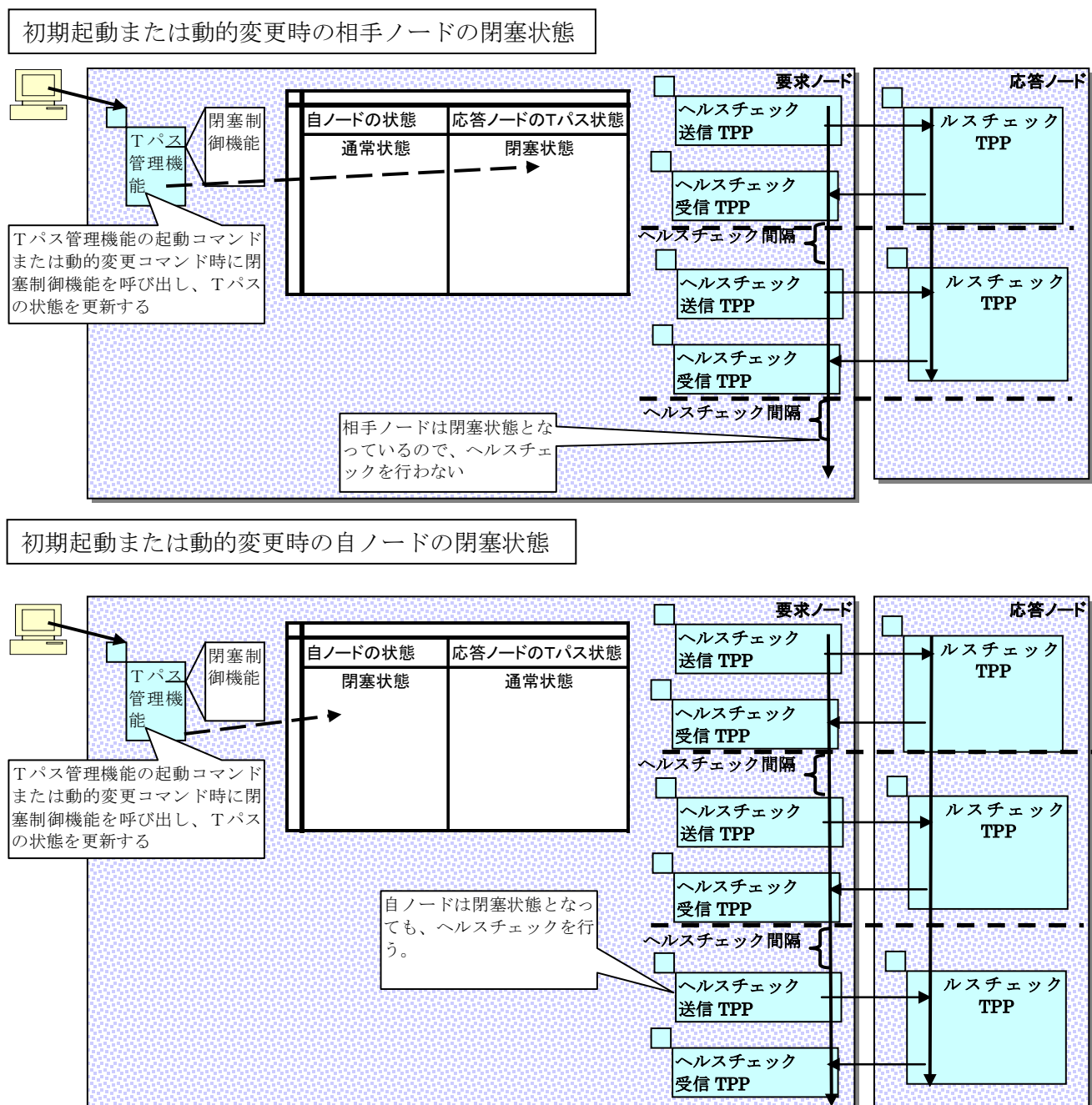


図 2-9 初期起動または動的变化時の閉塞状態

2.11.6 定義の動的変換機能

Tパス管理機能では、Tパスの管理を行うための環境定義を動的に変更する機能を提供します。変更可能な環境定義パラメータは、環境定義に記述された全パラメータです。

また、Tパス管理機能では、ノードの一覧情報をD I O S A M A P節から取得するため、D I O S A M A P節の環境定義の動的変換により、ノードが追加・削除された場合、Tパス管理機能も定義の動的変換を行なう必要があります。

2.12 流量制御機能

A P ノード、O L T P ノードの負荷情報を収集し、該当ノードに電文を送信するノードに情報を展開します。電文を送信する際に、電文流量制御機能が収集したノードの負荷情報を参照し、高負荷なノードには電文を送信しないよう、負荷分散を行います。

A P ノードでは、外部から受信した電文を O L T P ノードに中継する際に、高負荷な O L T P ノードを避けるよう負荷分散します。（ラウンドロビン指定の場合。ノードを指定する送信では指定ノードに送信されます。）

O L T P ノードでは、外部へ送信する電文を A P ノード経由で送信する際に、高負荷な A P ノードを避けるよう負荷分散します。（要求電文の応答など、特定の A P ノードに返信する必要がある場合は除きます）

2.12.1 負荷情報収集

流量制御機能では、ノード毎に下記の負荷情報を収集します。

- T P B A S E のキューの滞留数
- C P U 使用率
- デバイスビジー率
- メモリ使用率
- ディスク使用率

これらの情報に基づき電文流量制御機能は動作します。

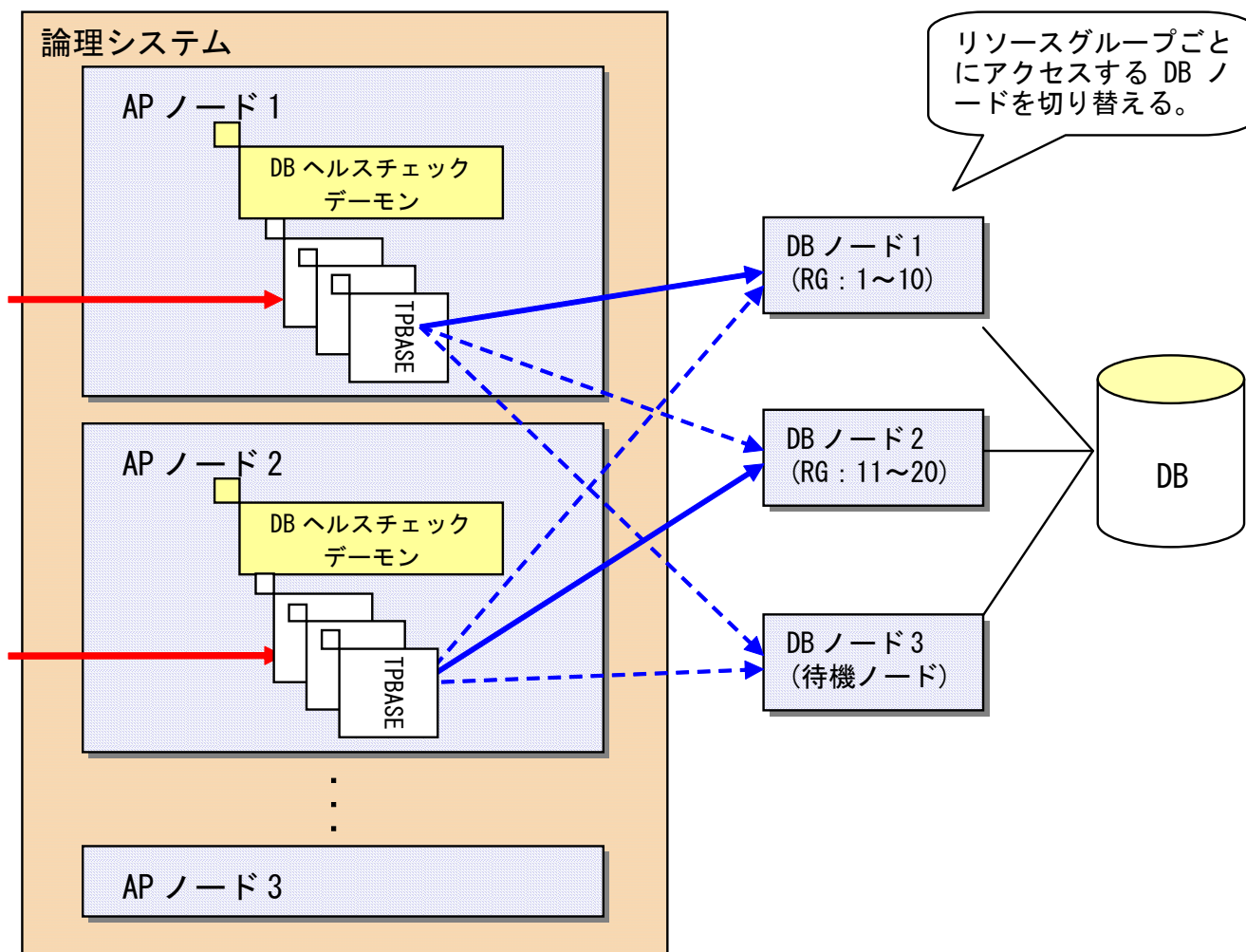
収集された負荷情報は、実際の数値ではなく、環境定義に定義した範囲定義を元に負荷レベルに変換されて管理されます。このいずれかの負荷レベルが環境定義に定義された閾値を超えた場合には高負荷ノードと判断されます。

高負荷と判断されたノードは電文の送信先としての優先度が下げられ、低負荷のノードの中からラウンドロビンにより送信先が選択されます。また、全てのノードが高負荷の場合は、ラウンドロビンにより均等に電文が送信されます。

2.13 データベース管理機能

2.13.1 DBマルチコネクション制御

DIOSA/XTP では、1 プロセスから全てのDBインスタンスへ予めコネクションを接続しておき、APパーティショニングやDBの障害状態などにより、利用するコネクションを切り替えてDBアクセスを行います。コネクションの利用先は、リソースグループ群（パーティション）毎に決定します。



また、DBインスタンスの組み合わせに名前を付けて複数のペアを定義可能です。このペアをインスタンスグループと呼びます。インスタンスグループの定義は環境定義にて行うことができます。

2.13.2 DBインスタンス振り分け機能

DBインスタンス振り分け機能は、RACを構成する複数のDBノードのうち、リソースグループごとに優先的に利用するDBノード（インスタンス）を決定する機能です。OracleのRAC機能は、複数のOracleサーバを経由して同一のDBに整合性を保ちながらアクセスする機能ですが、無秩序にOracleサーバを選択してDBをアクセスした場合、各サーバ上のキャッシュを同期させる処理（キャッシュフュージョン）が発生し、全体的なスループットが低下します。

DIOSA/XTP では、リソースグループごとに通常利用するDBノードを1台に限定することにより、キャッシュフュージョンの発生を抑えます。AP/OLTPノード上のDBヘルスチェックデーモンがDBノードのダウンを検出すると、本機能が稼働中のDBノードに接続コンテキストを切り替える（DB接続先切り替え関数により制御を行う）ことにより、システム全体としてのDBインスタンス切り替えを行います。

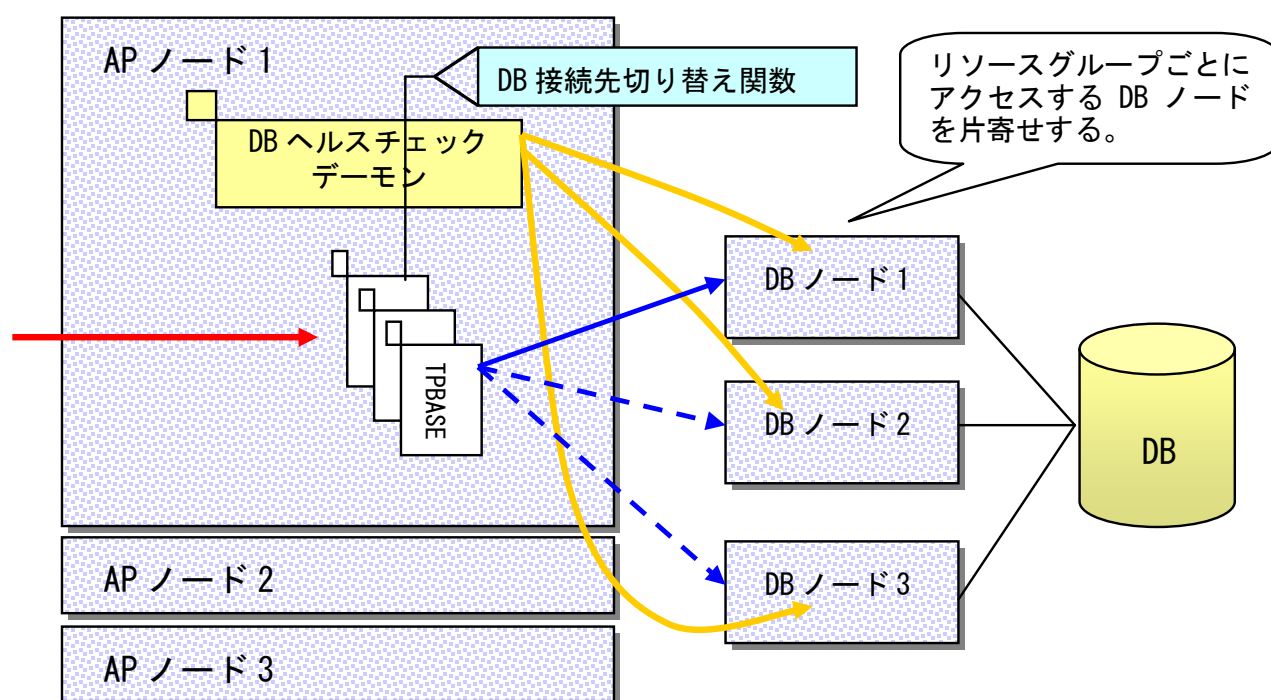


図 2-11 OracleのDBインスタンス振り分け機能

2.13.3 DBヘルスチェック機能

DBヘルスチェック機能は、一定間隔でのOracleのSMONの存在チェック、およびSQLの実行により、データベースインスタンスの稼働状況を確認する機能です。SMONの存在チェックおよびSQL実行の結果は、AP/OLTPノード上のDBヘルスチェックデーモンが、各DBノード上のDBヘルスチェックデーモンと通信を行うことで取得します。AP/OLTPノード上のDBヘルスチェックデーモンがエラーやタイムアウトを検出した場合は、当該データベースインスタンスがダウンしているとみなし、リソースグループが使用するデータベースインスタンスを活性のものに切り替えます。

AP/OLTPノード上のDBヘルスチェックデーモンは、データベースインスタンスのダウンを検出した場合、該当するDBノードを自動的に閉塞します。データベースインスタンスの復旧が完了し、利用が可能になったことを確認した後、DBノードの閉塞を解除してください。閉塞解除後、ヘルスチェックが成功することにより活性状態と判断され利用可能になります。

なお、環境変数「DIOSA_DBINTEGRATE_AUTO」に「YES」を設定しておくことにより、データベースインスタンス

ダウン検出時の自動閉塞処理が行われなくなり、復旧後に直ちに利用可能になります。

また閉塞には、閉塞状態変更コマンドを実行しますが、コマンドがエラーになった場合のリトライ回数を、環境変数「DIOSA_DBBLOCK_RETRY」で調整することができます。

本機能を使用する際は、環境変数「ORACLE_HOME」および「ORACLE_BASE」をO r a c l e環境に合わせて設定する必要があります。

2.13.4 DB関連API

DBの接続・切断などを行うためのAPIを提供します。

データベース関連APIを使用する場合は、事前に「diosaprcinit(プロセス初期化処理)」および「diosathrinit(スレッド初期化処理)」が実行されている必要があります。

Oracleへ常時接続(マルチコネクション)したい場合は、図 2-12 Oracle 常時接続(マルチコネクション)時のAPI使用方法に示す手順でAPIを使用してください。

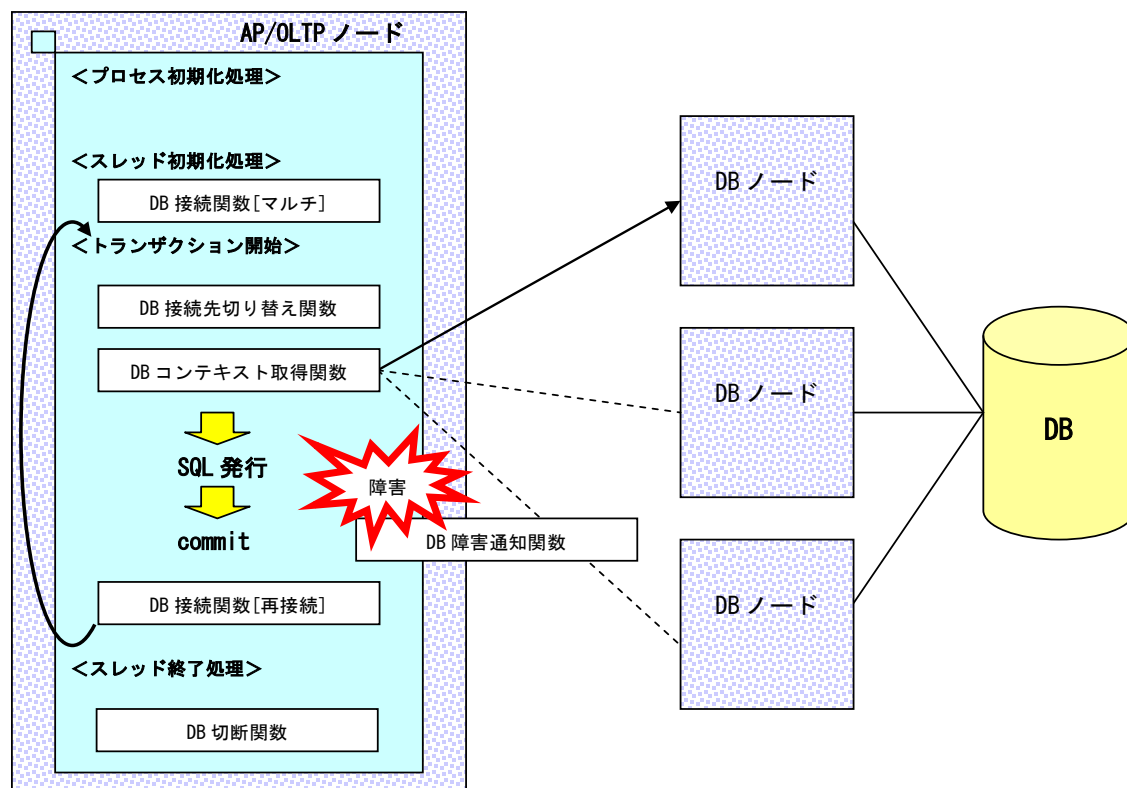


図 2-12 Oracle 常時接続(マルチコネクション)時のAPI使用方法

Oracleへ常時接続(シングルコネクション)したい場合は、図 2-13 Oracle 常時接続(シングルコネクション)時のAPI使用方法に示す手順でAPIを使用してください。

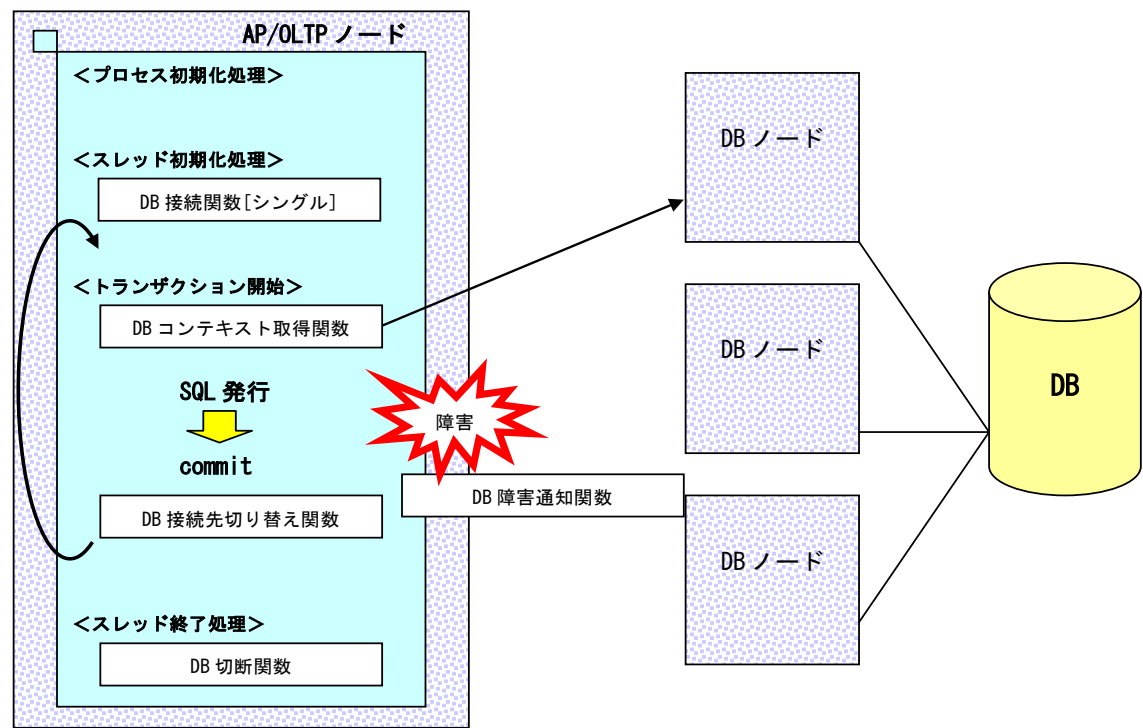


図 2-13 Oracle 常時接続(シングルコネクション)時のAPI使用方法

Oracleへ随時接続したい場合は、図 2-14 Oracle 随時接続時のAPI使用方法に示す手順でAPIを使用してください。

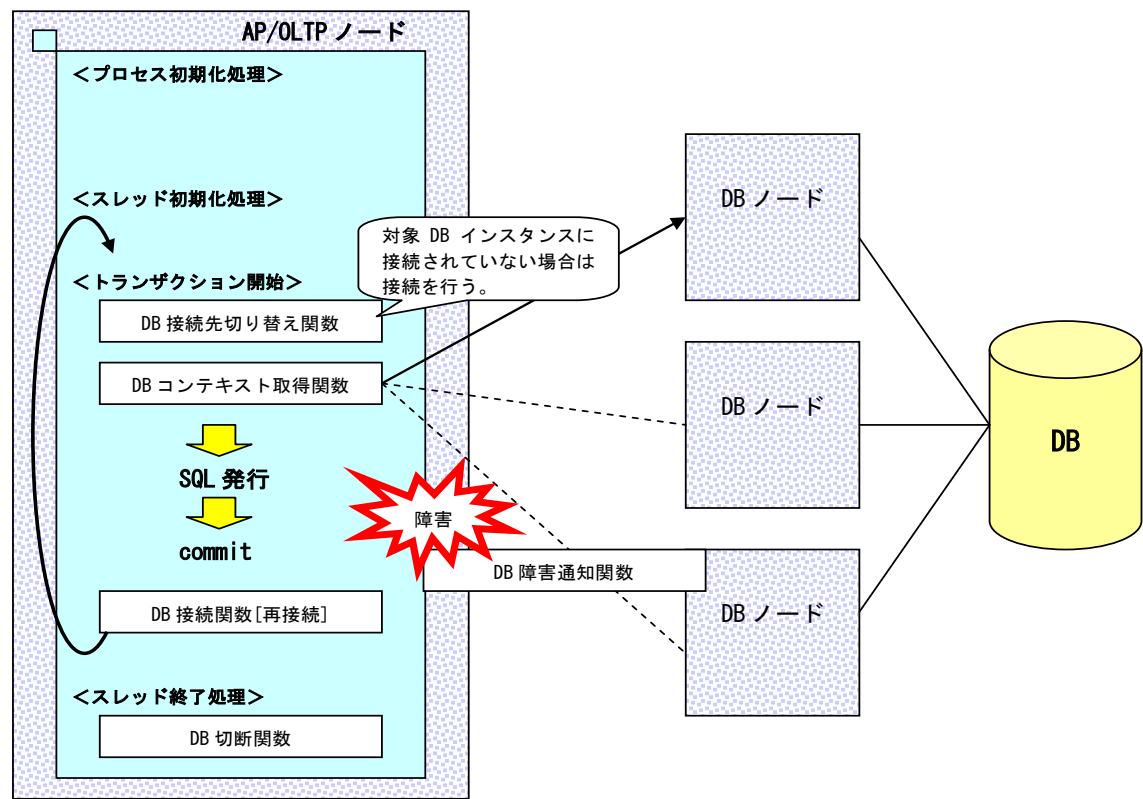


図 2-14 Oracle 随時接続時のAPI使用方法

各APIの説明は以下の通りです。

(1) **データベース接続関数 [マルチコネクション]**

データベースヘルスチェック機能により、活性と判断される全てのデータベースインスタンスに接続します。接続処理の多重度制御は行わずにコネクションの確立を待ち合わせますが、全てのデータベースインスタンスへの接続に失敗した場合はエラーリターンします。（接続処理の多重度制御については、データベース接続関数 [再接続] の項を参照してください）

※ マルチスレッドにおいてはスレッド単位に本関数を実行する必要があります。

(2) **データベース接続関数 [シングルコネクション]**

バッチAPI制御などでシングルコネクション接続する際に使用します。シングルコネクション接続時は、データベース接続先切り替え関数を使用しなくても即時にデータベースアクセスを行うことができます。

※ マルチスレッドにおいてはスレッド単位に本関数を実行する必要があります。

(3) **データベース接続先切り替え関数**

トランザクションの開始時に本関数を呼び出して、指定されたリソースグループIDまたはリソースグループセット名に対応するデータベースインスタンスに接続コンテキストを切り替えます。また、対象となるDBインスタンスに接続されていない場合は、本関数内で接続を行います。

トランザクション処理を行うデータベースインスタンスとの接続状態のチェックには、通常の接続状態チェックのほかに、データベース管理表で管理しているデータベースインスタンス毎の接続世代管理番号を使用します。前回接続確立時の接続世代管理番号と、現在のデータベース管理表の接続世代管理番号を比較して、異なる場合はコネクションが切断されているとみなし、再接続を行います。再接続処理は即座に行う必要があるため、多重度制御は行いません。（接続処理の多重度制御については、データベース接続関数 [再接続] の項を参照してください）

トランザクション処理を行うデータベースインスタンス以外については接続状態のチェックのみ行い、本関数の戻り値として、データベース接続関数 [再接続] を実行する必要があるかどうかを返却します。

本関数は、マルチコネクション・シングルコネクションに関わらず利用することができます。シングルコネクションの場合は、対象となるデータベースインスタンスが初期接続されていたデータベースインスタンス以外であれば、本関数で接続を行い、マルチコネクションの状態となります。

(4) **データベース接続関数 [再接続]**

本関数は、データベース接続先切り替え関数の戻り値で、再接続の必要があると判断された場合に実行することを想定しています。データベース接続先切り替え関数が、トランザクション実行対象のデータベースインスタンスとの再接続処理のみを行うのに対し、本関数は自プロセスと全てのデータベースインスタンスとの接続状態のチェックを行い、コネクションが切断しているものに対して再接続処理を行います。なお、再接続時は接続処理の多重度制御を行います。最大多重度を超過した分の接続処理は、次のデータベース接続関数 [再接続] の中で行うものとします。

・ 接続処理の多重度制御について

データベースへの接続要求の集中を避けるため、システム全体でのデータベース接続要求の最大値を制限する制御を行います。環境定義で定義された最大多重度を超過した場合は接続処理を行いません。

(5) **データベース切断関数**

データベースとの接続を切断します。

※ マルチスレッドにおいてはスレッド単位に本関数を実行する必要があります。

(6) **データベースコンテキスト取得関数**

カレントデータベースの接続コンテキストを返却します。

マルチコネクション接続時は、事前にデータベース接続先切り替え関数を実行して、接続先データベースインスタンスのコンテキストを決定する必要があります。

(7) **データベース障害通知関数**

コネクションを使用しているプロセスがSQLのCOMMITやROLLBACKの失敗によりデータベースインスタンスの障害を検出した場合に、本関数を使用して自プロセスとデータベースインスタンスとの接続を切断します。

2.14 閉塞管理機能

閉塞管理機能は、ノードや C0 等の閉塞状態を管理し、変更、参照をおこなうためのインタフェースを提供する機能です。閉塞の管理対象は、ノード、C0 があり、これらの閉塞制御を利用することにより、細やかなアプリケーションの制御を行うことが可能です。

なお、ノードや C0 の閉塞状態は、ノード間で自動的に同期が行われます。

また、閉塞状態は引き継ぐことが可能です。前回起動時の閉塞状態を引き継いで、DIOSA を再起動させることができます。

- ノード閉塞

ノード閉塞の状態として閉塞と予閉塞が存在し、ノード全体を閉塞します。

ノード閉塞の状態が閉塞の場合、他ノードから閉塞中のノードへの電文の送信や、閉塞中のノード内での TPP 間派生による電文の送信は行えません。ノード閉塞の状態が予閉塞の場合は、送信関数(diosasendtx)のパラメータの指定により、他ノードから予閉塞中のノードへの電文の送信（予閉塞中のノードに送信するか否か）を制御することができます。

なお、閉塞中のノード及び予閉塞中のノードから他ノードへの電文の送信は行えます。

- C0 閉塞

指定された C0 のみを閉塞します。特定のアプリケーションのみ動作させたくない場合に利用します。

2.14.1 機能説明

閉塞管理機能では、環境定義に定義されている、ノード、C0 について論理的な閉塞/閉塞解除を管理します。閉塞情報の引継ぎ情報を引き継ぎファイル上に持っているため、障害発生後の DIOSA 再起動の際に、前回運転時の閉塞状態を引き継ぐことができます。

(1) 初期閉塞状態

ノード閉塞状態に関しては DIOSA 起動時の閉塞状態の初期値を設定することができます。ノードの初期閉塞状態は、環境定義の DIOSAMAP セクションに記述します。C0 に関しては、初期状態は全て活性状態となります。

環境定義中の初期閉塞状態はコールドスタートした場合のみ反映され、ウォームスタート時は、前回運転時の閉塞状態（最後に閉塞状態更新情報を保持するノードの閉塞状態※1）が引き継がれます。

※1 起動時、論理システム内の全ノードで閉塞状態同期が行なわれ、最後に実施した閉塞状態情報を保持したノードの閉塞状態に同期されます。

(2) 閉塞状態更新

閉塞状態の更新は閉塞状態変更コマンド(dibcmupd)で行います。コマンド実行時、閉塞状態の更新はコマンド実行ノードが属する論理システム内の全てのノードに対して行われます。

(3) 閉塞状態参照

閉塞状態の参照は閉塞状態参照コマンド(dibcmref)で行います。

(4) **閉塞状態同期**

閉塞状態の同期は自動的に行われますが、何らかの理由により、ノード間の閉塞状態の情報に差異が発生した場合、閉塞状態同期コマンド（dibcmsync）を利用することで閉塞状態の同期が行えます。本コマンドにより、論理システム内で最新の閉塞情報を保持しているノードの閉塞状態に同期されます。

2.15 コマンド配信機能

コマンド配信機能は、指定された配信先に、指定されたコマンドを配信して実行し、その結果を確認可能な機能です。

配信先として、論理ノード単位、サーバグループ単位（複数論理ノードをまとめた単位）、論理システム単位（A P ノード群、DB ノード群、O L T P ノードをまとめた単位）が指定できます。

また、転送先の論理システム名を指定することにより、外部論理システムの配信先を指定することができます。

配信コマンドとして、U N I X コマンド、D I O S A / X T P コマンドが指定できます。

実行結果（s t d o u t、s t d e r r）がある場合は、実行結果の待合せを行い、コマンドを入力したノードに結果を出力することが可能です。

2.15.1 コマンド配信ユーザインタフェース

コマンド配信を利用するために、以下のA P I およびコマンドが提供されています。

(1) **C関数**

d i o s a c m d s e n d : コマンド配信を要求します。
d i o s a c m d c o n f : コマンド配信の結果を確認します。

(2) **オペレータコマンド**

d i c m d s e n d : コマンド配信を要求します。

2.15.2 コマンド配信宛先

(1) **転送先の外部論理システム名**

転送先の外部論理システム名を指定することにより、他サブシステム（S Y S M A P 節に定義されている論理システム）へコマンドを配信することができます。外部論理システム名と同時に、(2)～(6)で示す宛先情報を指定することで、他サブシステム上の任意の論理ノードに配信することができます。

サブシステム内の宛先を指定する場合は、転送先の外部論理システム名を省略します。

(2) **配信宛先**

コマンド配信宛先には以下の指定方法があります。

(a) 論理ノード指定

特定の論理ノードにコマンドを配信する場合に指定します。

(b) サーバグループ指定

特定のサーバグループにコマンドを配信する場合に指定します。

サーバグループとは、複数の論理ノードの集合を表します。環境定義C M D S E N D 節のS R V G R P 項にサーバグループ名と、サーバグループを構成する論理ノードを定義することで利用できます。

(c) 論理システム指定

特定の内部論理システム（D I O S A M A P 節に定義されている論理システム）にコマンドを配信する場合に指定します。

(3) **論理ノード属性種別**

特定の論理システムに配信する場合、配信対象とする論理ノード属性を指定することができます。

(a) 論理システム配下全ノード

指定論理システム配下の全ノードを配信対象とします。

(b) A P ノード

指定論理システム配下のA P ノードを配信対象とします。

(c) O L T P ノード

指定論理システム配下のO L T P ノードを配信対象とします。

(d) D B ノード

指定論理システム配下のD B ノードを配信対象とします。

(4) **配信対象種別**

複数ノードへの配信の場合、配信対象を指定することができます。

(a) A L L 型

指定した配信先ノードのうち、全ノードを対象とします。

(b) A N Y 型

指定した配信先ノードのうち、任意の 1 ノードを対象とします。

(5) **配信元ノードの配信除外指定**

複数ノードへの配信で配信元ノードも配信対象に含まれる場合、配信元ノードを配信対象から除外することができます。

配信宛先名に論理ノード名を指定した場合、あるいは配信元ノードが配信対象に含まれない場合、この指定は無視されます。

(6) **閉塞状態チェック**

A P I またはコマンドのパラメータで、閉塞チェックの有無および閉塞状態のノードへの配信結果を含めるか否かを指定できます。ただし、環境定義 (C M D S E N D I N F O 節の B L O C K C H K) の定義値に N O を指定している場合は、A P I またはコマンドの指定に関わらず閉塞チェックを行いません。

閉塞チェックなしの場合は、閉塞状態に関わらず、指定されたノードへのコマンド配信を行います。閉塞チェックありの場合は、閉塞状態または予閉塞状態のノードへのコマンド配信は行いません。

閉塞状態のノードへの配信結果を含める指定の場合は、要求元には閉塞状態のノードも含めた全配信先の結果を返却します。配信先に閉塞のノードが存在する場合は、結果ステータスは異常になります。

閉塞状態のノードへの配信結果を含めない指定の場合は、要求元には閉塞状態のノード以外の配信先の結果のみを返却します。配信先に閉塞のノードが存在する場合は、閉塞以外のノードへの配信が全て成功していれば結果ステータスは正常終了になります。ただし、配信対象の全ノードが閉塞状態の場合、要求元には宛先不正エラーを返却します。

(7) **指定例**

配信先指定例と配信先ノードの一覧を以下に示します。

2.15.3 コマンドルーティング

コマンドルーティングは、コマンドに対する配信先情報を環境定義に登録することにより、コマンドやAPIの配信先パラメータを省略することができる機能です。これにより、配信宛先を意識することなくコマンド配信を行うことができます。

コマンドルーティングの使用例を以下に示します。

コマンドルーティング定義例

```
%CMDRT
    NAME      = dixxxcmd
    LSTYPE    = AP
    TARGET    = ALL
;
```

コマンド配信例 1

```
dicmdsend -t dixxxcmd -d LS01
```

この例では、配信宛先パラメータとして宛先名（論理システム指定）のみを指定しています。この場合は、パラメータとして指定されていない情報のみをコマンドルーティングの定義で補います。よって、論理システムLS01配下の全APノードに対してコマンド配信を行います。

コマンド配信例 2

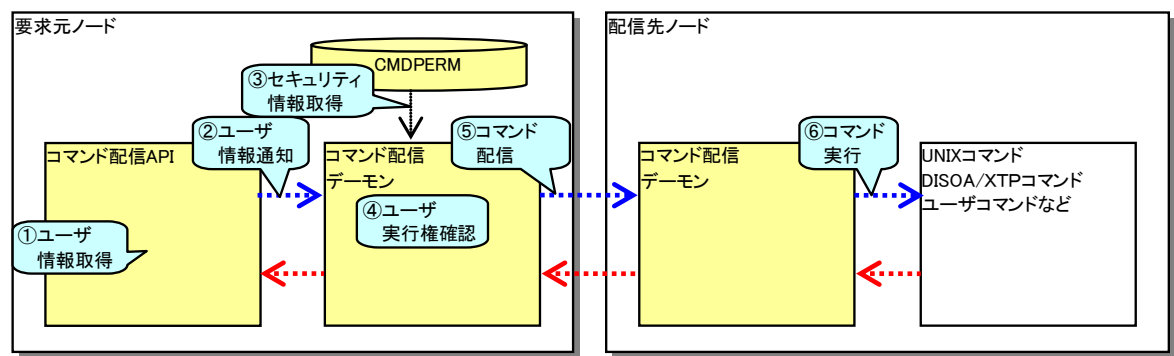
```
dicmdsend -t dixxxcmd
```

この例では、配信宛先パラメータを一切指定していません。そのため、宛先情報はコマンドルーティングの定義に従います。宛先名が省略され、論理ノード属性種別が指定されている場合は、自論理システムを配信先とします。よって、自論理システム配下の全APノードに対してコマンド配信を行います。

2.15.4 コマンド実行権限

コマンド配信要求を受け付けた際に、コマンド配信を要求したユーザおよびユーザグループが環境定義C M D S E N D節のC M D P E R M項でコマンド実行許可されているかどうかの確認を行います。環境定義においてコマンドの実行が許可されていない場合、コマンド配信を行うことができません。

これにより、特定のユーザおよびユーザグループに対して配信可能なコマンドを制限することができます。



コマンド実行権の使用例を以下に示します。

コマンド実行権定義例

```
%C M D P E R M
    D F L T P E R M = N O E X E C      ←未登録コマンドの実行不可

%C M D G R P
    N A M E = C M D G R P 0 1
    %C M D T E X T T E X T = d i c m d x x 0 1;
    . . .
    . . .
;

%C M D G R P
    N A M E = C M D G R P 0 2
    %C M D T E X T T E X T = d i c m d x x 0 2;
    . . .
    . . .
;

%E X P E R M
    U S E R T Y P E   = U S E R
    N A M E           = u s e r 1
    P E R M I S S I O N = E X E C
    %C M D   C M D G R P = C M D G R P 0 1;
;

%E X P E R M
    U S E R T Y P E   = U S E R G R P
    N A M E           = g r o u p 1
    P E R M I S S I O N = E X E C
    %C M D   C M D G R P = C M D G R P 0 2;
;
```



```
%EXPERM
    USERTYPE    = USERGRP
    NAME        = group2
    PERMISSION  = EXEC
    %CMD CMDGRP = CMDGRP02;
;
;
```

この例では、ユーザ “u s e r 1”、ユーザグループ “g r o u p 1、g r o u p 2” に対して実行可能なコマンドグループを定義し、未登録コマンドは実行不可としています。

各ユーザに関する動作は以下のようになります。

u s e r 1

コマンドグループ “CMDGRP01” に定義されたコマンドのみ実行することができます。その他のコマンドは実行することができません。

g r o u p 1に属するユーザ

コマンドグループ “CMDGRP02” に定義されたコマンドのみ実行することができます。その他のコマンドは実行することができません。

g r o u p 2に属するユーザ

コマンドグループ “CMDGRP02” に定義されたコマンドのみ実行することができます。その他のコマンドは実行することができません。

その他ユーザ

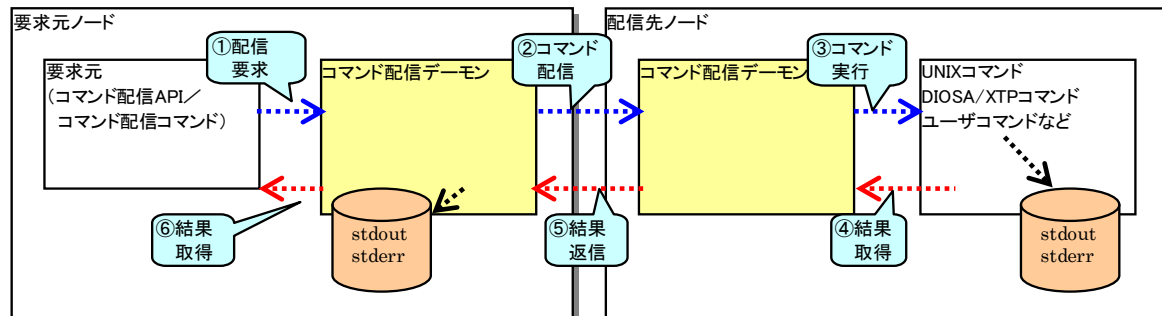
上記に含まれないユーザについては、全てのコマンドが実行不可となります。

2.15.5 コマンド配信結果

コマンド配信結果の確認方法は以下の通りです。

(1) 確認型

コマンド配信を行い、結果の確認が必要な場合に使用します。配信先で実行したコマンドが標準出力（`stdout`）、標準エラー（`stderr`）に結果を出力した場合は、要求元ノードに実行結果ファイルを作成し、要求元APIにファイルパスを返却します。

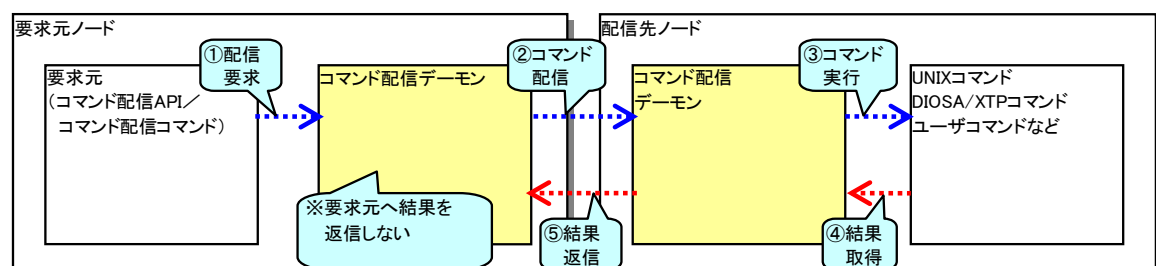


この場合、要求元では結果が返信されるまで、待合せを行います。

なお、コマンド配信コマンドを使用した場合は、必ず確認型のコマンド配信を行い、実行コマンドが標準出力、標準エラーに出力した情報をコンソールに出力します。

(2) 未確認型

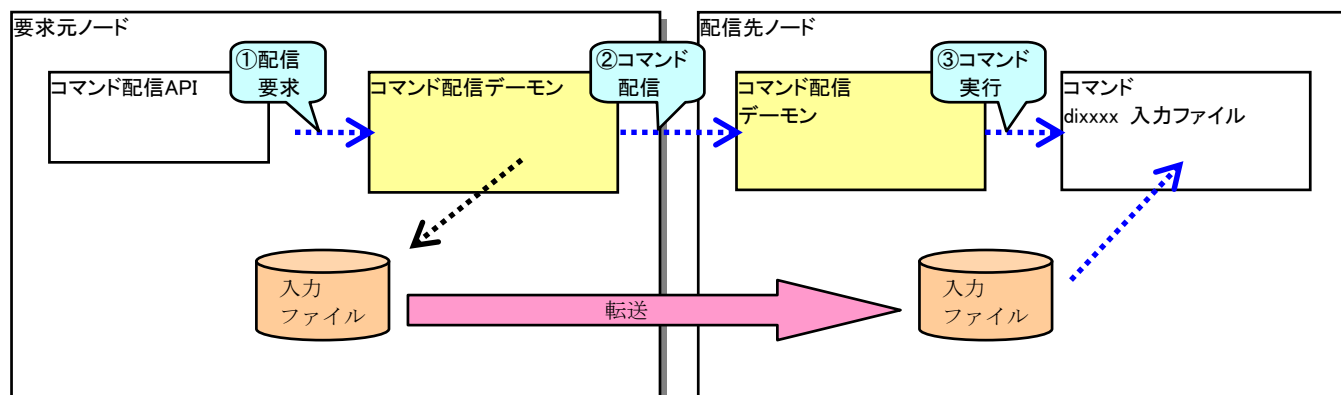
コマンド配信のみを行い、結果の確認が必要ない場合に使用します。



この場合、要求元では結果の待合せは行いません。

2.15.6 入力ファイル転送

配信コマンドを実行する際に入力ファイルが必要な場合、配信元ノードで指定したファイルを配信先ノードに転送し、コマンドのパラメータとして付加することができます。



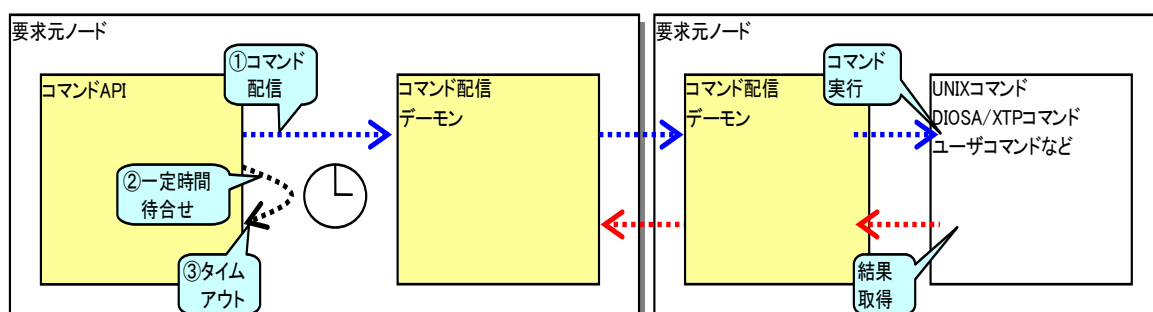
配信元の入力ファイルおよび配信先に転送した入力ファイルは、コマンドの実行が完了した後、削除することができます。

2.15.7 タイムアウト監視

コマンド配信において、プロセスのストール、システムダウンなどにより、配信結果が返ってこない場合があります。このような場合を考慮して、コマンド配信API（コマンド配信コマンドを含む）とコマンド配信デーモンにおいて、結果が返ってくるまでタイムアウト監視を行います。

(1) 配信応答のタイムアウト監視

要求元ノードのコマンド配信APIは、コマンド配信デーモンへ配信要求後、一定インターバル待合せ、その間応答がない場合はタイムアウトとして処理します。

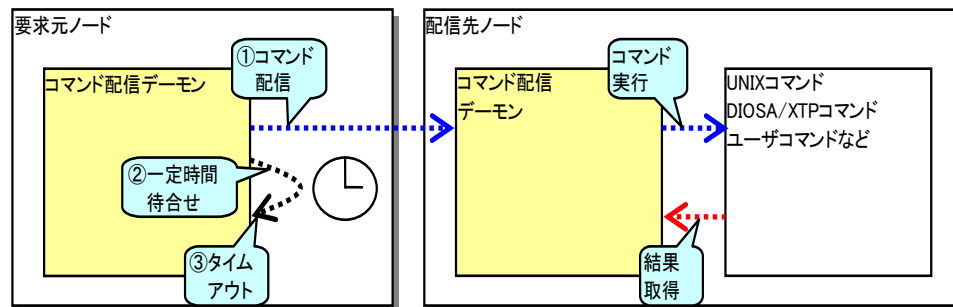


下記の3つのタイムアウト時間の指定が可能で、上位から優先順位が高くなります。

- ①コマンド、及びAPIのパラメータ値
- ②環境変数DIOXA_CDDAPITIMEOUT値
- ③環境定義CMDSSEND節のCMDSSENDINFO項のAPITIMEOUTパラメータ値

(2) 実行応答のタイムアウト監視

要求元ノードのコマンド配信デモンは、配信先ノードへコマンド配信後、一定インターバル待合せ、その間応答がない場合はタイムアウトとして処理します。

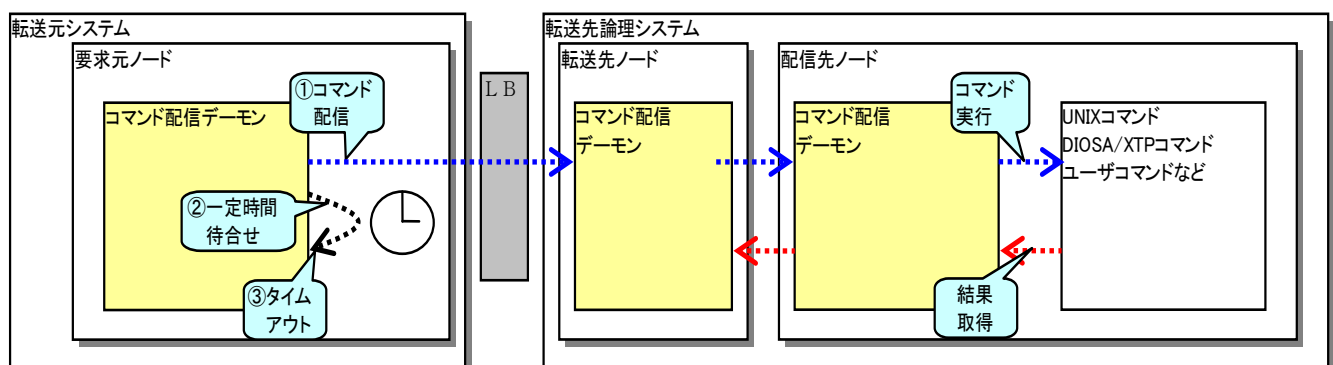


下記の3つのタイムアウト時間の指定が可能で、上位から優先順位が高くなります。

- ①コマンド、及びAPIのパラメータ値
- ②環境変数DIOSA_CDDEJECTIMEOUT値
- ③環境定義CMDSEND節のCMDSENIINFO項のEJECTIMEOUTパラメータ値

(3) 転送先システムからの応答待ちタイムアウト監視

外部論理システムへのコマンド配信の場合、要求元ノードのコマンド配信デモンは、転送先システムへ要求送信後、一定インターバル待合せ、その間応答がない場合はタイムアウトとして処理します。



タイムアウト時間の指定は、コマンドのパラメータ値で指定が可能です。

2.15.8 リトライ処理

コマンド配信に失敗した場合は、リトライ指定をすることで、リトライ処理を行うことができます。

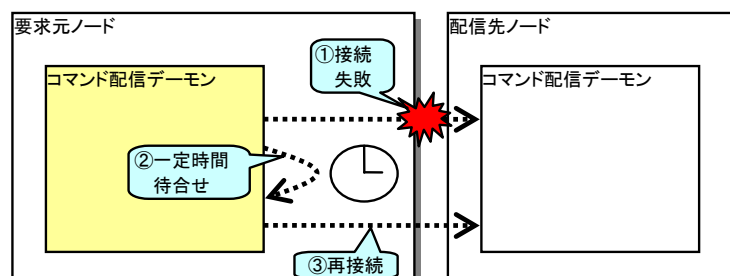
リトライ処理では、一定インターバル待合せ後に、再配信を行います。さらに、リトライ処理は指定されたリトライ回数のみ行います。

ただし、リトライの最大処理時間は（リトライ数×インターバル）秒となり、特に配信結果を確認型で取得する場合、この間はW A I T状態となるので注意が必要です。

(1) A L L型配信の場合

単一論理ノード、または論理システム、サーバグループA L L型指定の配信処理において、以下のような事象が発生した場合、一定インターバル待ち合わせ後に、該当処理を再度行います。

- 配信先ノード（配信先論理システム）が閉塞状態
- 配信先ノードへの接続（c o n n e c t）に失敗
- 配信先ノードへの送信（s e n d）に失敗

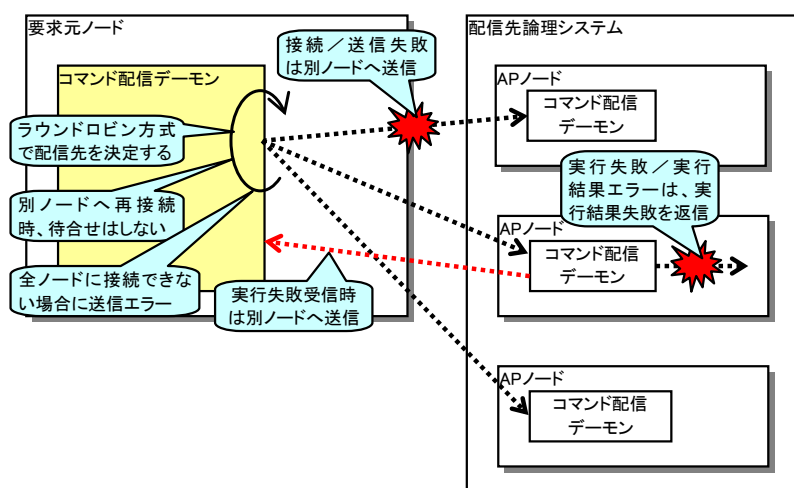


(2) A N Y型配信の場合

論理システム、サーバグループA N Y型指定の配信処理において、以下のような事象が発生した場合、リトライ指定にかかわらず、配信対象の別ノードに対して配信を行います。

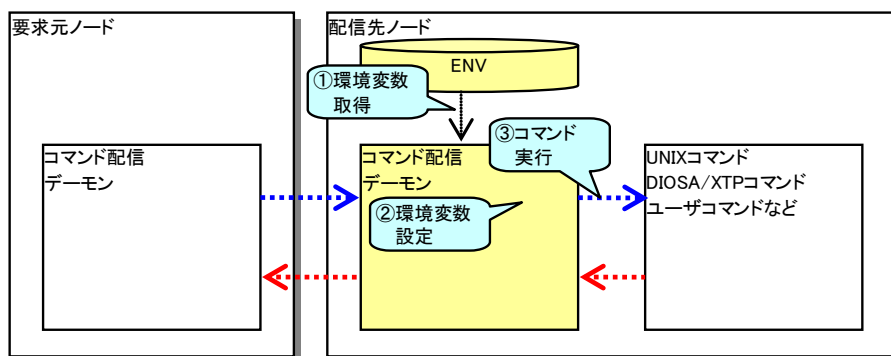
- 配信先ノード（配信先論理システム）が閉塞状態
- 配信先ノードへの接続（c o n n e c t）に失敗
- 配信先ノードへの送信（s e n d）に失敗
- 配信先ノードでコマンド実行に失敗

この場合、配信対象となる全ノードに配信できない場合に送信エラーとなり、リトライ指定があれば、一定インターバル待合せ後、再度上記配信処理を行います。



2. 15. 9 環境変数設定

コマンド実行時、環境定義C M D S E N D節のE N V項に環境変数が定義されていれば、環境変数の設定を行った後、コマンドを実行します。



2. 15. 10 コマンド配信履歴

コマンド配信履歴はコマンド配信の履歴を履歴情報として採取することができる機能です。コマンド配信履歴を採取することにより、過去にどのようなコマンド配信要求があり、どのような状況で終了したか確認することができます。

環境定義C M D S E N D節のC M D S E N D I N F O項にコマンド配信履歴を採取するための情報を定義することにより、利用することができます。

(1) 採取情報

コマンド配信履歴として採取する主な情報には以下のものがあります。

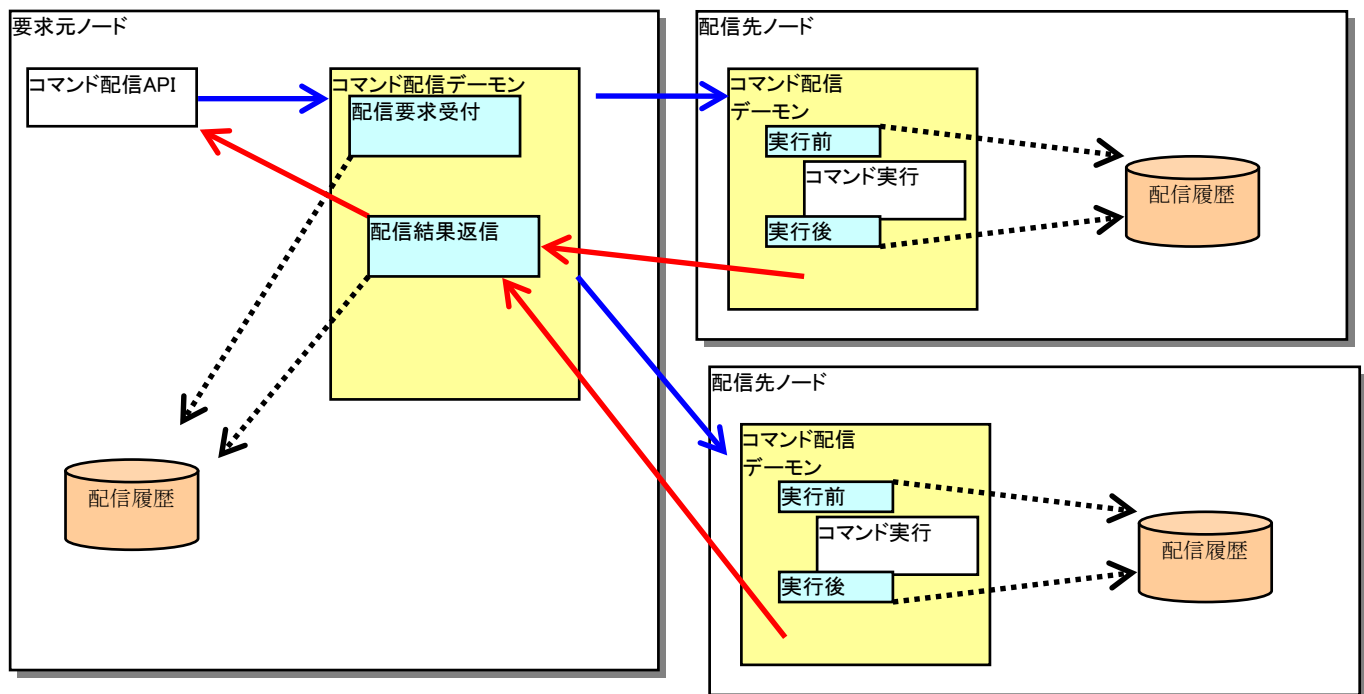
- 日時
- ユーザ名
- コマンド
- 配信先指定／実行ノード名
- 実行結果（ステータス） など

(2) 採取契機

コマンド配信デーモン起動時に、環境定義にコマンド配信履歴を採取するための情報が定義されていれば、自動的にコマンド配信履歴の採取を開始します。また、コマンド配信デーモン停止時には、自動的に採取を停止します。

採取開始中であれば、以下の契機でコマンド配信履歴を採取します。環境定義により、採取する契機を指定することができます。

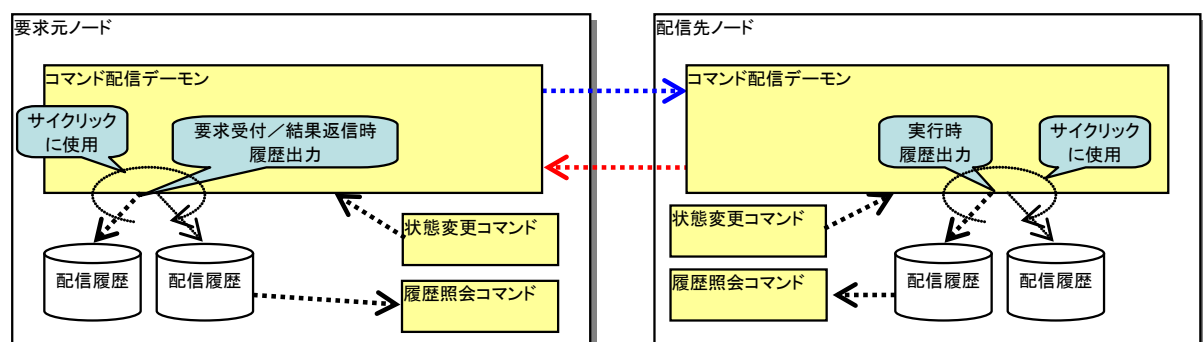
- コマンド配信要求受付時
- コマンド配信結果返信時（配信先ノード数分）
- コマンド実行前
- コマンド実行後（実行結果）



(3) 採取方式

コマンド配信履歴ファイルは、環境定義により 2～7 個まで指定でき、サイクリックに使用します。

ファイルサイズが、環境定義に指定したサイズを超えた場合は自動的にスワップを行い、以前の情報は削除します。



(4) コマンド

(a) ファイル初期化

コマンド配信履歴ファイルを削除します。コマンド配信デーモンが未起動時、またはコマンド配信デーモンが起動中で履歴採取状態が停止中の場合のみ行うことができます。

(b) 履歴採取開始/停止

コマンド配信履歴の採取開始および停止を行います。

(c) 強制スワップ

コマンド配信履歴ファイルのファイルサイズに関わらず、ファイルを強制スワップすることができます。

(d) コマンド配信履歴ファイル編集機能

コマンド配信履歴ファイルを編集出力する機能です。抽出条件を指定することで、特定の履歴情報のみを確認することができます。抽出条件には、論理ノード名、処理通番、開始日時等が指定できます。また、コマンド制御機能を利用した C O 実行時の履歴情報を抽出することが可能です。

第3章 アプリケーション開発

3.1 COプログラミング

本章では、CO制御上で動作するアプリケーションの開発方法について説明します。

3.1.1 プログラムの構造

大規模トランザクションシステム上のアプリケーション開発においては、以下の問題点の解決を図る必要があります。

- ・ 高度で多彩なアプリケーション処理要求

アプリケーションに要求されている機能が非常に複雑で高度なものとなっています。多様な新製品、複合化商品開発をタイムリに行う必要があります、開発の即応性・生産性・保守性の向上が要求されます。

- ・ 高度で複雑なシステム構成要求

規模の面、地理的な面から必然的に複数ホスト、複数システムにまたがる分散アプリケーションシステムとなります。このことによりアプリケーション構造がさらに高度となり、データ処理中心からデータ処理を管理する制御処理中心とならざるを得ません。

- ・ 高度な耐更性の要求

アプリケーションの機能追加から、ハードウェア構成変更まで多様なシステム変更に対する耐更新性が要求されます。

CO制御はイベント（メッセージ）駆動型プログラミングを採用しています。共通関数呼び出しで業務APを構築しようとするよりも、COをイベントで動作させることで、障害の局所化やプログラムを簡素にすることが可能となります。

3.1.2 CO制御とのインタフェース

(1) 呼び出しインタフェース

COは、CO制御機能から呼び出され、アーギュメントとして diosauca が渡されます。

diosauca には、実行環境情報（ノード名、TPモニタ名、TPBASE実行クラス名等）やトランザクション情報（トランザクションID、リトライ回数、トランザクション処理結果等）が渡されます。

diosadcuca 領域の詳細については、「APIリファレンス」を参照してください。

(2) 電文送受信インタフェース

DIOSA/XTPでは、電文送受信インタフェースとして、以下の2種類のAPIを提供しています。

(a) 電文送信API : diosasendtx

COからの要求により、以下の送信先に電文を送信します。

(i) 論理システム間電文送信

ルーティング機能を使用して、論理システム宛に電文を送信します。電文送受信要求のインタフェース構造体である diosadcuca の宛先指定を論理システム間に、宛先の端末名を指定とすることにより可能となります。

(ii) CO向け電文送信

任意のシステム上のCOを起動し、電文を引き渡します。diosadcuca の宛先指定を派生とすることによりこの要求となります。

CO向け電文送信要求には、以下の種類があります。

・システム内派生 ・TTP間派生

(iii) 連鎖

同一プロセスの同一トランザクションでCOを起動し、電文を引き渡します。diosadcuca の宛先指定を連鎖とすることによりこの要求となります。

(iv) 保留

処理中の電文を一旦保留します。diosadcuca の宛先指定を保留とすることによりこの要求となります。

(b) 電文受信マクロ : diosarecvtx

COからの要求により、電文をCOへ引き渡します。電文の発信元、電文特性、キー情報等の情報は、diosadcuca に返却します。

COは、起動されたら速やかに diosarecvtx により電文を受信しなければなりません。

(3) プログラムの呼び出しインタフェース

COからのプログラム呼び出しに特別な規定はありません。

ただし、DIOSA/XTPの機能で diosavcall (API) を使用すると、アプリケーション動的置換機能が使用できます。

また、プログラム例外発生時のメッセージに関数呼び出しシーケンスが表示される等の機能を使うことができますので、diosavcall の使用をお勧めします。

diosavcall の詳細は、「APIリファレンス」を参照して下さい。

(4) 終了処理インタフェース

COは、CO制御機能へリターンすることにより終了となります。

さらに、終了時に diosauca の「状態コード (S t a t u s) 」の項目、および「利用者コード (U s e r S t a t u s) 」の項目に以下の値をセットすることにより、終了処理に伴うコミット、ロールバック等の処理をCO制御機能に対して要求することができます。

利用者コードはAPが任意に設定できる項目で、設定された値はアボート # 1 出口やアボート # 2 出口、リトライ時のトランザクション初期化出口、CO呼び出し時に引き継がれます。

表 1 終了処理におけるDIOSUCAへのセット項目

状態コード	説 明
DIOSA_ST_DONE	正常終了
DIOSA_ST_ABORT	異常終了要求 (プロセス停止有無は環境定義に依存する)
DIOSA_ST_ABORTCONT	異常終了要求 (プロセスは継続する)
DIOSA_ST_ABORTSTOP	異常終了要求 (プロセスは停止する)
DIOSA_ST_ROLLBACK	ロールバックリトライ要求
DIOSA_ST_DEADLOCK	デッドロックリトライ要求
DIOSA_ST_RLBKCHAIN	ロールバック連鎖要求

3.1.3 通信プログラムの開発

電文送信には `diosasendtx` と呼ばれる A P I を使い、電文受信には `diosarecvtx` と呼ばれる A P I を使います。送受信には `diosadcuca` と呼ばれる領域を介しておこないます。宛先、電文長、送信モード（遅延、強制）等送受信情報はこの `diosadcuca` に設定します。この領域は A P が用意します。

ここでは、D I O S A / X T P 内のノード間で通信するプログラムの作り方について説明します。

C O 制御では、A P ノード⇔O L T P ノード間の電文送信機能を提供します。A P ノード→A P ノード、O L T P ノード→O L T P ノードの通信機能はサポートしていません。

(1) ノード間電文送信必須項目

ノード間通信に必要な情報は、電文、電文長、トランザクション I D と送信モードとなります。

トランザクション I D は宛先ノードの T P B A S E で動作するトランザクション I D 名を指定します。

送信モードは、遅延送信と強制送信を選択することができます。遅延送信はトランザクション正常終了、または `diosacommit` (A P I) 実行と同期を取って送信されます。強制送信は、即時送信となり `diosasendtx` (A P I) の延長で送信が実行されます。

`diosadcuca` 領域の詳細については、「A P I リファレンス」を参照してください。

(2) ノード名指定の電文送信

相手ノードのノード名を指定して電文送信をおこないます。相手ノード名は環境定義 D I O S A M A P に定義されている論理ノード名となります。

相手ノードに複数の T P B A S E がある場合、この T P B A S E はラウンドロビンに選択されます。

(3) ノード名と T P B A S E (T P モニタ) 名指定の電文送信

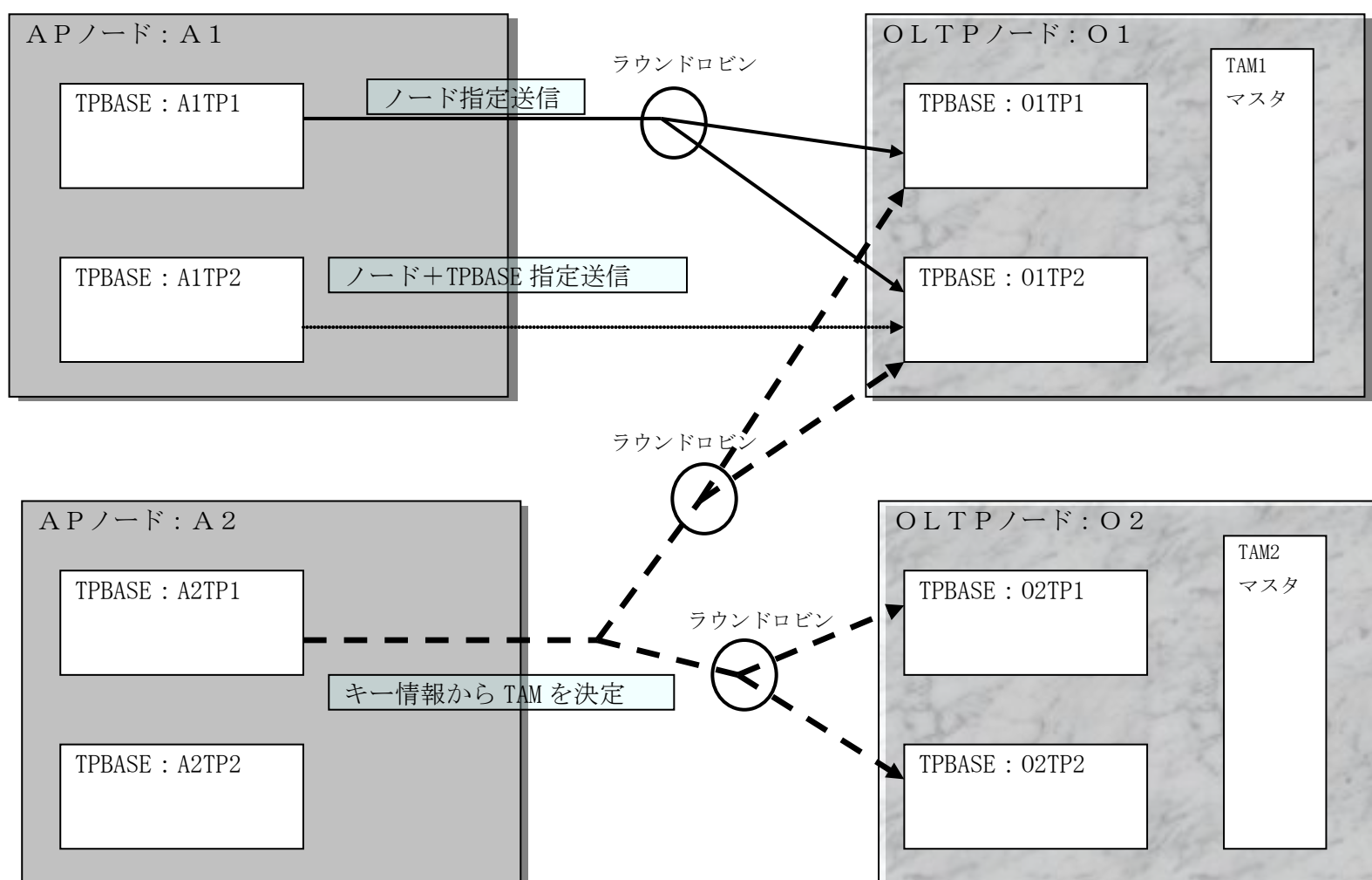
相手ノードのノード名と T P B A S E (T P モニタ) 名を指定して電文送信をおこないます。相手ノード名は環境定義 D I O S A M A P に定義されている論理ノード名となります。T P B A S E 名も D I O S A M A P に定義されている T P B A S E 名となります。

(4) T A M のキー情報指定の電文送信

相手ノードに T A M インスタンスが存在する場合有効 (D I O S A / X T P では A P ノード→O L T P ノード宛送信) となります。

キー情報には、T A M 表のメインキーを指定する方法と M A P I D を指定する方法が選択できます。どちらを選択したのかは A P がフラグを設定することで判断されます。(メインキーが指定されたか否かの判定を C O 制御ができないため)

このキー情報から D I O S A / X T P はどのノードに T A M マスタ表があるのかを判定してノードを決定します。宛先ノードに複数 T P B A S E がある場合は、ラウンドロビンに選択されます。



(5) ノード間電文受信

電文受信には diosarecvtx (API) を使用します。電文送信同様に diosadcuca を使います。diosadcuca には電文が送信されたノード名、TPBASE 名、キー情報等が設定されます。

(6) CO名決定論理

COを決定する方法には以下の3パターンあります。

(a) 電文送信時にCOを決定します。

電文送信時にCO名が決定できる場合は、diosadcuca にCO名を設定します。

(b) 受信電文解析出口でCO名を決定する

受信した電文からCO名を決定する場合、受信電文解析出口からCO名を返却することができます。電文送信時にCOが指定されていた場合、受信電文解析出口にそのCO名を渡しますが、変更したいときは別CO名を返却することができます。本出口のCO名が最優先となります。

(c) 環境定義にCO名を定義する

環境定義では、トランザクションID毎にCO名を定義することができます。

このCO名は、受信電文にCO名が指定されておらず、かつ受信電文解析出口が未定義、または受信電文解析出口からもCO名が返却されなかった場合、本定義のCOを呼び出します。

環境定義のCO名が優先度低となります。

また、環境定義にもCOが定義されずCO名が決定できない場合は、エラーCOを呼び出します。

3.1.4 電文保留

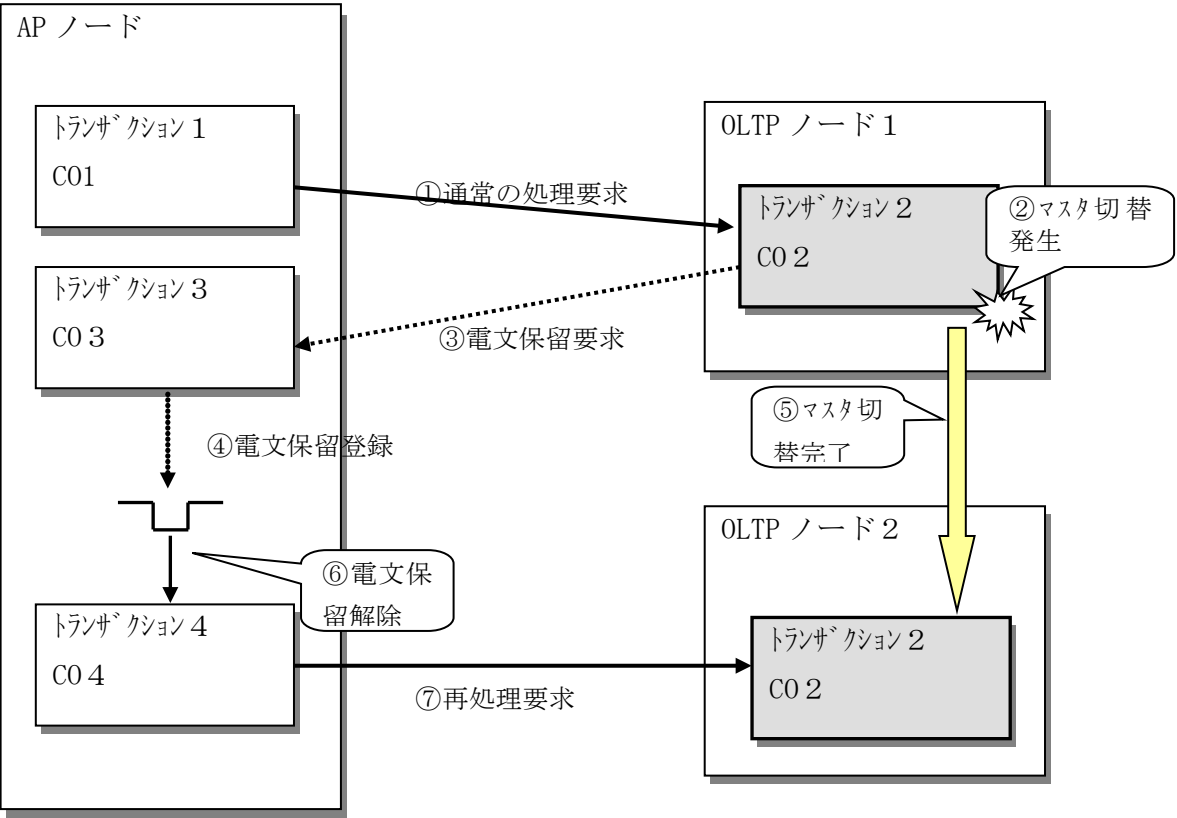
本機能は、業務運用中に電文を一時的に退避しておき、後に退避した電文を一括して再投入をおこなうことを目的としています。デッドロック、ロールバックリトライは即時実行するリトライ機能となり、電文保留機能は電文保留から再実行まで時間を要する場合の遅延リトライ機能となります。

電文保留、再投入するための機能には以下があります。

- ① AP(TPBASE)、OLTP(TPBASE)ノード間で電文送受信がおこなえる API を提供しています。受信 API は diosarecvtx、送信 API は diosasendtx と呼びます。
- ② 保留トランザクションへの登録は、diosasendtx を使っておこないます。diosasendtx へは電文、電文長等と保留トランザクション登録である旨の ID を与え保留電文を登録することができます。
- ③ CO 制御の環境定義(\$COCENV)に保留用トランザクション ID(RESTTXID)を定義することができます。保留用トランザクション ID は TPBASE のクラス毎に定義することが可能ですが、1つの保留用トランザクション ID を複数クラスの保留トランザクションとして使うことも可能です。
- ④ CO の途中、またはコミット時に電文保留要件が発生した場合、保留電文となるトランザクション開始時に受信したオリジナル電文を取得することができます。
- ⑤ 保留用トランザクションの閉塞、閉塞解除、状態照会がおこなえるように、電文保留制御コマンドを提供しています。
- ⑥ 保留が解除された後、実行されるトランザクション ID は保留用トランザクション ID ですが、実際に再処理するトランザクション ID は別(再処理)トランザクション ID となります。CO 制御は保留トランザクション登録時に再処理するトランザクション ID(CO も可)を登録させます。保留用トランザクション ID から再処理トランザクションへの付け替え(トランザクション切り替え)は CO 制御がおこないます。

本利用の手引では、電文保留機能が TAM のマスタ切り替え時に本来捨てられるはずの電文を一旦保留しておき、マスタ切り替え完了後に、切り替え先マスタの存在するノードに再投入する方法を例として説明します。

マスタ切り替え発生はOLTP ノードで検出して、OLTP ノードで受信した電文を一旦 AP ノード上で保留しておき、TAMマスタ切り替え完了後に A P ノードで保留解除して OLTP ノードに再投入する方法の説明となります。



(1) TAM マスタ切り替え発生を知る

TAM マスタ切り替えには、計画的切り替えと障害時切り替えの2つが存在します。

利用者はこの2つの切り替え事象を OLTP ノード上で知ることができます。

マスタ切り替え事象は、トランザクション処理中に割り込んできます。例えば、ある CO が MAPID(1) の TAM 表を更新するために、キー読み込み(diosaimread1)、更新(diosaimrewrite)しようとした場合、キー読み込みが正常終了しても、更新処理がマスタ切り替えに割り込まれると、更新処理はエラーとなってしまいます。(TAM アクセス系処理の延長で発生します)

利用者は、この事象発生を常にチェックすることが求められます。

(a) 利用者が TAM マスタ切り替えを検出

切り替え検出条件は後述します。

(i) AP ノード上の検出

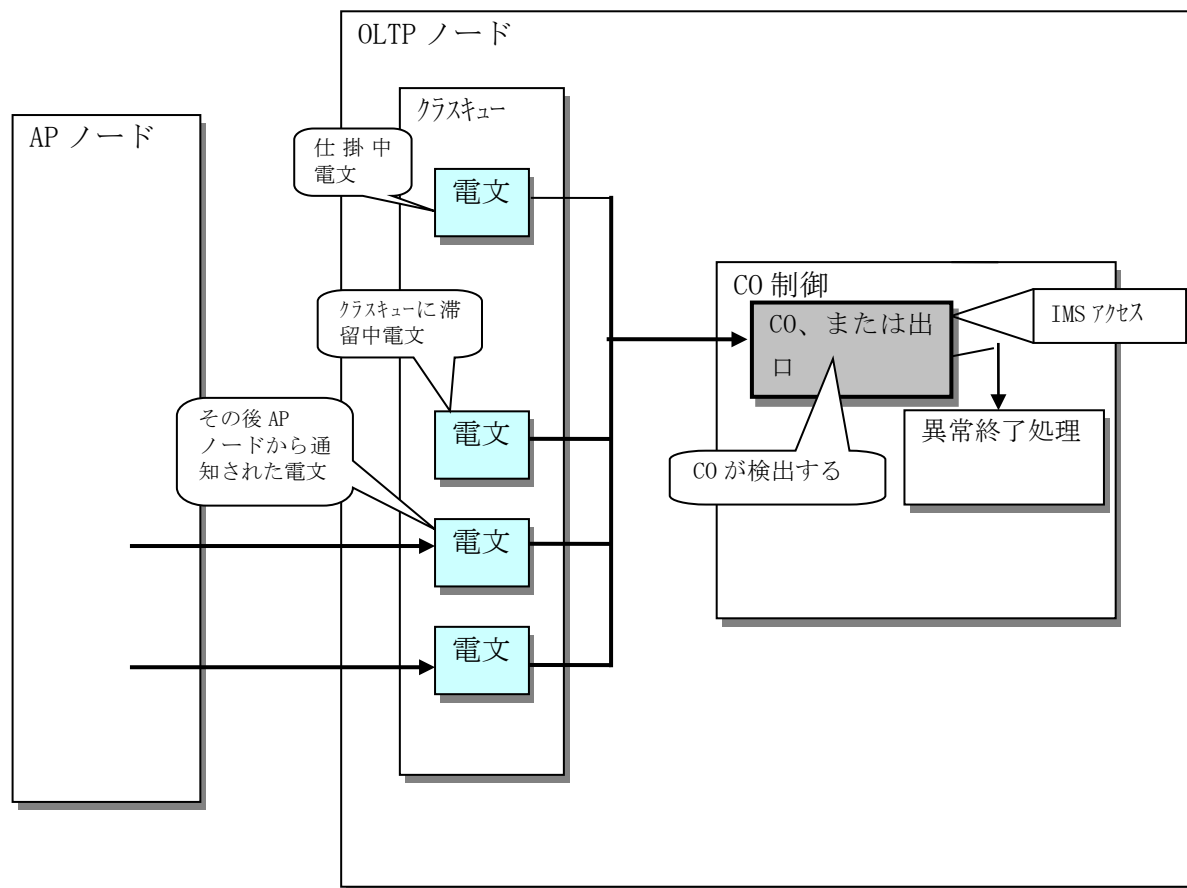
AP ノード上で検出することはできません。

(ii) OLTP ノード上の検出

利用者自身が検出します。

処理中トランザクションの CO、または出口（インメモリサーバへのアクセスが可能な出口 詳細は「DIOSA/XTP API リファレンス 付録 A API 一覧を参照」）で実行された TAM アクセス API の戻り値で知ることができます。詳しくは「DIOSA/XTP API リファレンス 2章 メモリキャッシュ」を参照してください。

TAM のマスタ切り替えが発生すると、以降の TAM アクセスは全て不可となります。利用者は CO、または出口を異常要求（diosauca-Status を DIOSA_ST_ABORT、DIOSA_ST_ABORTCONT、DIOSA_ST_ABORTSTOP のいずれかで終了してください。）また、diosauca-UserStatus には DIOSA_SWITCH、DIOSA_ESWITCH かを一意に識別できる情報を設定して終了してください。diosauca-UserStatus の値はアボート処理の中で参照します。



(2) アボート処理

(a) アボート処理の出口呼び出し

アボート処理とはトランザクションが異常終了を要求した、または CO 制御が継続不可と判断したときに動作します。

アボート処理の出口呼び出しは以下のようになります。



- ① アボート # 1 出口は、A P が異常終了要求を返却した。また、C O 制御が継続不可エラーを検出した場合、ロールバック前に呼び出される出口です。障害情報を採取する程度の処理をおこないます。
- ② ロールバック出口は、ロールバック処理をA P として実装したい場合に用意します。
- ③ アボート # 2 出口は、アボート処理のロールバック後呼び出される出口です。ロールバック後呼び出されますので、インメモリサーバへの更新が可能となります。後述する電文保留要求を強制送信 (diosasendtx) することができます。
- ④ コミット出口は、コミット処理をA P として実装したい場合に用意します。

(b) アボート # 1 出口の TAM マスタ切り替え検出

インメモリサーバへのアクセスはできないので利用者が検出することはできませんが、diosauca の情報からマスタ切り替え中であるか判断することができます。

ExitKey が DIOSA_EX_APREQ の場合、利用者が異常状態を検出したことを意味します。マスタ切り替えの判定は、UserExitKey で判定します。利用者は異常終了要求する前に UserStatus に異常終了理由を設定しておきます。この UserStatus でマスタ切り替え中（計画、障害）を判断できるように設計してください。後述しますが電文保留要求を強制送信 (diosasendtx) することができます。

(c) ロールバック出口の TAM マスタ切り替え検出

アボート # 1 出口同様 diosauca の情報からマスタ切り替え中であるか判断することができます。

マスタ切り替え中はトランザクション内の処理が既にロールバックされている状態ですが、ロールバック処理(diimrollback)とトランザクション開始処理(diimtxstart)は常に呼び出すようにしておいてください。マスタ切り替え中であるからロールバックを呼ばない等の制御は不要です。

後述しますが電文保留要求を強制送信(diosasendtx)することができます。

(d) アボート # 2 出口の TAM マスタ切り替え検出

アボート # 1 出口同様 diosauca の情報からマスタ切り替え中であるか判断することができます。また、インメモリサーバへのアクセスが可能なのでアクセス要求が DIOSA_SWTICH、DIOSA_ESWITCH が返却されます。

マスタ切り替え中は既にロールバック処理は完了している状態なので、インメモリサーバへのアクセスがマスタ切り替え中でも、異常終了要求することは必要ありません。そのまま正常に終了させます。正常終了した場合アボート # 2 の更新を反映するためにコミット出口を呼び出し、そこで異常終了と判定されます。(C0 制御が異常終了と判定する)

出口が異常終了を要求すると、C0 制御はもう一度ロールバック出口を呼び出して空処理後トランザクションを終了します。

(e) コミット出口の TAM マスタ切り替え検出

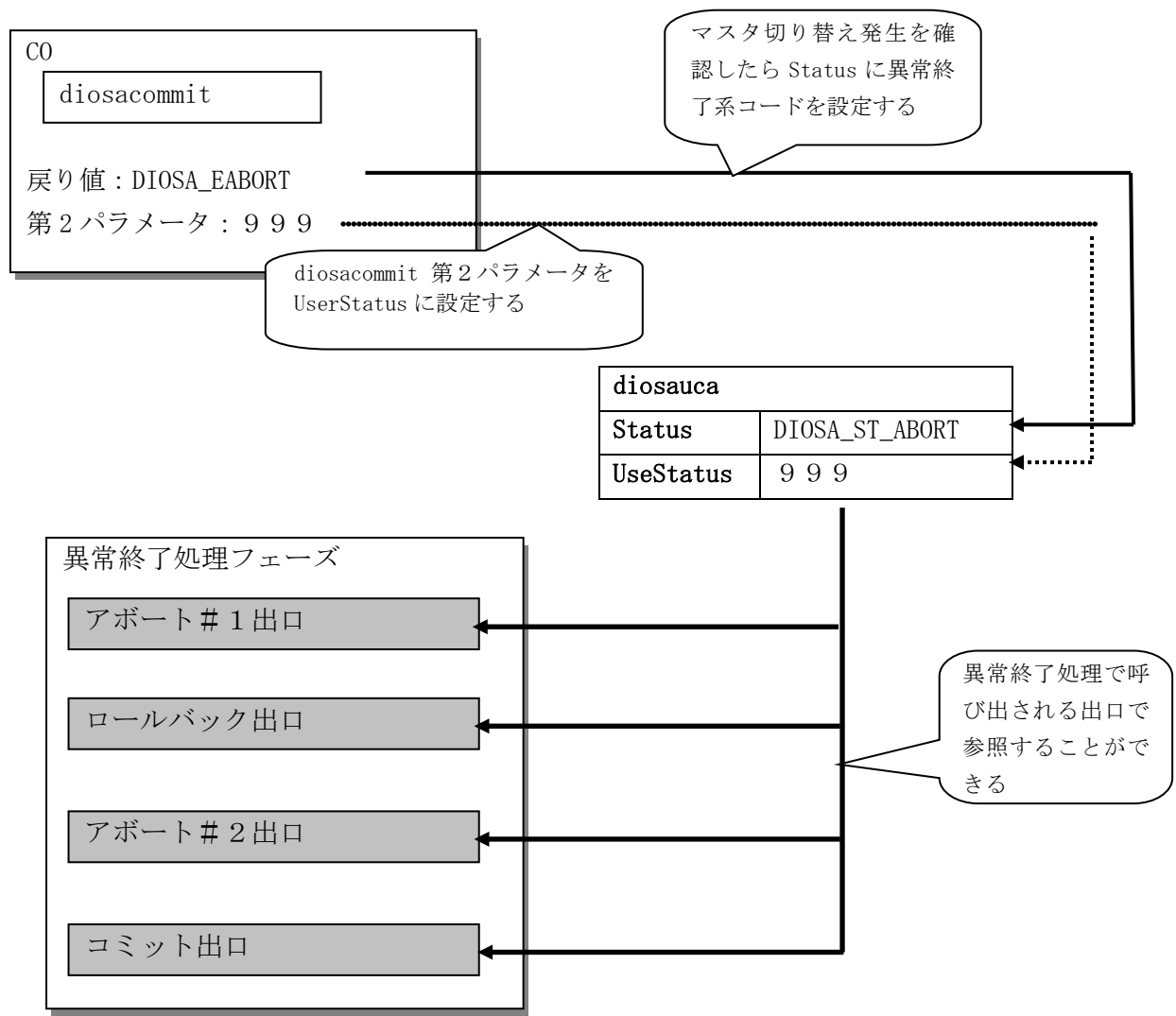
アボート # 1 出口同様 diosauca の情報からマスタ切り替え中であるか判断することができます。また、インメモリサーバへのアクセスが可能なのでアクセス要求が DIOSA_SWTICH、DIOSA_ESWITCH が返却されます。

また、インメモリサーバのコミット処理(diimcommit)を呼び出しても、エラーが発生してコミットはできません。

出口を正常終了要求しても不正な更新処理が残されることはありません。異常終了要求をすると、念のためもう一度ロールバック出口を呼び出して空処理後トランザクションを終了します。

本出口のコミット処理で初めてマスタ切り替えを検出した場合ですが、この電文も保留対象としたい場合は、本出口から後述する電文保留要求を強制送信(diosasendtx)することができます。

- (f) diosacommit(API)コミット出口の TAM マスタ切り替え検出
- コミット出口は、diosacommit(API)からも呼び出されます。
- コミット出口から diosacommit の戻り情報の与え方を記述します。
- C0、出口呼び出しインタフェースで使用する diosauca インタフェースでは、C0 に返却できる情報が限られています。出口終了時の動作指示を指定する Status と利用者が使用できる UserStatus です。
- 出口内で diosacommit の戻り値に DIOSA_SWITCH、DIOSA_ESWITCH が返却された場合、出口終了時の動作指示として異常終了要求のステータス (DIOSA_ST_ABORT、DIOSA_ST_ABORTCONT、DIOSA_ST_ABORTSTOP) を返却します。また、利用者コードにはマスタ切り替えが発生した旨を示す利用者固有のコードを設定してください。
- Status と UserStatus を設定して C0 を終了すると C0 制御は、diosacommit の戻り値に DIOSA_EABORT が返却されます。利用者コードは diosacommit の第 2 パラメータに返却しますので、マスタ切り替え発生有無を確認するようにしてください。



- (g) diosarollback(API)ロールバック出口の TAM マスタ切り替え検出
- ロールバック出口は、diosarollback(API)からも呼び出されます。
- インメモリサーバのロールバック処理でエラーが返却されることはありません。
- diosarollback が終了後、再びインメモリサーバへのアクセスをおこなった場合、DIOSA_SWITCH、DIOSA_ESWITCH が返されます。

(3) **オリジナル受信電文を取得する**

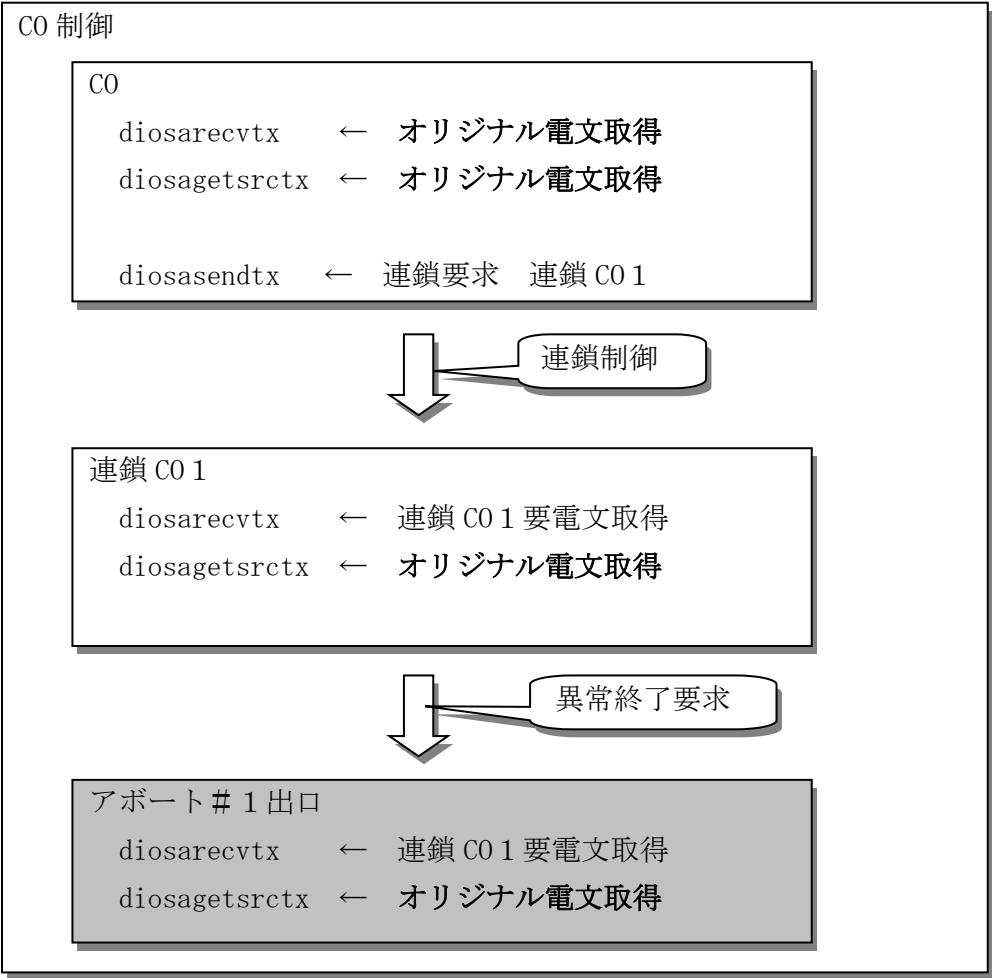
保留する電文の取得と電文情報について記載します。

diosarecvtx（電文受信 API）は最新の電文と電文情報を返却します。C0 が電文を連鎖した場合、最新の電文は連鎖電文となります。

しかし、実際に保留する電文は連鎖電文ではなく、トランザクション初期化時に受信した電文となります。C0 制御はこの電文をオリジナル電文と呼びます。

オリジナル電文の情報は diosagetsrctx（トランザクション開始時電文の取得）という API を使って取得することができます。この API はプロセス初期化、プロセス終了、受信電文解析出口以外の C0、出口で使うことができます。最初に受信した電文なので圧縮されている場合は、圧縮された電文となります。

OLTP ノード上で、保留用電文を AP ノード宛に送信するときは、オリジナル電文を送信するようにします。保留電文を AP ノードに送信する論理をつくるには、まずオリジナル電文を取得します。



(4) 電文保留用トランザクション ID の環境定義

電文保留用トランザクション ID は環境定義項目となります。diosa 環境定義の COCENV 節に定義することができます。

保留用トランザクション ID は TPBASE のクラス毎に定義することが可能ですが、1 つの保留用トランザクション ID を複数クラスの保留用トランザクションとして使うことも可能です。

ただし、クラスは受信できる電文の最大長がありますので、受信電文長が小さいクラスの保留トランザクション ID に電文長が大きなクラスの電文を登録するとトランザクションが電文を受信できなくなります。保留トランザクションを複数クラスで使用する場合は受信電文長が大きなクラスに保留トランザクションを定義してください。またクラス毎に違うトランザクション ID を定義することで保留の分散が可能となります。

(a) (1) クラスの定義

クラスの定義をします。%DEF_CLASS はクラスの既定値を定義します。%CLASS で定義された項のパラメータで指定されなかった定義の既定値となります。%CLASS は存在するクラスの定義をします。%CLASS で定義されなかった MSGMAX(電文最大長)は%DEF_CLASS の MSGMAX が既定値として採用されます。

既定値の定義にはトランザクション ID 定義の%DEF_TRANS もあります。

```
環境定義
$ COCENV
  %DEF_CLASS
    MSGMAX = 32
    RESTTXID = RESTORETXID
  ;
  %CLASS
    NAME = クラス名
    RESTTXID = CORESTTXID001
  ;
;
```

クラスの既定値定義
電文最大長
電文保留用トランザクション ID
電文保留用トランザクション ID

(b) (2) 電文保留用トランザクション ID の定義

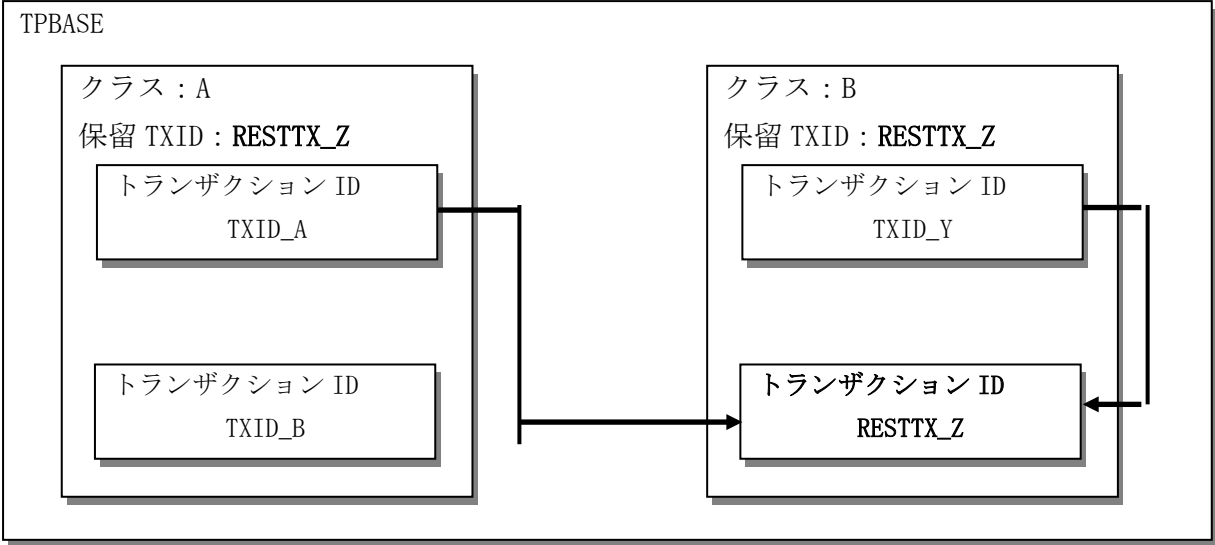
電文保留用トランザクション ID の定義は、%TRANS で定義されます。

電文保留用トランザクション ID とクラスが対応するときは、%CLASS の RESTTXID とクラス下の%TRANS に同じ名前が定義されます。

```
環境定義
$ COCENV
  %CLASS
    NAME = クラス名
    RESTTXID = CORESTTXID001
  ;
  %TRANS
    NAME = CORESTTXID001
    CONAME = CONAME001
  ;
;
```

電文保留用トランザクション ID
トランザクション ID(保留用とする)
CO 名

保留用トランザクションを複数クラス下で共有する場合は、%CLASS の RESTTXID には同一名が現れ、実体を持つ%TRANS は 1 つだけ定義されます。



```
環境定義
$ COCENV
%CLASS
    NAME      =  A
    RESTTXID  =  RESTTX_Z
;
%TRANS
    NAME      =  TXID_A
    CONAME    =  CONAME001
;
%TRANS
    NAME      =  TXID_B
    CONAME    =  CONAME001
;
%CLASS
    NAME      =  B
    RESTTXID  =  RESTTX_Z
;
%TRANS
    NAME      =  TXID_Y
    CONAME    =  CONAME001
;
%TRANS
    NAME      =  RESTTX_Z
    CONAME    =  CONAME001
;
;
```

電文保留用トランザクション ID

トランザクション ID
CO 名

トランザクション ID
CO 名

電文保留用トランザクション ID

トランザクション ID
CO 名

トランザクション ID(保留用とする)
CO 名

(5) **保留実行宛先、トランザクション ID、CO 情報**

利用者は、保留用トランザクションを登録する A P ノード、T P モニタ、電文保留を実行するトランザクション ID、CO 名、保留解除後に実行するトランザクション ID、CO 名、保留解除後に再実行するトランザクション I D、CO 名を決定しなければなりません。

以下の図は、OLTP ノードのトランザクション(A)がマスタ切り替えを検出して、再度 OLTP ノードでトランザクション(A)が実行されるまでを説明します。

OLTP ノードから AP ノードに電文保留登録を依頼する。(図中②)

- (a) 宛先 A P ノード名
- (b) 宛先 T P モニタ名
- (c) 電文保留を実行するトランザクション ID(トランザクション(B))
- (d) 電文保留を実行する CO 名 (C02)

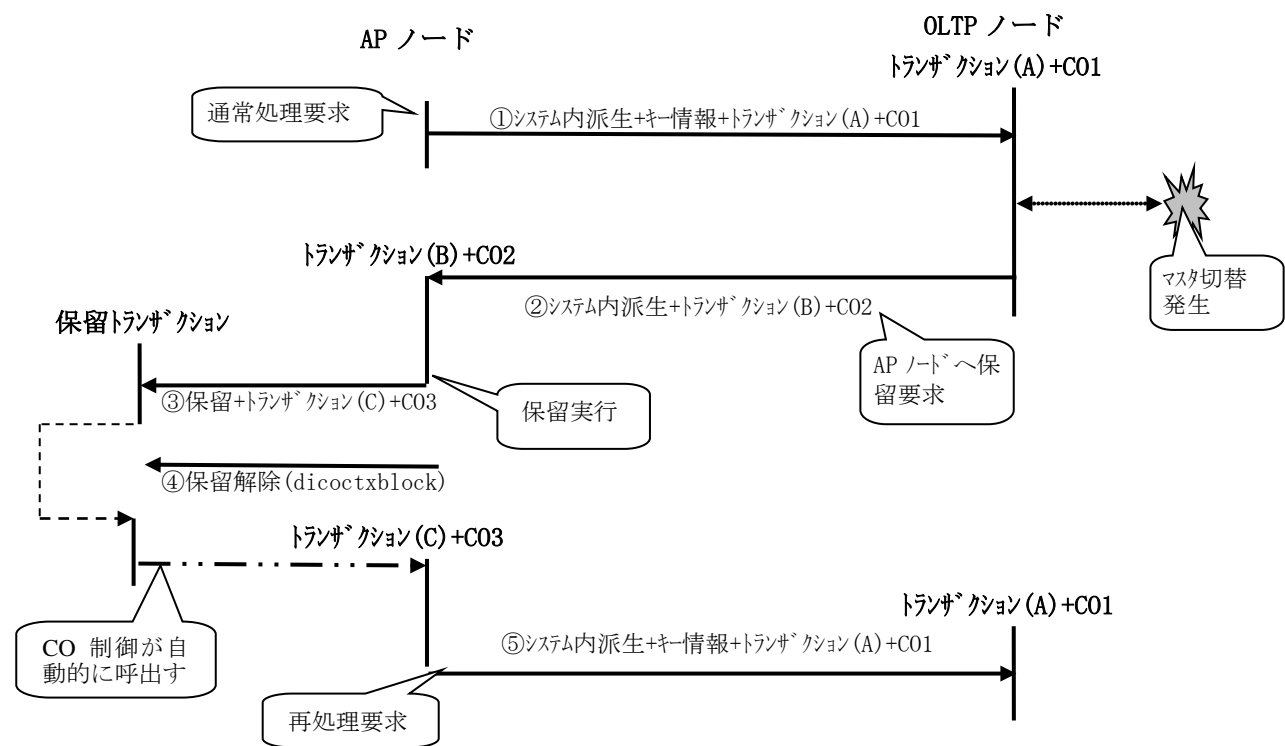
AP ノード上で電文保留登録する。(図中③)

- (e) 電文保留解除後実行するトランザクション ID (トランザクション(C))
- (f) 電文保留解除後実行する CO 名 (C03)

電文保留解除後、再処理を実行する。(図中⑤)

- (g) 宛先 OLTP 情報、または宛先キー情報
- (h) 最初処理した宛先キー情報を持ち回り、再処理に与える
- (i) 再処理するトランザクション I D (トランザクション(A))
- (j) 再処理する CO 名 (C0 1)

dosadcuca の情報と利用者管理情報を使って、保留電文登録をおこないます。

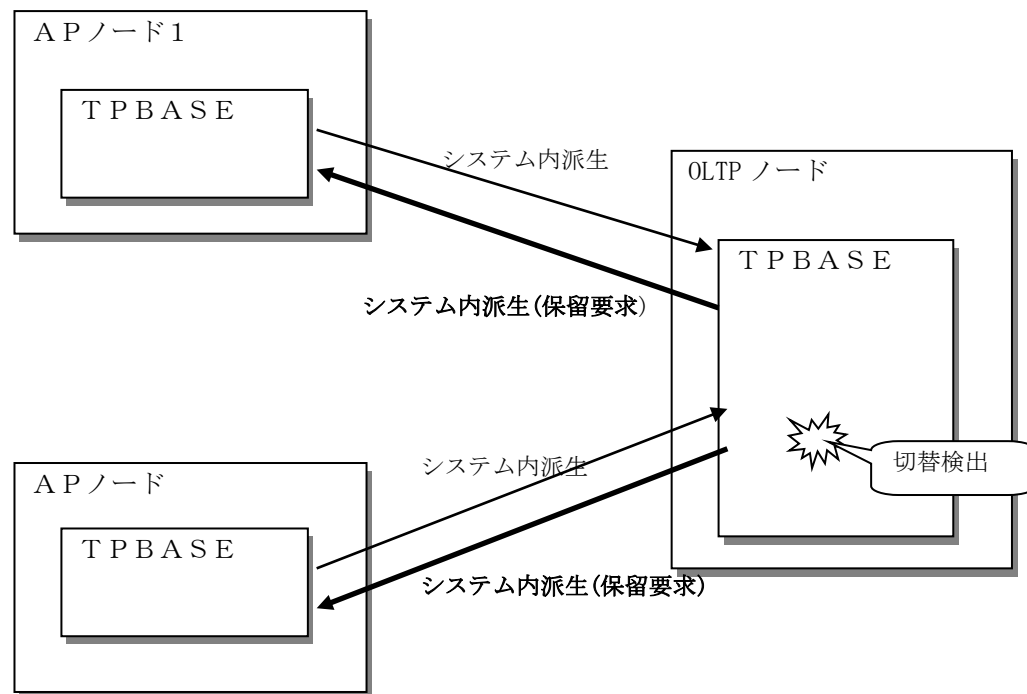


(6) 宛先 AP ノードと宛先 TP モニタ名

(a) 送信元 AP ノードの TP モニタに送信する

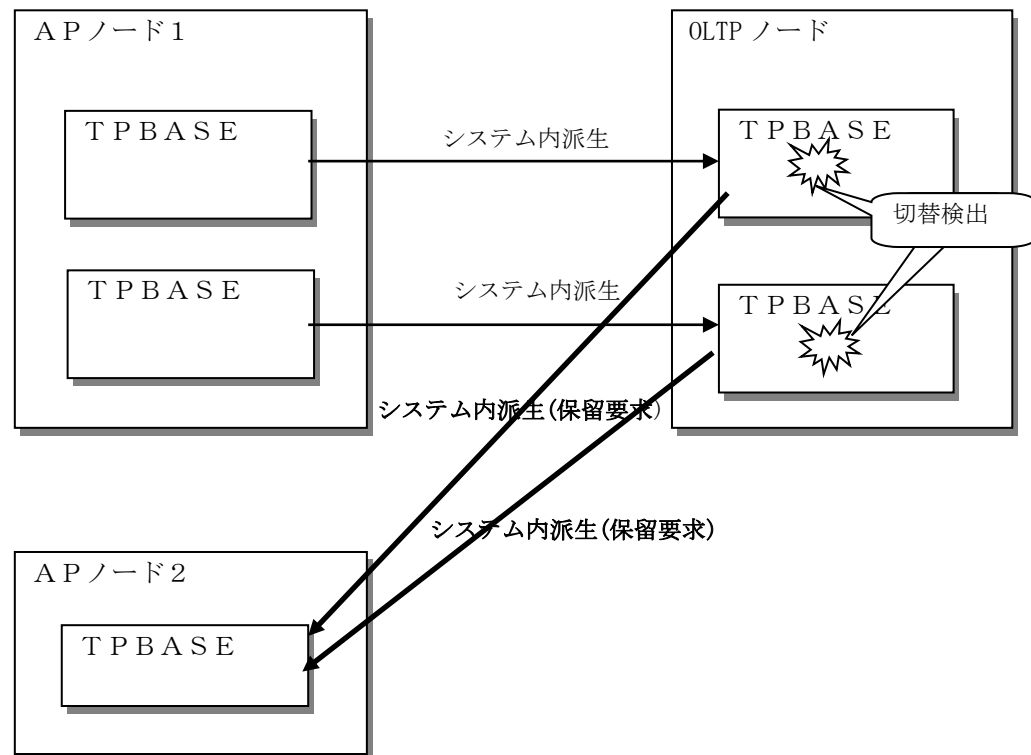
送信元の利用者が diosasendtx で送信元情報としてノード名、TP モニタ名を指定していた場合、送信元のノード名、TP モニタを宛先に指定し直すことで送信元に電文を返却することができます。ノード名のみ指定されていた場合は、ノードは決まりますが、TP モニタは指定することはできません。（TP モニタは diosa がラウンドロビンで自動決定します）

この送信元情報はトランザクション開始時の電文受信の電文情報にあり、diosagetsrctx の diosadcuca で照会することができます。



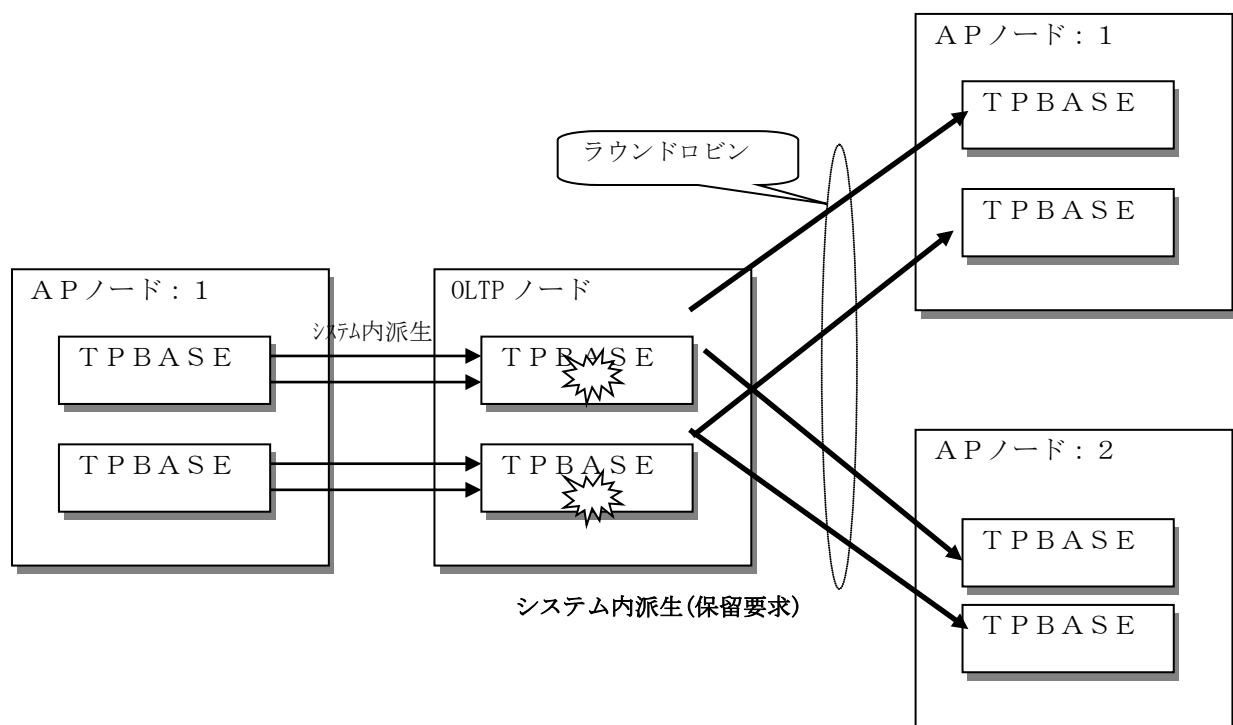
(b) 固定 AP ノードの TP モニタに送信する

保留電文を一箇所に集めたい場合は、事前に AP ノードと TP モニタ名を決めておきます。利用者は固定の AP ノード名、TP モニタ名を diosasendtx に指定します。



(c) 送信先 AP ノードを限定しない

送信先 AP ノードを限定しない場合は、宛先ノード名、TP モニタ名を指定しません。AP ノード、TP モニタ名を diosa が決定（ラウンドロビン）して送信します。保留電文が AP ノード+TP モニタに分散されます。



(7) **宛先キー情報**

AP ノードで再処理要求をおこなうときに diosasendtx の宛先キー情報(diosadcuca- DafInfo)を設定します。このキー情報を与えることで diosa はキーが存在するマスタノードを再選択することができます。

本キー情報は diosa が持ち回ることができます。一連の電文保留処理の際 diosasendtx の送信元情報(diosadcuca- SafInfo)としてキー情報を設定することにより、再処理要求をする C0 でキー情報を送信元情報(diosadcuca- SafInfo)から宛先キー情報(diosadcuca- DafInfo)にコピーすることができます。

(8) 電文保留実行情報の決定

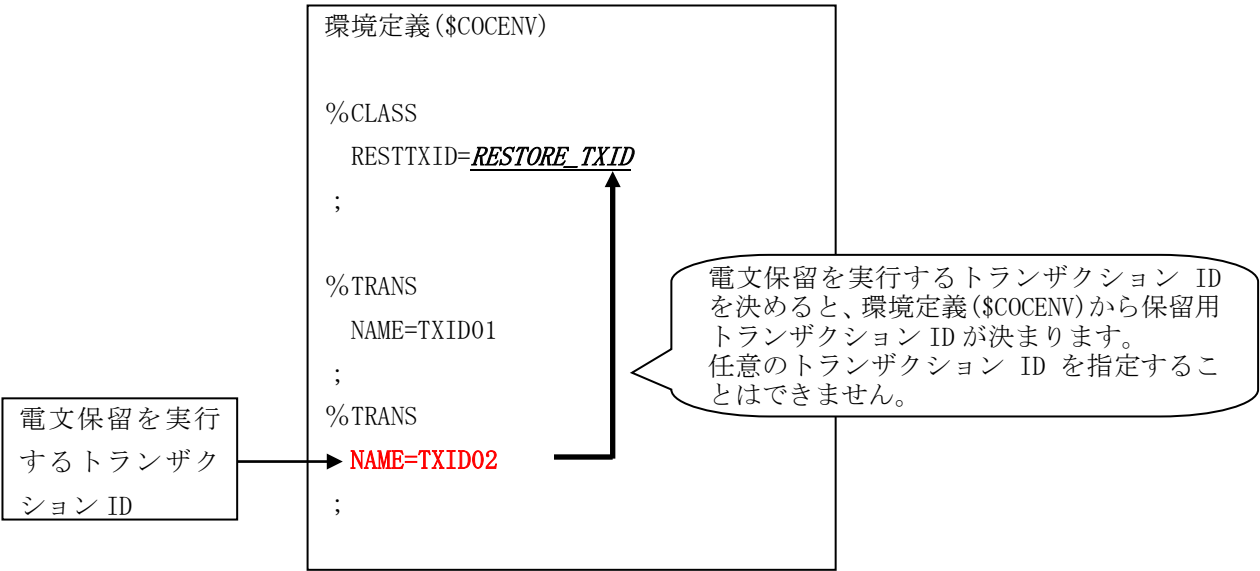
電文保留実行情報とは、電文保留を実行するトランザクション ID と C0 名です。

AP ノードと TP モニタが決定すると、今度は TP モニタのトランザクション ID によりクラスのプロセスが決定します。クラスが決定するとそのクラスに定義された保留電文用トランザクション ID が決定されます。

(%COCENV-%CLASS-RESTTXID) 電文保留を実行するトランザクション ID を決めることは、保留用トランザクション ID も決定することに注意してください。

実行と保留トランザクション ID がきまると電文受信が可能となります。電文を受信すると今度は、電文保留を実行する C0 が呼び出されます。この C0 が実際に保留トランザクションに電文を登録します。

C0 は保留トランザクション登録専用の C0 を作成することをお勧めします。



(9) **利用者管理情報**

利用者管理情報とは、保留解除後に実行するトランザクション ID と CO 名、保留解除後に再実行するトランザクション ID と CO 名があります。

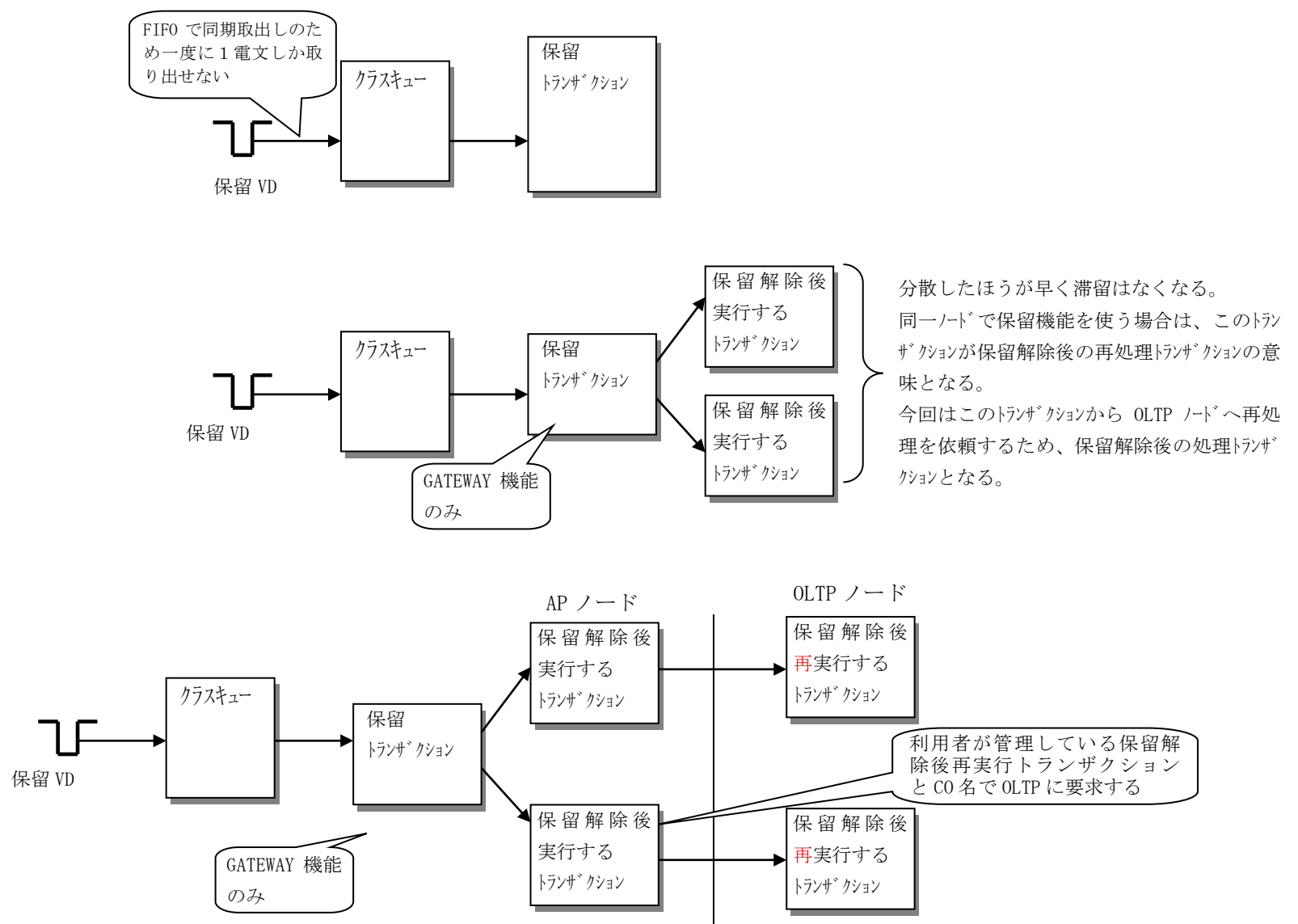
(a) 保留解除後に実行するトランザクション ID と CO 名

保留を解除したときにトランザクション型 VD 経由の電文は VD から FIFO で取り出されるため、大量に滞留していると滞留がなくなるまで時間がかかります。そのため CO 制御は保留トランザクション上で利用者処理をおこなわせません。保留トランザクションの閉塞が解除した後に処理を委託するトランザクション ID と CO 名を登録してもらいそのトランザクション ID 上で処理をおこないます。本来保留要求があったノードで保留処理をおこなう機能もあります。

(b) 保留解除後に再実行するトランザクション ID と CO 名

今回の処理の流れでは、保留解除後に再び OLTP ノードに送信しなければなりません。CO 制御は、保留解除後に再実行するトランザクション ID と CO 名に対する電文送信 (diosasendtx) のみをおこないます。

保留解除後に再実行するトランザクション ID と CO 名とは、OLTP ノードでマスタ切り替えに遭遇したトランザクション ID と CO 名ですが、別トランザクション ID、別 CO で実行することも可能です。利用者側で決定してください。



(10) **保留対象電文を AP ノードに送信する**

保留対象電文を AP ノードに送信するには、diosasendtx を使用します。

t_diosa_dcuca の設定項目を以下に記載します。

本説明では必須項目としていますが、指定しなくてもよい項目もあります。「API リファレンス」を参照してください。（AP→OLTP の SafInfo は、送信元に保留要求を返せるように必須項目としています）

AP→OLTP、OLTP→AP は、通常の業務処理で使います。

保留要求は OLTP→AP を使います。

保留電文登録は、AP ノード上で保留登録時使います。

t_diosa_dcuca		AP→OLTP	OLTP→AP	保留電文登録
TextChar		DIOSA_MSG_DERIV	DIOSA_MSG_DERIV	DIOSA_MSG_REST
SafInfo	LsName	○	—	—
	LNodeName	○	—	—
	TpMonitor	○	—	—
	KeyInfo	—	—	—
DafInfo	LsName	○	—	—
	LNodeName	○	★	—
	TpMonitor	○	★	—
	KeyInfo	★	—	—
TxId		○	○ 注 1	○ 注 4
CoName		○	○ 注 2	○ 注 5
MsgSize		○	○	○
SendMode		○	○ 注 3	○

○ ：必須項目

★ ：選択指定

- 注 1 : 電文保留を実行するトランザクション ID
- 注 2 : 電文保留を実行する CO 名
- 注 3 : DIOSA_SEND_FORCE（強制送信）を指定すること
- 注 4 : 保留解除後に再実行するトランザクション ID
- 注 5 : 保留解除後に再実行する CO

(11) **保留電文を登録する**

保留対象電文を受信し、電文保留を実行する CO が起動されます。

ここで、保留電文の登録を行います。diosasendtx、t_diosa_dcuca の設定項目は上を参照してください。

(12) 電文保留制御

電文保留は TPBASE 機能を使い保留トランザクションを閉塞することで、実行を保留します。また、保留トランザクションの閉塞を解除すると保留していた電文の処理が実行されます。C0 制御は、閉塞、閉塞解除、状態照会するコマンドを提供しています。

なお、環境定義（IMENV 節）で閉塞、解除を自動でおこなう機能を定義することができます。（詳細は「DIOSA/XTP 環境定義マニュアル」を参照してください）

dicotxblock コマンドを使用します。

(a) 保留トランザクションの閉塞

通常、保留トランザクションは閉塞状態にしておきます。dicotxblock コマンドを全 AP ノードで実行してください。運用的には閉塞状態にすることでマスタ切り替えを意識することなく保留トランザクションに電文を貯めることができます。

(b) 保留トランザクションの閉塞解除

マスタ切り替えが終了すると、保留トランザクション（厳密には保留トランザクションのトランザクション型 VD へ電文が滞留する）の閉塞を解除します。全 AP ノードで実行してください。

解除すると滞留していた電文が処理されます。閉塞解除も dicotxblock コマンドが使えます。

(c) 保留トランザクションの状態照会

保留トランザクションと対応する VD 状態を照会します。

3.1.5 C O制御利用者出口の開発

利用者出口は、アプリケーションが動作する前後に呼び出される特別なプログラムです。

出口は、以下のものがあります。

全ての出口はC O呼び出し時に渡される `diosauca` が同様に渡されます。受信電文解析出口のみ特別な情報交換領域が追加されます。

- ・プロセス初期化出口
- ・電文解析出口
- ・トランザクション初期化出口
- ・トランザクション終了出口
- ・アボート# 1 出口
- ・アボート# 2 出口
- ・コミット出口
- ・ロールバック出口
- ・プロセス終了出口

(1) プロセス初期化出口

C O制御T P Pプロセス開始時に一度だけ呼び出されます。

A Pが用意する共有メモリのアタッチ処理や、プロセス内共有メモリ確保等をおこなうことができます。

O r a c l e 等D Bを使う場合初期接続等もできます。

(2) 電文解析出口

トランザクション開始時、最初に呼び出されます。電文の事前解析、圧縮電文を解凍、C O名を決定することができます。

本出口インタフェースのみ `t_diosa_analyze` の型をもつパラメータが追加されます。本パラメータに圧縮解凍後の電文、電文長、C O名等を返却することができます。

(3) トランザクション初期化出口

トランザクション初期化の最後に呼び出されます。

トランザクション開始前準備をおこないます。

(4) トランザクション終了出口

トランザクションを正常終了する直前で呼び出されます。

コミット前の共通的業務処理等をおこないます。

(5) アボート# 1 出口

A Pがアボート要求を返却し、また、C O制御が継続不可エラーを検出した場合、ロールバック前に呼び出される出口です。

障害情報を採取する程度の処理をおこないます。

また、処理結果を送信したい場合は `diosasendtx` の送信モードに強制送信を指定すると強制的に電文送信

をおこなうことができます。

(6) **アボート#2 出口**

アボート処理のロールバック後呼び出される出口です。

ロールバック後呼び出されますので、インメモリサーバ、DBへの更新が可能となります。

また、処理結果を送信したい場合は diosasendtx の送信モード、遅延、強制とも電文送信が可能となります。遅延を指定した場合は、本出口終了後のコミット処理で電文送信が完了します。

(7) **コミット出口**

コミット処理をAPとして実装したい場合に用意します。

本出口は、トランザクション終了時(トランザクション終了出口呼び出し後)、diosacommit (API) の延長、アボート処理終了時(アボート#2 出口呼び出し後)に呼び出されます。

本出口がエラーを返却した場合、トランザクション終了はアボート処理に遷移し、アボート処理ではロールバックしてトランザクションを異常終了します。diosacommit 内のコミット出口がエラーを返却した場合は、APの責任で実行中のCO、または出口を異常終了要求で終了してください。

Oracle 等DBのコミットもここで実行可能です。

(8) **ロールバック出口**

ロールバック処理をAPとして実装したい場合に用意します。

本出口は、アボート処理(アボート#1 出口呼び出し後、アボート#2 出口呼び出し前)、diosarollback (API) の延長、リトライ処理、ロールバック連鎖で呼び出されます。

(9) **プロセス初期化出口**

CO制御TPPプロセス終了時に一度だけ呼び出されます。

プロセス終了のための資源解放等をおこないます。

3.1.6 メモリ管理機能とのインタフェース

メモリ管理機能における、アプリケーションプログラムの作成方法について説明します。

(1) メモリ割り当て／再割り当て／解放

領域種別が更新可共有メモリの割り当て、及び一括解放対象サービス内メモリのメモリ割り当て／メモリ再割り当て／メモリ解放するプログラム例を以下に記します。

利用例

```
#include <diosa.h>

int sample1()
{
    int      Ret = 0;
    char*    ShmArea = NULL;
    char*    PrcArea = NULL;

    /******
    /* メモリ割り当て(共有メモリ)
    /* メモリ識別子 : MEMID01
    /* エントリキー : ENTRY01
    /* 領域種別      : 更新可共有メモリ
    /* メモリサイズ : 512
    /******
    Ret = diosamalloc( "MEMID01", "ENTRY01", DIOSA_APNOPROT, 512, (void **)&ShmArea );
    if( DIOSA_DONE != Ret ){
        /* エラー処理 */
    }

    /******
    /* メモリ割り当て(プロセスメモリ)
    /* メモリ識別子 : MEMID02
    /* エントリキー : ENTRY02
    /* 領域種別      : 一括解放対象サービス内メモリ
    /* メモリサイズ : 1024
    /******
    Ret = diosamalloc( "MEMID02", "ENTRY02", DIOSA_SVCALL, 1024, (void **)&PrcArea );
    if( DIOSA_DONE != Ret ){
        /* エラー処理 */
    }

    if( /* メモリ領域不足 */ ){
        /******
        /* メモリ再割り当て(プロセスメモリ)
        /* メモリ識別子 : MEMID02
        /* エントリキー : ENTRY02
        /* 領域種別      : 一括解放対象サービス内メモリ
        /* メモリサイズ : 1024 -> 2048
        /******
        Ret = diosarealloc( "MEMID02", "ENTRY02", DIOSA_SVCALL, 2048, (void **)&PrcArea );
        if( DIOSA_DONE != Ret ){
            /* エラー処理 */
        }
    }

    /******
    /* メモリ解放(プロセスメモリ)
    /* メモリ識別子 : TEST02
    /* 領域種別      : 一括解放対象サービス内メモリ
    /* 解放領域      : NULL
    /******
    Ret = diosamfree( "MEMID02", DIOSA_SVCALL, NULL );
    if( DIOSA_DONE != Ret ){
        /* エラー処理 */
    }

    return 0;
}
```

diosarealloc 関数を利用する場合、diosamalloc 関数に指定したメモリ識別子(MEMID02)を diosarealloc 関数に指定する必要があります。

diosamfree 関数を利用する場合、メモリ識別子指定とアドレス指定の 2 つの方法があります。上記の例では、メモリ識別子指定(MEMID02)でメモリ解放しています。

C0 制御サーバ上で動作するアプリケーションで領域種別が一括解放対象サービス内メモリの場合、メモリ解放を行わなくても、トランザクション初期化で一括解放されます。

(2) **メモリアドレス取得**

前述(1)でメモリ割り当てした更新可共有メモリをメモリ識別子(MEMID01)指定でメモリアドレスを取得するプログラム例を以下に記します。

利用例

```
#include <diosa.h>

int sample2()
{
    int      Ret = 0;
    char*    ShmArea = NULL;

    /* *****/
    /* メモリアドレス取得(共有メモリ)          */
    /* メモリ識別子設定: MEMID01                */
    /* 領域種別設定   : 更新可共有メモリ        */
    /* *****/
    Ret = diosamaddr( "MEMID01", DIOSA_APNOPROT, (void **)&ShmArea );
    if( DIOSA_DONE != Ret ){
        /* エラー処理 */
    }

    return 0;
}
```

上記プログラム(sample2)を前述(1)でメモリ割り当てした C0 制御サーバとは別の C0 制御サーバ上で実装することで、プロセス間で共有メモリのアドレスを取得できます。

3.1.7 ロック制御機能とのインタフェース

ロック制御機能における、アプリケーションプログラムの作成方法について説明します。

(1) ファイル型ロック取得／ロック解放

ファイル型ロックの取得／解放するプログラム例を以下に記します。

利用例

```
#include <diosa.h>

int sample1()
{
    int      Ret = 0;
    /******
    /* ファイル型ロック取得
    /******
    Ret = diosalock( 1, DIOSA_INNODE );
    if( DIOSA_DONE != Ret ){
        /* エラー処理 */
    }

    /******
    /* ファイル型ロック解放
    /******
    Ret = diosaunlock( 1, DIOSA_INNODE );
    if( DIOSA_DONE != Ret ){
        /* エラー処理 */
    }
    return 0;
}
```

CO 制御 TPP 上で動作するアプリケーションの場合、ロック解放を行わなくても、トランザクション終了処理で、取得しているファイル型ロックが強制的に解放されます。

(2) DB 型ロック取得／ロック解放

DB 型ロックの取得／解放するプログラム例を以下に記します。

利用例

```
#include <diosa.h>

int sample2()
{
    int      Ret = 0;
    /******
    /* DB 型ロック取得
    /******
    Ret = diosalock( 1, DIOSA_INSYSTEM );
    if( DIOSA_DONE != Ret ){
        /* エラー処理 */
    }

    /******
    /* DB 型ロック解放
    /******
    Ret = diosaunlock( 1, DIOSA_INSYSTEM );
    if( DIOSA_DONE != Ret ){
        /* エラー処理 */
    }
    return 0;
}
```

CO 制御 TPP 上で動作するアプリケーションの場合、ロック解放を行わなくても、トランザクション終了処理で、取得している DB 型ロックが強制的に解放されます。

3.1.8 アプリケーショントレース機能とのインタフェース

アプリケーショントレース機能における、アプリケーションプログラムの作成方法について説明します。

(1) トレース情報ファイル直接出力

トレース情報をファイル直接出力するプログラム例を以下に記します。

利用例

```
#include <diosa.h>

int sample1(int Param1)
{
    int      Ret = 0;
    /***/
    /* トレース情報ファイル直接出力 */
    /***/
    Ret = diosaapptrcf( "Param1", &Param1, sizeof(Param1), 0, 9 );
    if( DIOSA_DONE != Ret && DIOSA_IGNORE != Ret ){
        /* エラー処理 */
    }
    return 0;
}
```

この例では、引数 Param1 に指定したデータをトレース情報としてファイルに直接出力します。

[参考]

第4引数にはエラーコードを指定します。第4引数は、トレース情報と合わせて数値情報（例えば errno 等）を出力したい場合に使用します。

第5引数には出力レベルを指定します。トレース情報の重要度にあわせて出力レベルを指定してください。APIの詳細については、「API リファレンス」を参照してください。

(2) トレース情報メモリ経由出力

トレース情報をメモリ経由で出力するプログラム例を以下に記します。

利用例

```
#include <diosa.h>

int sample2(int Param1)
{
    int      Ret = 0;
    /***/
    /* トレース情報メモリ経由出力 */
    /***/
    Ret = diosaapptrcm( "Param1", &Param1, sizeof(Param1), 0, 9 );
    if( DIOSA_DONE != Ret && DIOSA_IGNORE != Ret ){
        /* エラー処理 */
    }
    return 0;
}
```

この例では、引数 Param1 に指定したデータをトレース情報として共有メモリ上のトレース情報保存領域に格納します。（トレース情報保存領域に格納したトレース情報はトレース情報出力デーモンがファイルに出力します。）

[参考]

第 4 引数にはエラーコードを指定します。第 4 引数は、トレース情報と合わせて数値情報（例えば `errno` 等）を出力したい場合に使用します。

第 5 引数には出力レベルを指定します。トレース情報の重要度にあわせて出力レベルを指定してください。API の詳細については、「API リファレンス」を参照してください。

3.1.9 アプリケーション動的置換機能とのインタフェース

アプリケーション動的置換機能を利用した、アプリケーションプログラムの作成方法について説明します。

(1) 関数呼び出し

アプリケーション動的置換機能を使用して、ユーザ作成の関数を呼び出す方法を以下に示します。

プログラム例

```
#include <diosa.h>

int sample1(void)
{
    int    Ret;
    long   FuncRet;
    char   *Param[2];

    /******
    /* 関数呼び出し */
    /******
    Param[0] = 10;
    Param[1] = "parameter";
    FuncRet = 0;

    Ret = diosavcall( "sample2", &FuncRet, 2, Param );

    if ( DIOSA_DONE != Ret || 0 != FuncRet ){
        /* エラー処理 */
        return -1;
    }
    return 0;
}
```

この例では、以下のインタフェースの関数を呼び出しています。（第一パラメータに 10、第二パラメータに "parameter" を渡しています。）

```
long sample2 ( int param1, char *param2 );
```

API の詳細については、「API リファレンス」を参照してください。

3. 1. 10 コマンド配信機能とのインタフェース

コマンド配信機能における、アプリケーションプログラムの作成方法について説明します。

(1) コマンド配信

diosacmdsend 関数によりコマンド配信要求をする際は、コマンド配信 UCA 領域に配信先情報と配信結果の取得方法を設定し、配信コマンドを指定して関数を呼び出します。

(a) 同期確認型コマンド配信

配信結果確認方法に、DIOSA_CMDSND_RESULT_WAIT を指定した場合、diosacmdsend 関数内で配信結果を待ち合わせ、配信先ノード数分の配信結果を返却することができます。

利用例

```
#include <diosa.h>

int sample()
{
    int          Ret;
    char          CmdText[1023+1];
    t_diosa_cmdsenduca SendUca;
    t_diosa_cmdconfuca* ConfUca = NULL;

    /* 初期化 */
    memset( CmdText, '¥0', sizeof( CmdText ) );
    memset( &SendUca, '¥0', sizeof( SendUca ) );

    /* 配信先情報の設定：LS01 配下の全ノードに配信
     *      配信エラー時は 10 秒間隔で 3 回リトライ */
    strcpy( SendUca.SendInfo.DstName, "LS01" );
    SendUca.SendInfo.LsType          = DIOSA_CMDSND_DST_LSALL;
    SendUca.SendInfo.Target          = DIOSA_CMDSND_TARGET_ALL;
    SendUca.SendInfo.SelfFlg         = DIOSA_ON;
    SendUca.SendInfo.ForceFlg        = DIOSA_OFF | DIOSA_CMDSND_BLKEXCL_OFF;
    SendUca.SendInfo.ExecTimeOut     = 60;
    SendUca.SendInfo.ApiTimeOut      = 120;
    SendUca.SendInfo.RtryCnt          = 3;
    SendUca.SendInfo.RtryIntvl       = 10;

    /* 配信結果確認方法の設定：配信結果を待ち合わせ、実行結果ファイルを取得する */
    SendUca.ResultInfo.ResultMode    = DIOSA_CMDSND_RESULT_WAIT;
    SendUca.ResultInfo.ResultFileFlg = DIOSA_ON;

    /* 配信コマンドの設定 */
    strcpy( CmdText, "dixxxxxx" );
    SendUca.CmdTextLen = strlen(CmdText);
    SendUca.CmdText    = CmdText;

    /* コマンド配信の実行 */
    Ret = diosacmdsend( &SendUca, &ConfUca, NULL );

    if( DIOSA_DONE != Ret && DIOSA_ALMOST != Ret ){
        /* コマンド配信エラー処理 */

        return -1;
    }

    /* コマンド配信結果の確認処理 */
    for( int i=0; ConfUca->NodeCnt > i; i++ ){
        if( DIOSA_DONE != ConfUca->NodeConf[i].SendStatus ){
            /* コマンド配信結果ステータスエラー処理 */
        }
    }
}
```

```

        continue;
    }

    /* コマンド配信結果ステータス正常処理 */

}

return 0;
}

```

この例では、論理システム“LS01”配下の全ノードへ、コマンド“dixxxxxx”を配信します。

配信結果は関数内で待ち合わせを行うことにより取得し、ConfUca に配信先ノード数分の配信結果が返却されます。実行結果ファイル（stdout、stderr）がある場合は、ノード単位コマンド実行結果返却領域の StdoutFilePath および StderrFilePath 項目に出力ファイルパスが設定されます。

[参考]

上記例と同様のコマンド配信を dicmdsend コマンドで実行する場合、下記のようにコマンドを実行します。

```
dicmdsend -d LS01 -p -x all -s yes -w 60 -v 120 -c 3 -i 10 -t "dixxxxxx"
```

(b) 非同期確認型コマンド配信

配信結果確認方法に、DIOSA_CMDSND_RESULT_CONF を指定した場合、diosacmdsend 関数内では配信結果の待ち合わせは行わず、diosacmdconf 関数により結果を取得することができます。

利用例

```

#include <diosa.h>

int sample()
{
    int          Ret;
    int          Socket;
    char         CmdText[1023+1];
    t_diosa_cmdsenduca SendUca;
    t_diosa_cmdconfuca* ConfUca = NULL;

    /* 初期化 */
    Socket = 0;
    memset( CmdText, '¥0', sizeof( CmdText ) );
    memset( &SendUca, '¥0', sizeof( SendUca ) );

    /* 配信先情報の設定：LS01 配下の全 OLTP ノードに配信 */
    strcpy( SendUca.SendInfo.DstName, "LS01" );
    SendUca.SendInfo.LsType      = DIOSA_CMDSND_DST_LSOLTP;
    SendUca.SendInfo.Target      = DIOSA_CMDSND_TARGET_ALL;
    SendUca.SendInfo.SelfFlg     = DIOSA_ON;
    SendUca.SendInfo.ForceFlg    = DIOSA_OFF | DIOSA_CMDSND_BLKEXCL_OFF;
    SendUca.SendInfo.ExecTimeOut = 60;
    SendUca.SendInfo.ApiTimeOut  = DIOSA_CMDSND_DEFAULT;
    SendUca.SendInfo.RtryCnt     = DIOSA_CMDSND_DEFAULT;
    SendUca.SendInfo.RtryIntvl   = DIOSA_CMDSND_DEFAULT;

    /* 配信結果確認方法の設定：配信結果を別関数で受け取る */
    SendUca.ResultInfo.ResultMode = DIOSA_CMDSND_RESULT_CONF;
    SendUca.ResultInfo.ResultFileFlg = DIOSA_ON;

    /* 配信コマンドの設定 */
    strcpy( CmdText, "dixxxxxx" );
    SendUca.CmdTextLen = strlen(CmdText);
    SendUca.CmdText    = CmdText;
}

```



```

/* コマンド配信の実行 */
Ret = diosacmdsend( &SendUca, NULL, &Socket );

if( DIOSA_DONE != Ret ){

    /* コマンド配信エラー処理 */

    return -1;
}

/* ~~~ */

/* コマンド配信結果取得 */
Ret = diosacmdconf( Socket, 120, &ConfUca );

if( DIOSA_DONE != Ret && DIOSA_ALMOST != Ret ){

    /* コマンド配信結果取得エラー処理 */

    return -1;
}

/* コマンド配信結果の確認処理 */
for( int i=0; ConfUca->NodeCnt > i; i++ ){

    if( DIOSA_DONE != ConfUca->NodeConf[i].SendStatus ){

        /* コマンド配信結果ステータスエラー処理 */

        continue;
    }

    /* コマンド配信結果ステータス正常処理 */

}

return 0;
}

```

この例では、論理システム“LS01”配下の全 OLTP ノードへ、コマンド“dixxxxxx”を配信します。

配信結果は diosacmdconf 関数内で待ち合わせを行うことにより取得し、ConfUca に配信先ノード数分の配信結果が返却されます。diosacmdconf 関数には、diosacmdsend 関数で返却されたソケット識別子を引数として渡す必要があります。

[参考]

上記例と同様のコマンド配信を dicmdsend コマンドで実行する場合、下記のようにコマンドを実行します。

```
dicmdsend -d LS01 -p oltp -x all -s yes -w 60 -v 120 -t "dixxxxxx"
```

3.1.11 タイマ制御機能とのインタフェース

タイマ制御機能における、アプリケーションプログラムの作成方法について説明します。

(1) C0 タイマ登録

C0 タイマを登録するには diosatmccoset 関数を利用します。diosatmccoset 関数を利用する際には、タイマの制御情報(t_diosa_tmcuca)、C0 制御情報(t_diosa_cosetuca)、送信メッセージを設定した上で関数を呼び出す必要があります。

※C0 制御やバッチ AP 制御を利用せずにタイマ登録アプリを作成する場合には、diosatmccoset 関数に加えて diosatmccommit 関数を組み合わせて実行する必要があります。

(a) インターバル指定

時刻形式に DIOSA_INTER、DIOSA_IMMEDIATE を設定した場合、一定時間間隔ごとに実行されるタイマを登録することができます。DIOSA_IMMEDIATE の場合、登録直後に初回タイマが実行されます。

利用例

```
#include <diosa.h>

int SAMPLESET() {

    int          Ret;
    int          RetCnt;
    int          RetRbk;
    t_diosa_tmcuca TmcUca;
    t_diosa_cosetuca CosetUca;
    char          Data[200];

    /* 初期化 */
    memset( &TmcUca, 0x00, sizeof( t_diosa_tmcuca ) );
    memset( &CosetUca, 0x00, sizeof( t_diosa_cosetuca ) );
    memset( &Data, 0x00, sizeof( Data ) );

    /* C0 タイマ登録(インターバル型:15 秒×10 回) */
    snprintf( TmcUca.TimerId, 17, "TMC_TEST" );
    TmcUca.Mode = DIOSA_INTER;
    TmcUca.Count = 10;
    TmcUca.KeyCheck = DIOSA_OFF;

    TmcUca.Time.Year = 0;
    TmcUca.Time.Month = 0;
    TmcUca.Time.Day = 0;
    TmcUca.Time.Hour = 0;
    TmcUca.Time.Minute = 0;
    TmcUca.Time.Second = 15;

    snprintf ( Data, 200, "CO_TEST_MSG" );

    snprintf ( CosetUca.CoName, 31, "TCORECV" );
    snprintf ( CosetUca.TxId, 7, "TR0101" );
    CosetUca.Textalen = ( short )sizeof( "CO_TEST_MSG" );

    Ret = diosatmccoset( &TmcUca, &CosetUca, Data );

    if(( DIOSA_DONE != Ret ) && ( DIOSA_ALREADY != Ret )){
        printf( " *** diosatmccoset ERROR(%d)¥n", Ret );
    }else{
        printf( " *** diosatmccoset DONE (%d)¥n", Ret );
    }
}
```

```

    return Ret;
}

```

(b) 時刻指定

利用例

```

#include <diosa.h>

int SAMPLESET() {

    int          Ret;
    int          RetCmt;
    int          RetRbk;
    t_diosa_tmucu TmcUca;
    t_diosa_cosetuca CosetUca;
    char          Data[200];

    /* 初期化 */
    memset( &TmcUca, 0x00, sizeof( t_diosa_tmucu ) );
    memset( &CosetUca, 0x00, sizeof( t_diosa_cosetuca ) );
    memset( &Data, 0x00, sizeof( Data ) );

    /* CO タイマ登録(時刻指定型) */
    snprintf( TmcUca.TimerId, 17, "TMC_TEST" );
    TmcUca.Mode = DIOSA_CLOCK;
    TmcUca.KeyCheck = DIOSA_OFF;

    TmcUca.Time.Year = 2012;
    TmcUca.Time.Month = 7;
    TmcUca.Time.Day = 7;
    TmcUca.Time.Hour = 12;
    TmcUca.Time.Minute = 0;
    TmcUca.Time.Second = 0;

    snprintf ( Data, 200, "CO_TEST_MSG" );

    snprintf ( CosetUca.CoName, 31, "TCORECV" );
    snprintf ( CosetUca.TxId, 7, "TR0101" );
    CosetUca.Textalen = ( short )sizeof( "CO_TEST_MSG" );

    Ret = diosatmccoset( &TmcUca, &CosetUca, Data );

    if(( DIOSA_DONE != Ret ) && ( DIOSA_ALREADY != Ret )){
        printf( " *** diosatmccoset ERROR(%d)¥n", Ret );
    }else{
        printf( " *** diosatmccoset DONE (%d)¥n", Ret );
    }

    return Ret;
}

```

(2) タイマ削除

タイマを削除するには、diosatmccreset 関数を利用します。diosatmccreset 関数を利用する際には、タイマの制御情報(t_diosa_tmucu)内のタイマ ID を設定した上で関数を呼び出す必要があります。

※CO 制御やバッチ AP 制御を利用せずにタイマ削除アプリを作成する場合には、diosatmccreset 関数に加えて diosatmcccommit 関数を組み合わせて実行する必要があります。

利用例

```

#include <diosa.h>

int SAMPLEDEL() {

    int          Ret;
    int          RetCmt;
    int          RetRbk;

```

```

t_diosa_tmcuca TmcUca;

/* 初期化 */
memset( &TmcUca, 0x00, sizeof( t_diosa_tmcuca ) );

/* タイマ削除 */
snprintf( TmcUca.TimerId, 17, "TMC_TEST" );

Ret = diosatmcreset( &TmcUca );

if( DIOSA_DONE != Ret ){
    printf( " *** diosatmcreset ERROR(%d)¥n", Ret );
} else{
    printf( " *** diosatmcreset DONE (%d)¥n", Ret );
}

return Ret;
}

```

(3) タイマ実行によって C0 が受信する電文形式

C0 タイマが実行されると、タイマ登録時に指定した C0 制御情報に従って登録電文が送信されます。

このとき C0 が受信する電文は、タイマ登録時に送信メッセージとして登録した電文となります。(ただし、登録情報内の電文サイズに誤りがないことを前提とします)

不要なヘッダ情報等が挿入されることはありません。

3.2 ユーザアプリケーションプログラム

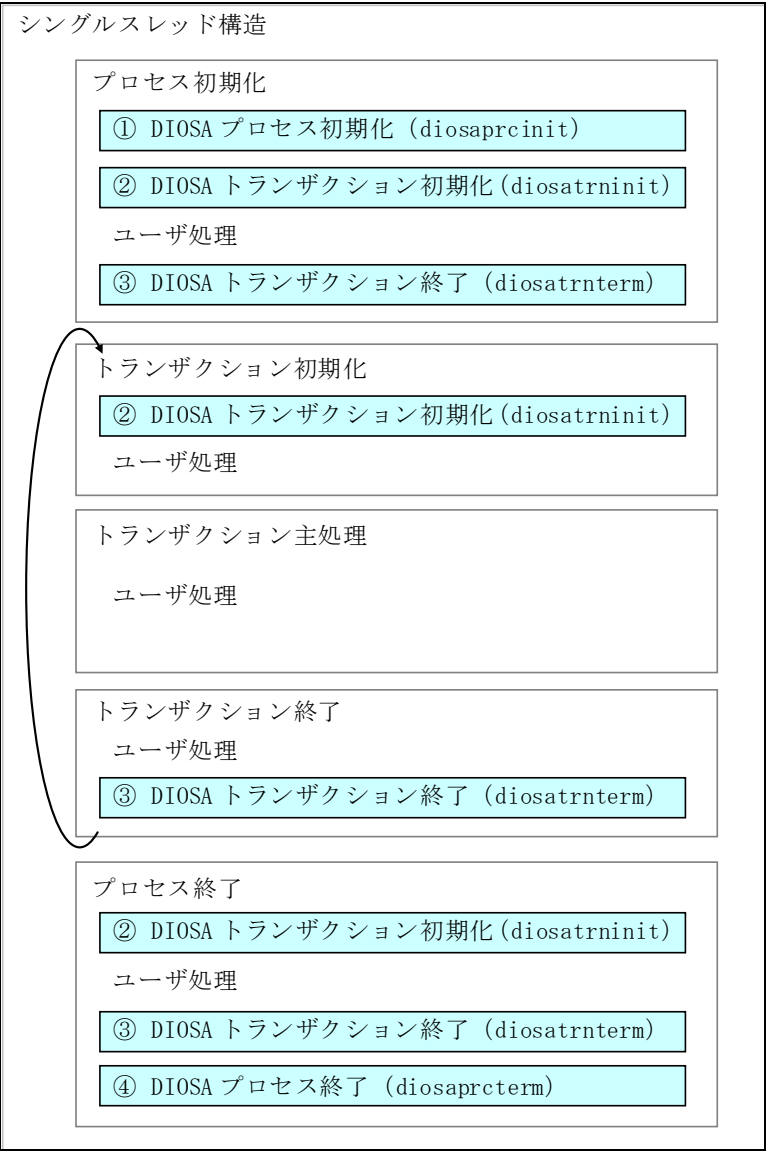
本章では、ユーザアプリケーションの作り方について説明します。

3.2.1 プログラムの構造

ユーザアプリケーションを作るためには、必要な初期化処理や終了処理を適切な順番に従って実行する必要があります。シングルスレッド構造の場合とマルチスレッド構造の場合について、それぞれのプログラム構成を説明します。

(1) シングルスレッド構造

シングルスレッド構造のプログラム構成を説明します。



① DIOSAプロセス初期化

プログラムの先頭で実行します。本APIでは、DIOSAの各種APIを使用するための準備を行います。

② DIOSAトランザクション初期化

DIOSAプロセス初期化の後やトランザクション処理の先頭で実行します。本APIのタイミングでSG動的変更の内容が当該プロセスに反映されます。

D I O S A の各種 A P I は、本 D I O S A トランザクション初期化から D I O S A トランザクション終了までの区間で使用することができます。

③ D I O S A トランザクション終了

D I O S A トランザクション初期化を行った場合に、トランザクション終了の最後に本 A P I を実行します。なお、本 D I O S A トランザクション終了後に D I O S A の各種 A P I を使用する場合は、再度 D I O S A トランザクション初期化を実行してください。

トランザクション初期化が異常終了した場合でも、次のトランザクションを開始する前には、必ず D I O S A トランザクション終了を実行してください。

④ D I O S A プロセス終了

プログラムの最後に実行します。

なお、例外発生等で D I O S A トランザクション終了を呼び出さずにプロセスを終了する場合、D I O S A プロセス終了処理も呼び出さないようにしてください。

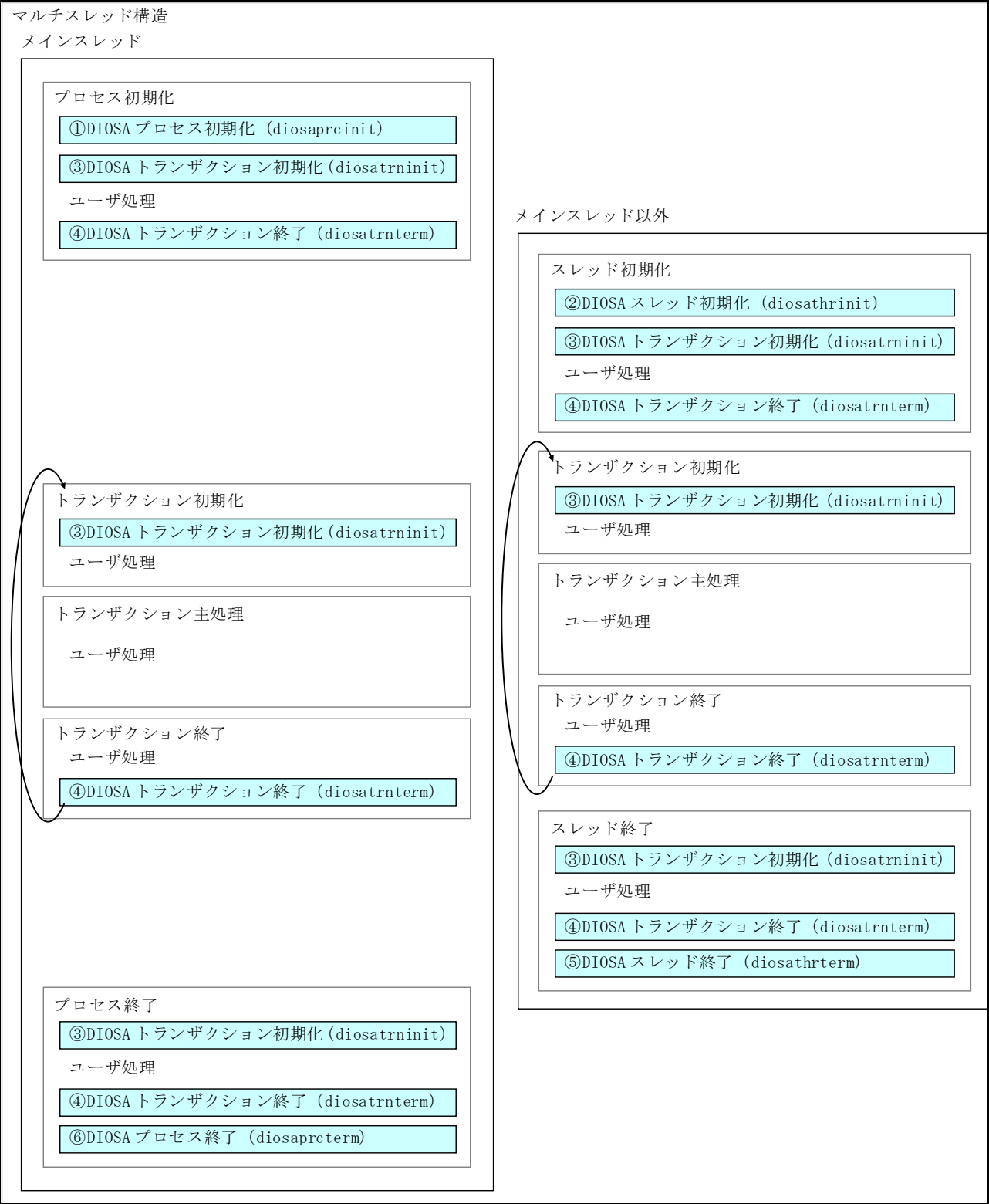
プロセス終了後、再度プロセス初期化処理を呼び出して処理を継続することはできません。

(2) マルチスレッド構造

マルチスレッド構造のプログラム構成を説明します。

なお、マルチスレッド構造をとるプログラムから子プロセスを生成することはできません。

また、子スレッドの生成は下記図中の③と④の間で行ってください。



- ① DIOSAプロセス初期化
- シングルスレッド構造と同様に、プログラムの先頭で実行します。本APIでは、DIOSAの各種APIを使用するための準備を行います。
- ② DIOSAスレッド初期化
- メインスレッド以外のスレッドの先頭で実行します。DIOSAプロセス初期化を行うメインスレッドでは実行する必要はありません。本APIでは、スレッドでDIOSAの各種APIを使用するための準備

備を行います。

なお、スレッド生成前のメインスレッドにおいて予めD I O S Aプロセス初期化が行われている必要があります。

③ D I O S Aトランザクション初期化

D I O S Aプロセス初期化やD I O S Aスレッド初期化の後、及びトランザクション処理の先頭で実行します。本A P IのタイミングでS G動的変更の内容が当該プロセスに反映されます。

D I O S Aの各種A P Iは、本D I O S Aトランザクション初期化からD I O S Aトランザクション終了までの区間で使用することができます。

④ D I O S Aトランザクション終了

D I O S Aトランザクション初期化を行った場合に、トランザクション終了の最後に本A P Iを実行します。なお、本D I O S Aトランザクション終了後にD I O S Aの各種A P Iを使用する場合は、再度D I O S Aトランザクション初期化を実行してください。

トランザクション初期化が異常終了した場合でも、次のトランザクションを開始する前には、必ずD I O S Aトランザクション終了を実行してください。

⑤ D I O S Aスレッド終了

スレッドの最後に実行します。

⑥ D I O S Aプロセス終了

プログラムの最後に実行します。なお、D I O S Aプロセス終了は、必ず全ての子スレッドが終了してから(⑤のD I O S Aスレッド終了が完了してから)実行してください。

なお、例外発生等でD I O S Aトランザクション終了を呼び出さずにプロセスを終了する場合、D I O S Aプロセス終了処理も呼び出さないようにしてください。

プロセス終了後、再度プロセス初期化処理を呼び出して処理を継続することはできません。

3.3 通信制御プログラミング

本章では、各種利用者出口の作り方について説明します。

3.3.1 電文種別決定出口

DIOSA/XTP 以外のシステムと接続する場合、通信に用いる電文の形式は DIOSA/XTP が使用する電文形式（DIOSA プロトコル）とは限りません。そのため、DIOSA プロトコル以外にも対応できるよう、TPBASE のリスナ出口において受信電文を解析する利用者出口を提供します。

(1) **使用方法**

1. GetMsgTypeExit という関数を含む libdxtpnmsgtype.so というライブラリを作成します。
2. 作成したライブラリを DIOSA のインストールディレクトリよりも読込優先順位の高いパスに配置します。

(2) **呼出契機**

リスナが DIOSA プロトコル以外の電文を受信した場合に呼び出されます。

(3) **プログラムインタフェース**

リスナ出口から呼び出され、引数として受信電文の先頭ポインタと読込済みの受信電文長を与えます。利用者は受信電文の解析結果を元に総電文長とトランザクション ID を返却します。

```
int GetMsgTypeExit(t_diosa_getmsgtypeuca *p_pstGetMsgTypeUca)
```

○t_diosa_getmsgtypeuca 構造体
void *MsgPtr 受信電文（入力型）
int LdDtLen 読込み済み電文長（入力型）
int MsgLen 総電文長（出力型）
char TxId[7] トランザクション ID（出力型）

(4) **プログラム作成方法**

1. パラメータで渡された受信電文がトランザクション ID と総電文長を決定できるだけの長さを持っているか確認します。
2. 受信電文を解析し、起動するトランザクション ID と総電文長を取得します。
トランザクション ID、総電文長を取得できない場合は異常終了します。

(5) **注意事項**

ファイルアクセスなど本利用者出口の処理が遅延する原因となるような処理を行ってはいけません。
リスナ出口はマルチスレッドのためスレッドセーフに作成する必要があります。
本出口で利用者に渡される受信電文の長さは TPBASE の SG である MSGHD_SIZE で指定された長さです。
本出口で利用者に渡される受信電文のバッファはバイト境界が正しいことを保証しません。
出口名は GetMsgTypeExit に固定とします。

3.3.2 データベース接続出口

データベース接続出口は、データベース管理機能から呼び出され、接続出口内でデータベースに接続し、SQL コンテキストをデータベース管理機能へ返却するためのサブルーチンです。

(1) **環境定義**

データベース接続出口を使用するためには、次のような環境定義をしなければなりません。使用しない場合の定義は不要です。

```
【データベース管理】
$DBCTRL
%CONTROL
    CONNECTEXIT = データベース接続出口関数名
;
```

(2) **呼出契機**

アプリケーションからデータベースの接続要求があった場合、本出口が呼び出されます。

(3) **プログラムインタフェース**

データベース管理機能から呼び出され、引数として接続先ネット・サービス名を与えます。利用者は接続先ネット・サービス名を元に、データベース接続を行ったSQL コンテキストを返却します。

```
int データベース接続出口名(t_diosa_dbinfouca *p_DbInfoUca)

○t_diosa_dbinfouca 構造体
char DbName[137] 接続先ネット・サービス名（入力型）
sql_context * SqlCtx SQL コンテキスト（出力型）
```

3.4 アプリケーションの生成

3.4.1 C O プログラム

メイン C0 および下位層の C0・B0、各種出口は、共有ライブラリとして生成します。

mainco_1、mainco_2、bo_1、bo_2、…、bo_5 の C0・B0 から構成される共有ライブラリ libsample.so を生成する例を示します。

(1) **コンパイル**

C0・B0、各種出口をコンパイルするために、オプションとして以下を指定します。

オプション	説明
+DD64	64bit オブジェクトを生成するように指定します。
-D_POSIX_C_SOURCE=199506L	POSIX スレッドが使用可能となるように指定します。
-I DIOSA/XTP インストールディレクトリ/include	DIOSA/XTP インクルードファイル格納位置を指定します。(※1)

※1 開発マシンの環境に合わせてパスを指定してください。

以下にコンパイル実行例を記載します。

```
# cc -c +DD64 -D_POSIX_C_SOURCE=199506L -I $(DIR_DIOSA)/include mainco_1.c
```

(2) **実行ファイルの生成**

共有ライブラリを生成するために、必要となるオプションはありません。

以下にリンク実行例を記載します。

```
# ld -b -o libsample.so mainco_1.o mainco_2.o bo_1.o … bo_5.o
```

3.4.2 ユーザアプリケーションプログラム

(1) コンパイル

プログラムをコンパイルするために、オプションとして以下を指定します。

オプション	説明
+DD64	64bit オブジェクトを生成するように指定します。
-mt	マルチスレッドが使用可能となるように指定します。
-D_POSIX_C_SOURCE=199506L	POSIX スレッドが使用可能となるように指定します。
-I DIOSA/XTP インストールディレクトリ/include	DIOSA/XTP インクルードファイル格納位置を指定します。(※1)

※1 開発マシンの環境に合わせてパスを指定してください。

以下にコンパイル実行例を記載します。

```
# cc -c +DD64 -mt -D_POSIX_C_SOURCE=199506L -I $(DIR_DIOSA)/include sample.c
```

(2) 実行ファイルの生成

実行ファイルを生成するために、オプションとして以下を指定します。

オプション	説明
-L DIOSA/XTP インストールディレクトリ/lib	DIOSA/XTP ライブラリファイル格納位置を指定します。(※1)
-L Oracle インストールディレクトリ/lib	Oracle Database ライブラリファイル格納位置を指定します。(※1)
-L TPBASE インストールディレクトリ/lib	TPBASE ライブラリファイル格納位置を指定します。(※1)
-ldxtp	DIOSA/XTP の API が使用できるようにライブラリ (libdxtp.so) を指定します。
-lpthread	POSIX スレッドのライブラリを指定します。

※1 開発マシンの環境に合わせてパスを指定してください。

対象プロダクトをインストールしていない場合は、指定不要です。

以下にリンク実行例を示します。

※紙面の都合により 2 行で記載しています。

```
# cc -o sample sample.o -L $(DIR_DIOSA)/lib -L $(DIR_ORACLE)/lib
-L $(DIR_TPBASE)/lib -ldxtp -lpthread
```

3.4.3 利用者出口

CO 制御サーバ上で動作する各種出口のプログラミングと出口の生成方法について説明します。
なお、バッチ AP 制御を使ったアプリケーションも同様の方法で生成できます。

(1) **EXIT のプログラミング**

受信電文解析出口を除き、各種出口の形式は以下のようになります。

```
#include <diosa.h>

void exit_XXX( t_diosa_uca *uca )
```

受信電文解析出口の形式は以下のようになります。

```
#include <diosa.h>

void exit_MSGANL( t_diosa_uca *uca, t_diosa_analyze *ana )
```

(2) **コンパイル**

プログラムをコンパイルするために、オプションとして以下を指定します。

オプション	説明
+DD64	64bit オブジェクトを生成するように指定します。
-D_POSIX_C_SOURCE=199506L	POSIX スレッドが使用可能となるように指定します。
-I DIOSA/XTP インストールディレクトリ/include	DIOSA/XTP インクルードファイル格納位置を指定します。(※1)

※1 開発マシンの環境に合わせてパスを指定してください。

以下にコンパイル実行例を記載します。

```
# cc -c +DD64 -D_POSIX_C_SOURCE=199506L -I $(DIR_DIOSA)/include sample_exit.c
```

(3) **実行ファイルの生成**

各種出口は、共有ライブラリとして生成します。

実行ファイルを生成するために、オプションとして以下を指定します。

オプション	説明
-L DIOSA/XTP インストールディレクトリ/lib	DIOSA/XTP ライブラリファイル格納位置を指定します。(※1)
-L Oracle インストールディレクトリ/lib	Oracle Database ライブラリファイル格納位置を指定します。(※1)
-L TPBASE インストールディレクトリ/lib	TPBASE ライブラリファイル格納位置を指定します。(※1)
-ldxtp	DIOSA/XTP の API が使用できるようにライブラリ (libdxtp.so) を指定します。
-lpthread	POSIX スレッドのライブラリを指定します。

※1 開発マシンの環境に合わせてパスを指定してください。

対象プロダクトをインストールしていない場合は、指定不要です。

以下にリンク実行例を示します。

※紙面の都合により 2 行で記載しています。

```
# cc -o sample sample_exit.o -L $(DIR_DIOSA)/lib -L $(DIR_ORACLE)/lib  
-L $(DIR_TPBASE)/lib -ldxtp -lpthread
```

D I O S A / X T P V1.1
利用の手引

2016 年 6 月 10 版

日本電気株式会社
東京都港区芝五丁目 7 番 1 号
TEL (03) 3454-1111 (大代表)

©NEC Corporation 2011, 2016

日本電気株式会社の許可なく複製・改変などを行うことはできません。
本書の内容に関しては将来予告なしに変更することがあります。