DIOSA/XTP V2.1

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国政府の輸出許可が必要です。

なお、輸出許可申請手続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 DIOSA/XTP 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、OS、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。

2017 年 3 月 3 版

本書の関連説明書としては次のものがあります。

- DIOSA/XTP 導入の手引き
- DIOSA/XTP メモリキャッシュ 利用の手引き
- DIOSA/XTP データストア 利用の手引き
- DIOSA/XTP データ変換・通信オプション 導入の手引き
- DIOSA/XTP データ変換・通信オプション 利用の手引き
- DIOSA/XTP API リファレンス
- DIOSA/XTP コマンドリファレンス
- DIOSA/XTP 環境定義リファレンス
- DIOSA/XTP メッセージリファレンス

目次

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・高稼動性、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・高稼動性を実現します。

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発生産性向上
- 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動を支援します。

- オンライン業務の稼動中に、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動中に、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間に立ち、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。

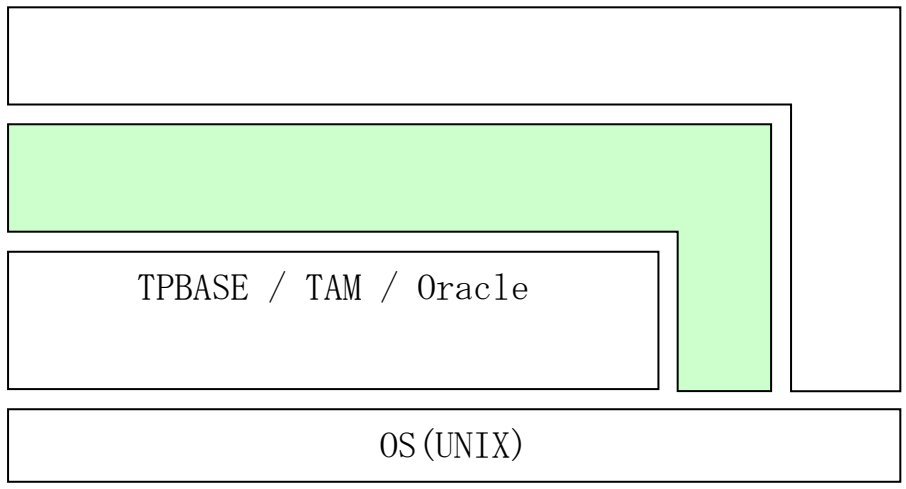
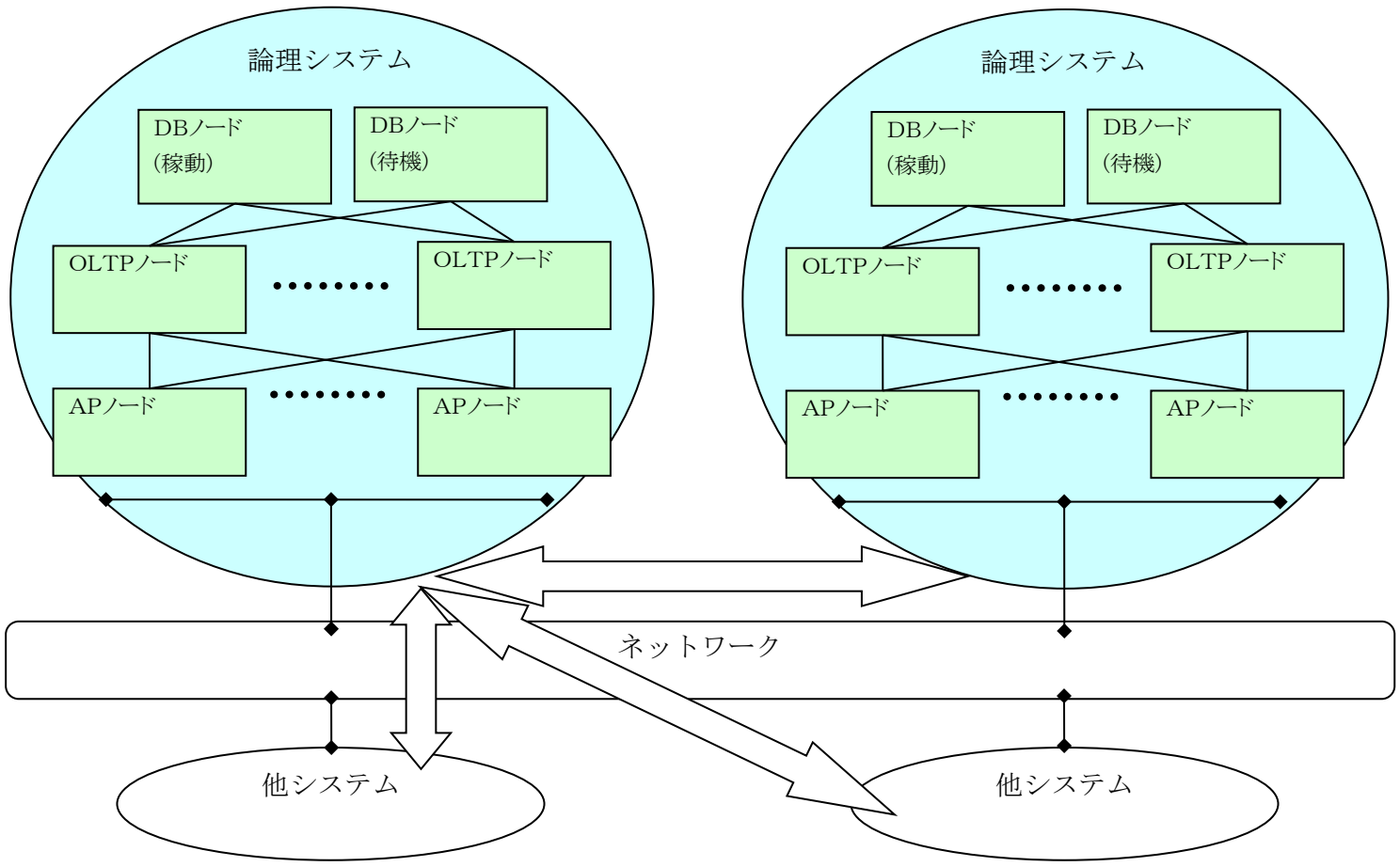


図 1-1 DIOSA/XTP の位置付け

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	論理ノード間通信管理機能 論理システム間通信管理機能 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)　論理ノード間パス

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)　論理システム間パス

論理システム間で確立する論理的な通信パスの事を指します。

(5)　CO(制御)/出口

CO 制御機能(サーバアプリケーションの制御用フレームワーク)、バッチアプリケーション制御機能から呼び出される業務プログラムのことを CO(Control Object)と呼びます。CO 制御機能では、CO をイベント(メッセージ)により連結して処理することが可能です。

CO 制御サーバ上で動作する業務プログラムのためのプロセス初期化処理など、定型的な処理をするための処理を出口と呼びます。DIOSA/XTP の各機能では、決められたタイミングで出口(業務プログラム)を呼び出しています。

CO 制御は、TPBASE の TPP として動作します。本マニュアルでは、「TPBASE」、「TP モニタ」、「TPBASE モニタ」は同意となります。

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 を呼び出しておこないます。CO をイベントで連携動作させることで、共通関数呼び出しで業務 AP を設計するよりも、障害の局所化やプログラムの簡素化が可能となります。

DB は Oracle と TAM(本マニュアルでは IM(インメモリキャッシュ)と表記される場合もあります)が利用できます。

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

受信した電文毎の業務に対応した CO を呼び出します。

また、CO のように業務に依存しない定型業務として、プロセス初期化/終了、トランザクション初期化/終了等の利用者出口を呼び出すこともできます。利用者出口は環境定義することで呼び出し可能となります。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すため、運用中に置換することができます。(詳細はアプリケーション動的置換機能を参照してください)

(2)　CO、TXID 所在管理機能

AP が電文(以降、断りがない限りメッセージ、イベントは同じ意味です)の送信をおこなう時、送信先の CO 名または TXID(TPBASE のトランザクション ID)を意識するだけで電文送信が可能となります(電文送信インタフェースの簡略化)。

CO 名と TXID は、環境定義($COCENV)することができます。CO 制御は環境定義から CO 名、または TXID の存在する、論理ノード名、TPBASE モニタ名を決定して電文送信を実行します。

(3)　電文送受信機能

電文の送受信をおこないます。

受信 API(diosarecvtx)と送信 API(diosasendtx)を提供し、利用者は論理システム間(外部論理システム宛)、論理システム内(内部論理システム宛)等の宛先により API を区別することなく送受信することができます。

論理システム内の電文送信では、CO、TXID 所在管理機能を使うことができます。

(4)　CO 間連携機能

論理システム内で複数の CO が連携した業務処理をおこなうことが可能です。

CO 間連携機能には、連鎖機能、派生機能があります。

(5)　コミット/ロールバック機能

CO 制御は AP にかわり、コミットまたはロールバックを行います。AP はトランザクション処理が正常終了、異常終了するのかを CO 制御に通知するだけで、コミット、ロールバックは自動(暗黙)的に実行されます。

また、利用者が任意(明示的)でコミット、ロールバックできるAPIを提供します。

(6) **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。

(7) **APのループ/ストール監視機能**

COのCPU消費時間と経過時間を監視します。

CPU時間制限値を超えた場合、プロセス例外を発生させてアボート処理を呼び出します。

経過時間制限値を超えた場合、警告メッセージを出力するか、強制停止(プロセス例外)するかを環境定義で指定することができます。

(8) **ロールバック連鎖機能**

APが論理的に異常を検出した時に、現在のCO処理をロールバックして、事前に登録されたCOの実行を行います。

(9) **CO制御への強制リターン機能**

APが論理的に異常を検出した時やデッドロックを検出した時に、COBOLのGOBACK MAIN相当の動作が可能です。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。

(10) **ロールバックリトライ機能**

APが論理的に異常を検出した時に、ロールバックを行って再処理します。

(11) **デッドロックリトライ機能**

デッドロック発生時、ロールバックを行って再処理します。

(12) **電文保留機能**

受信電文を一時的に退避しておき、再処理することが可能です。

ロールバックリトライと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再実行しますが、電文保留機能の再処理は別トランザクションで遅延して再実行します。電文保留機能による再実行は、別ノードでおこなわれることもあります。

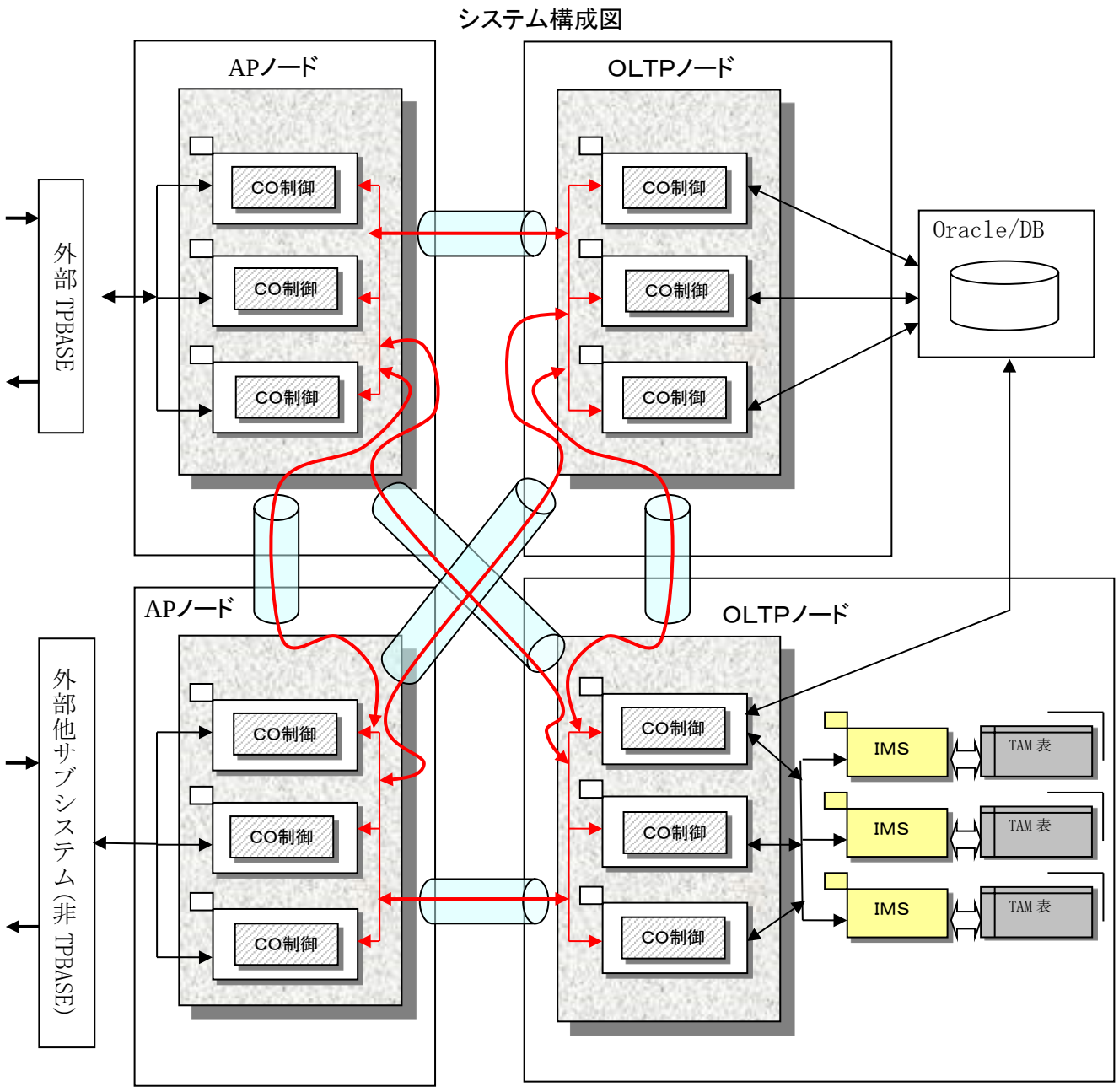
(13) **DBアクセス機能**

DBとして、OracleとTAM(IM)が利用できます。CO制御が更新対象とするDBは、Oracleの1インスタンスまたはTAMとなります。

利用者は更新したいDBを環境定義することができます。環境定義ではOracle、TAMとも更新可能DBとして定義することが可能となりますが、この場合更新するDBは、利用者が選択することになります。

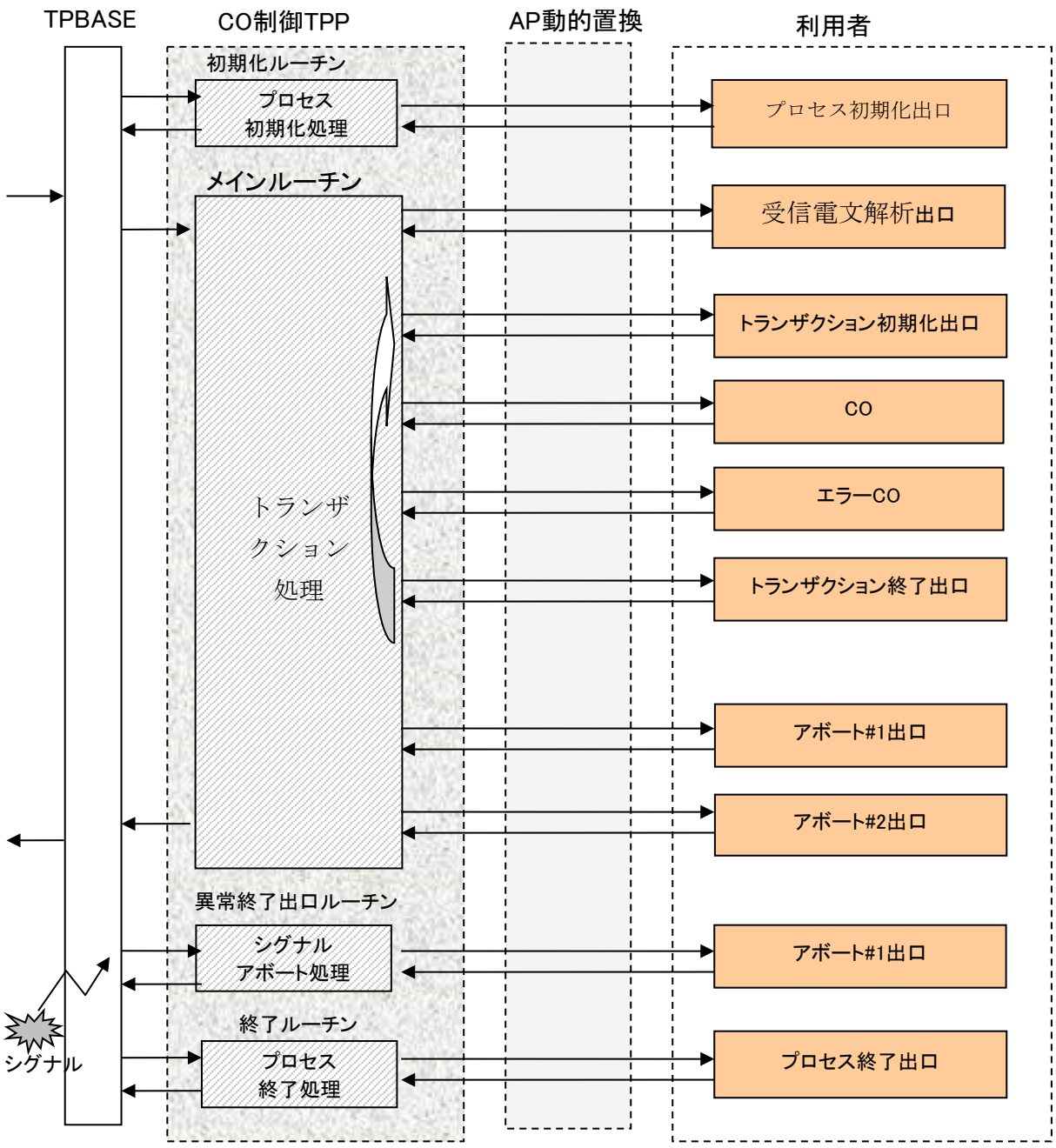
更新するDB以外の参照は自由です。Oracle更新情報をTAMから参照する。逆にTAM更新情報をOracleから参照することができます。ただし、同時更新は利用者の責任となります。

システム内の CO 制御と DB(Oracle、IM)の配置関係を以下に示します。



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運用中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
	CO	電文受信時(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) CO から連鎖要求時(連鎖 CO 呼び出し)
	エラーCO	CO の呼び出し失敗時、呼び出す CO が決定できない時、CO が閉塞されている時等
	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)。ただしアボート時を除きます。
	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(ロールバック前) 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)
	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 や利用者出口との間での情報交換には、DIOSAUCA(Diosa User Communication Area)と呼ばれる領域を使います。

AP は DIOSAUCA から実行環境の情報(ノード名、TPBASE 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)、アボート理由等を取得し、終了ステータス(正常/異常/リトライ)等の情報を CO 制御に返却します。

2.1.3　　CO、TXID 所在管理機能

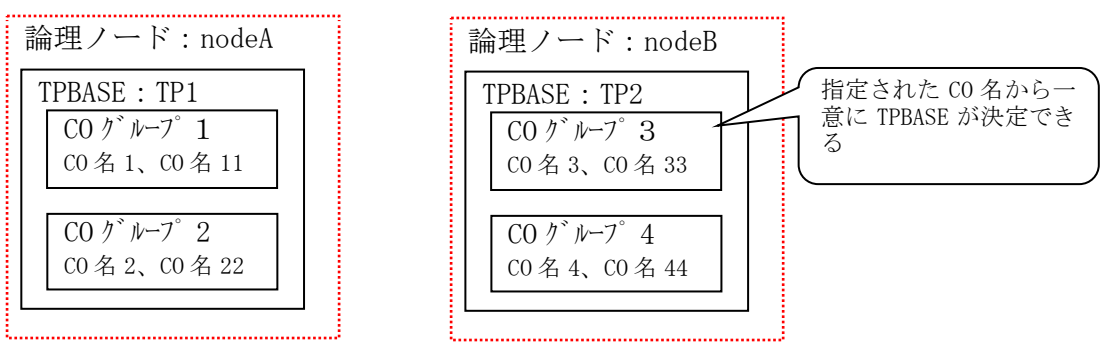
CO、TXID(トランザクション ID)の所在管理機能とは、CO、または TXID が存在する論理ノード、TPBASE モニタを管理する機能です。本機能を使うことにより利用者は CO 名、または TXID を指定するだけで電文送信が可能となります。本機能は、論理システム内のみ有効な機能です。

CO、TXID の所在を管理するため環境定義$DIOSAMAP と$COCENV が必要となります。論理ノードと論理ノード配下の TPBASE モニタを$DIOSAMAP に定義します。TPBASE モニタのクラス、トランザクション ID（TXID と同意）、CO 名等を$COCENV に定義します。

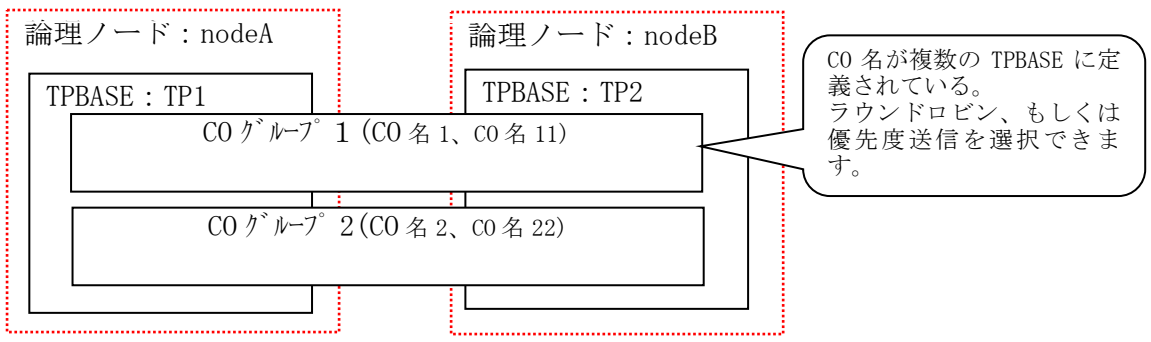
CO 名は CO グループと呼ばれるグループにより管理されます。CO グループは TXID に属する定義となります。CO 制御は CO 名から CO グループを検索して TXID を特定します。TXID が特定すると、論理ノード名、TPBASE モニタが特定されます。宛先が複数存在する場合は、電文送信のパラメータによりラウンドロビンか優先度を元に電文送信を行います。TXID 指定は CO 名の検索をせず TXID の検索をおこない、論理ノード名、TPBASE モニタを決定します。

CO グループ、TXID は、分散配置型と負荷分散型があり混在もサポートします。以下は CO グループを例とした配置図です。(TXID 指定は CO グループを TXID と読み替えてください)

【　CO グループ分散配置型　】



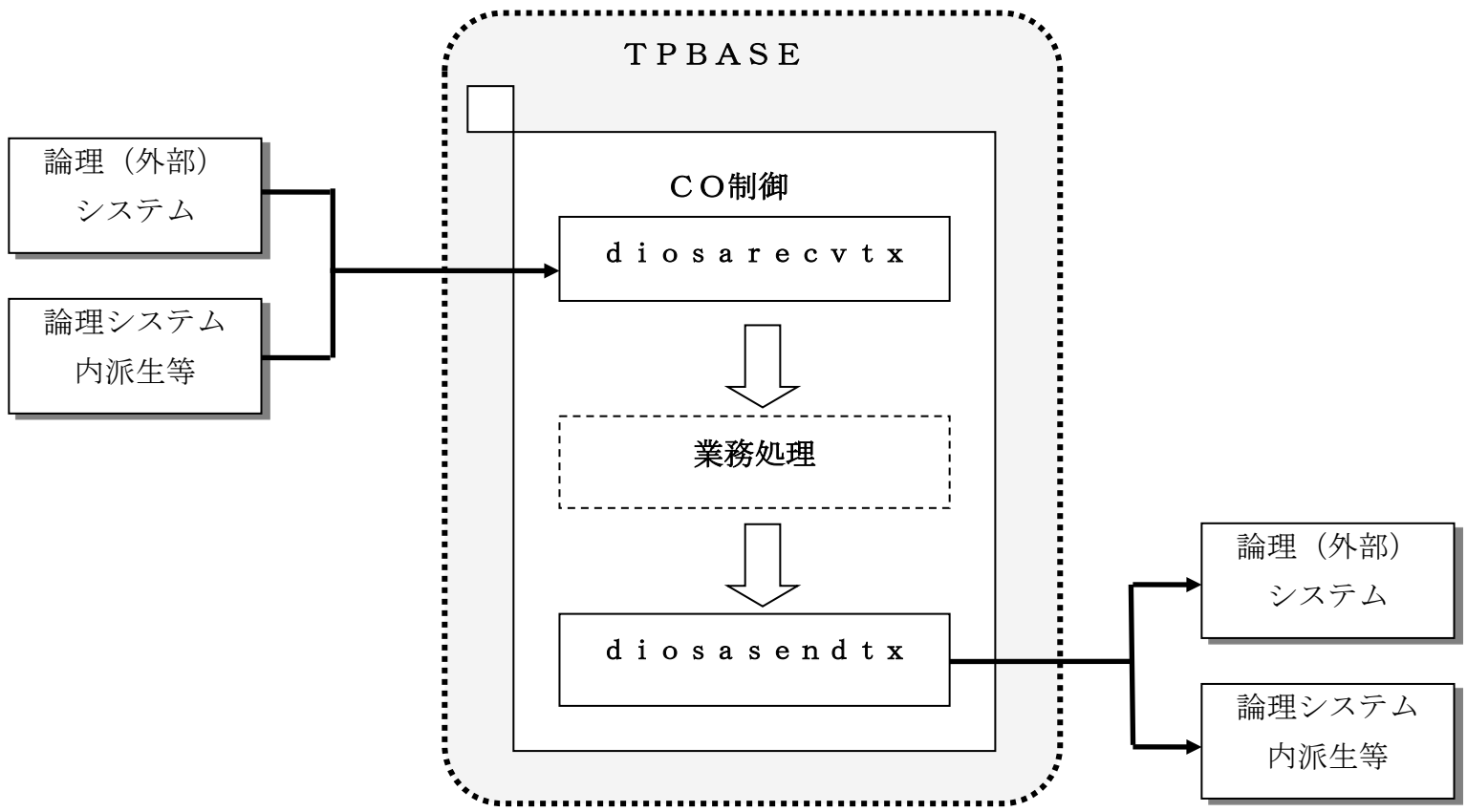
【　CO グループ負荷分散型　】



2.1.4　　電文送受信機能

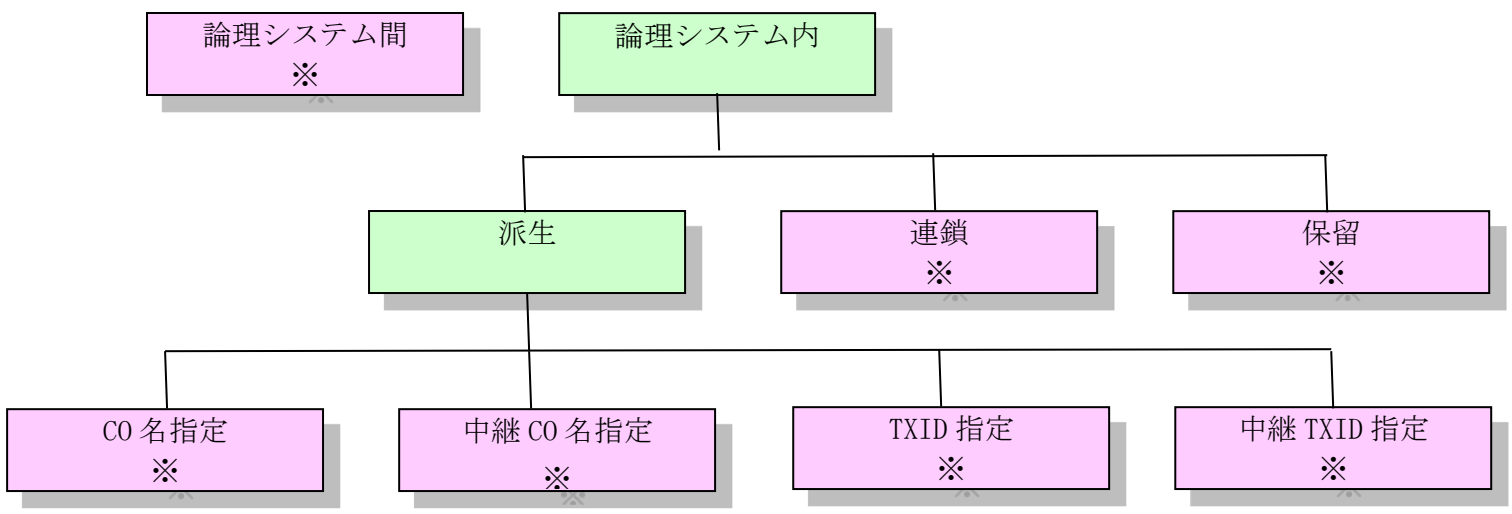
全ての電文送受信を API の diosarecvtx(電文受信)と diosasendtx(電文送信)で行うことができます。

電文送受信機能は、CO 制御 TPP 上でのみ使用することが可能です。



送信電文には、論理システム間(外部論理システム宛)、論理システム内に大別されます。論理システム内は、派生、連鎖、保留に分けられ、さらに派生は、CO 名指定、中継 CO 名指定、TXID 指定、中継 TXID 指定に分けられます。この電文の区分を電文タイプと呼び、利用者はこの電文タイプと宛先情報、およびその他機能を組み合わせて電文を送信します。

電文受信時は、送信時の電文タイプ、送信元情報(電文タイプにより参照可能情報が異なります)、また電文以外に利用者任意の情報を受け取ることもできます。



※：diosasendtx に指定できる電文タイプ

(1)　　論理システム間

送信宛て論理システムのアクセスポイント名、または端末名を指定して送信要求を行います、アクセスポイント名、端末名は、環境定義($SYSMAP)が必要です。論理システムによっては、端末名が指定できない(TPBASE モニタが端末名

を自動生成する)システムもあります。

(2) CO 名指定、中継 CO 名指定派生

CO 名を指定して送信要求を行います。CO 名が存在する論理ノード名、TPBASE モニタ名宛てに電文送信が可能です。CO 名の所在情報は環境定義($DIOSAMAP(論理ノード名、TPBASE モニタ名)、$COCENV(CO 名))に指定します。中継は派生先で論理システム間電文を送信するための論理システム間情報を指定して送信する形式であり、それ以外は CO 名の派生電文と同じです。

(3) TXID 指定、中継 TXID 指定派生

TXID を指定して送信要求を行います。TXID が存在する論理ノード名、TPBASE モニタ名宛てに電文送信が可能です。TXID の所在情報は環境定義($DIOSAMAP(論理ノード名、TPBASE モニタ名)、$COCENV(TXID))に指定します。中継は派生先で論理システム間電文を送信するための論理システム間情報を指定して送信する形式であり、それ以外は TXID の派生電文と同じです。

(4) 連鎖

同一トランザクション内電文送信を要求します。詳細は後述します。

(5) 保留

電文の一時保留を行います。詳細は後述します。

(6) その他宛先情報

CO 名指定、TXID 指定の送信要求により送信宛先が決定しますが、該当宛先が複数ある場合、その他宛先情報を付加することで、送信範囲を絞り込むことができます。絞り込み情報には、ノードタイプ(AP ノード、OLTP ノード)、論理ノード名、TPBASE モニタ名があります。DB に IM を使う場合は、IM のメインキー、MAPID を指定することで TAM マスタが存在する論理ノードを決定することも可能となります。

(7) ラウンドロビン送信と優先度送信

宛先が複数存在した場合、ラウンドロビンと優先度(通信コストのかからない宛先を優先する)送信を選択することが可能です。ラウンドロビンは全宛先をラウンドロビンに選択します。

優先度送信は、以下の順に送信先を決定します。

① 同じ TPBASE モニタ

② 同じ論理ノードの TPBASE モニタ、複数ある場合はラウンドロビン

③ 他論理ノードの TPBASE モニタ、複数ある場合はラウンドロビン

(8) 通常送信と強制送信

コミットと同期をとって電文を送信する通常送信と、即時に電文を送信する強制送信を選択することができます。通常送信はトランザクションを正常終了する、またはコミット API(diosacommit)発行すると送信が実行され、アボート要求、またはロールバック API(diosarollback)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、コミット 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信したい場合等は強制送信を指定します。

(9)　　電文保証

電文毎に電文保証有無を指定できます。

電文保証を行う場合、その旨を diosasendtx(電文送信)API に指定することで、該当電文が電文保証対象となります。詳細は電文保証機能を参照してください。

(10)　　共通 CO

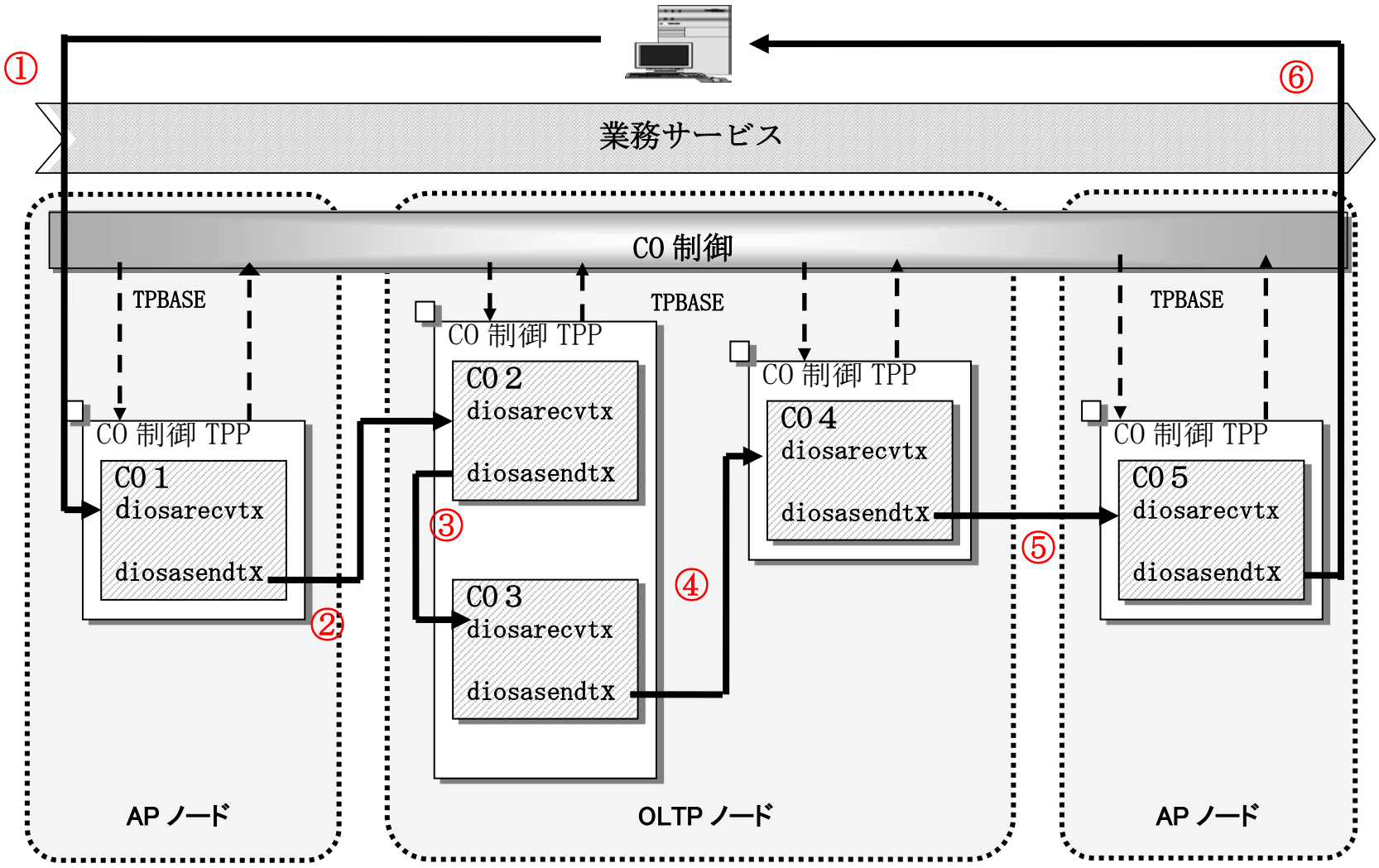
全ての論理ノード、全ての TPBASE モニタに存在する CO を共通 CO と呼びます。CO 名指定の派生、中継 CO 名指定の派生の時に有効となる機能です。

共通 CO は環境定義に定義する必要はありません。CO 制御は、環境定義されていない CO を共通 CO として、現在 CO 実行中の TPBASE モニタクラスのトランザクション ID で別トランザクション(プロセス)として実行します。

2.1.5　　CO 連携機能

CO 連携機能を利用することで、論理(内部)システム内で、複数の CO 群が連携して処理する実装が可能となります。CO 連携機能では、大別して 2 種類の連携方式(システム内派生/連鎖)を提供します。

以下のイメージ図では、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処理を、複数論理ノード・複数サーバプロセス上で動作する CO が連携して処理しています。



① 論理システム間（要求／応答）
② 論理システム内派生要求
③ （ＣＯ制御ＴＰＰプロセス)連鎖要求
④ 論理システム内派生要求
⑤ 論理システム内派生(中継)要求
⑥ 論理システム間（要求／応答）

(1)　　連鎖機能

受信電文に対する CO 処理から別の業務 CO を呼び出すことができる機能を連鎖機能と呼びます。

連鎖機能は、トランザクション区間内で実行可能な機能です。

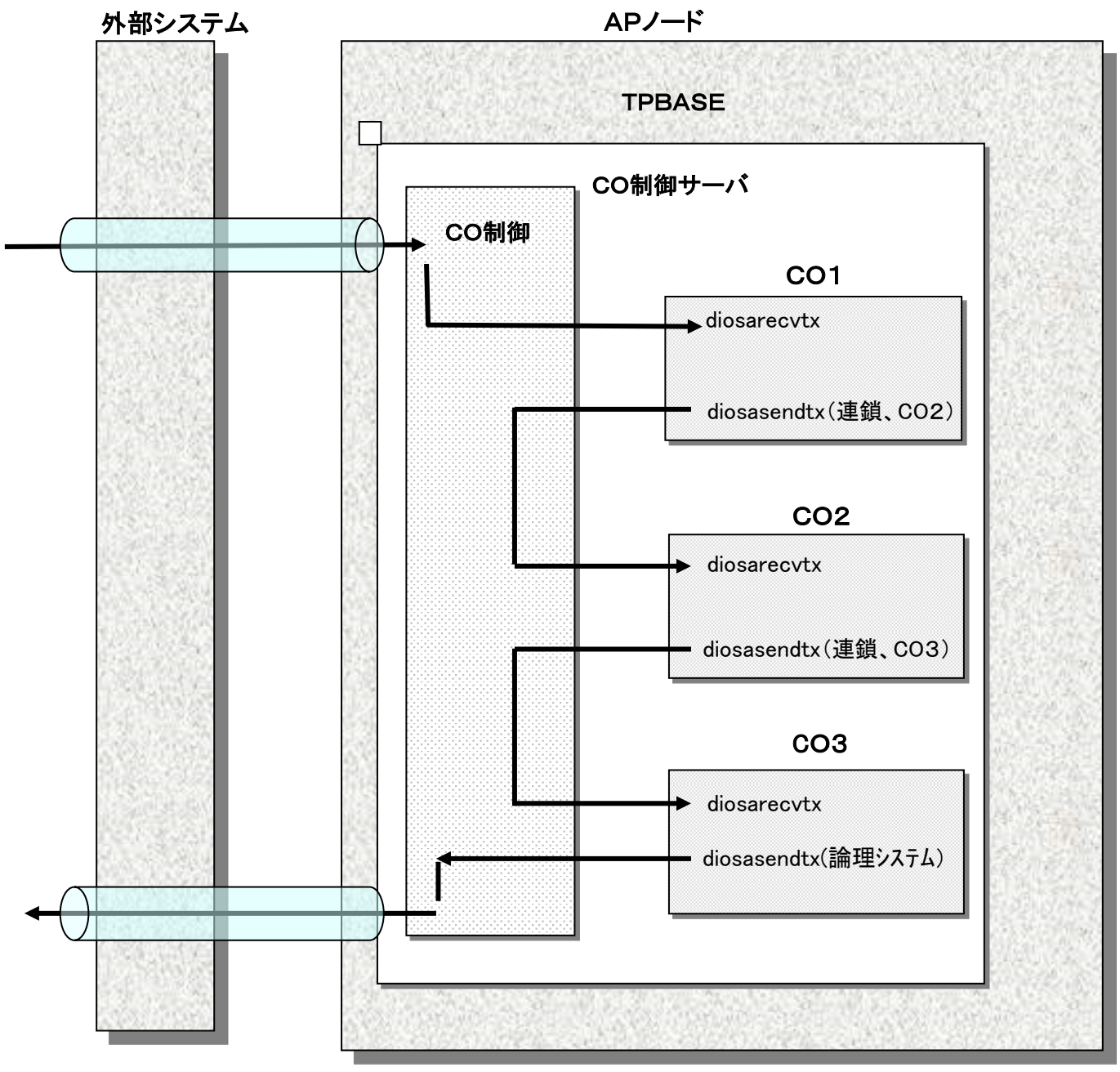
連鎖機能を使うことで以下のような処理をすることができます。

・受信電文処理に加えて別の処理を行い、同時コミットしたい。

・受信電文処理を確定後、別の処理を行いたい。

・受信電文処理をロールバックして、別の処理を行いたい。

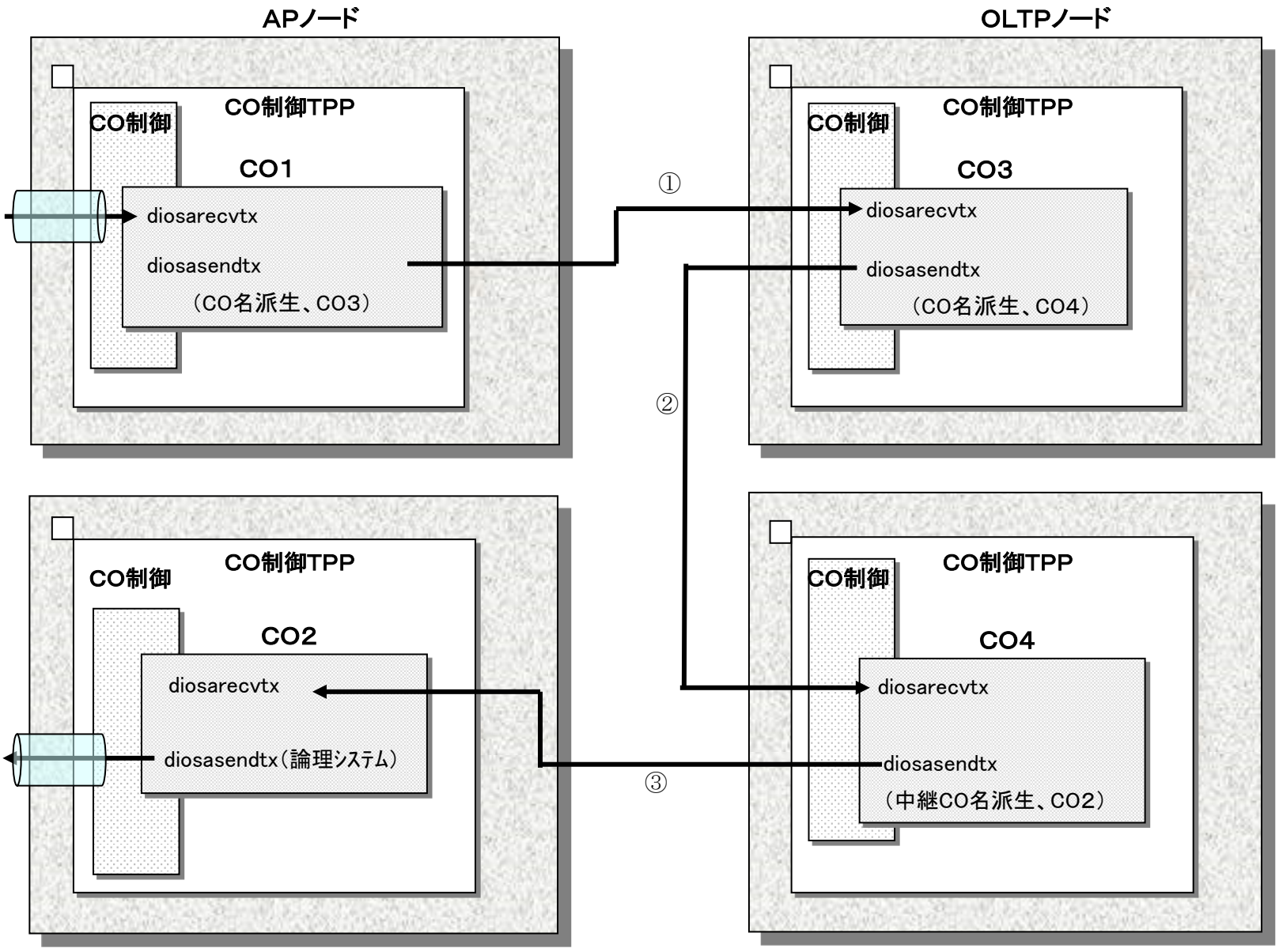
電文のコミット、ロールバックは利用者が明示的にコミット API(diosacommit)、ロールバック API(diosarollback)を使うか、CO 制御がおこなう暗黙のコミット、ロールバックを使うことができます。



(2)　　派生機能

別プロセス(他ノード、他 TPBASE モニタ、自 TPBASE モニタ)のトランザクションを呼び出すことのできる機能を派生機能と呼びます。

派生機能には、CO 名指定による派生と TXID 指定による派生があります。各派生には中継機能を持つ派生があるため、計 4 つの派生が存在します。以下の図は CO 名指定の派生と中継 CO 名指定の派生の呼び出し図です。



CO 制御は環境定義から CO が動作可能な TPBASE モニタを管理します。そのため、電文送信 API(diosasendtx)は、電文タイプに CO 名指定派生要求と CO 名を指定します。CO 制御は指定された CO 名が動作可能な TPBASE モニタを選択して電文を送信します。上図では CO3 は TPBASE3、CO4 は TPBASE4、CO2 は TPBASE2 に存在するため、①～③の呼び出しが可能となります。CO4 は呼び出し先 CO2 において、論理システム間に電文を送信したいため、中継 CO 名の派生を行います。

TXID 指定の派生は CO 名が TPBASE モニタの TXID に替わるだけです。

(a)　中継機能と論理システム間情報について

通常、他論理システムから電文を受信した場合、結果を電文送信元の他論理システムに返却(応答)します。ここで、AP ノードは他論理システムとの電文送受信を行う GateWay と位置づけ、業務処理は OLTP ノードで行わせるシステムを構築した場合、電文は AP ノード(電文受信)→OLTP ノード(業務)→AP ノード(電文送信)と流れていきます。最後の AP ノードに論理システム間の宛先情報を与えることができるのが中継機能です。CO 制御はこの間、送信元他論理システム情報と電文受信した AP ノードの情報を持回り、diosarecvtx で AP に情報を開示します(図では、CO3 と CO4 で持回り情報が参照できます)。CO4 は電文を送信、または返信する相手を決定して、その宛先情報を電文送信 API(diosasendtx)に指定します。電文を送信元の他論理システムに返却(応答)する場合は、CO 制御が持回った他論理システム情報を指定します。また、受信端末に電文を返却(応答)する場合は、これも CO 制御が持回った電文受信した AP ノード宛てに電文を送信して、そこで待っている端末宛てに電文を送信します。電文を返却(応答)するのではなく別の他論理システムに送信したい場合は、その論理システムの宛先情報を指定します。AP ノードは受け取った宛先情報に電文送信を行います。(宛先決定業務が AP ノードにある場合、中継機能は不要となります)

2.1.6　　コミット/ロールバック機能

CO 制御は DB に対するトランザクション開始、コミット、ロールバックなどのトランザクション制御を自動的におこないます。このため、AP は TAM(IM)アクセスや Oracle アクセスを使用した業務処理を作成すればよく、トランザクション制御処理を作成する必要がありません。

また、CO 処理中の任意のタイミングで AP がコミット、ロールバックを実行したい場合、コミット、ロールバックの API を提供します。

コミット、ロールバックは、更新 DB が IM の場合は TAM(IM)が実行対象となり、Oracle の場合は OracleDB インスタンスが実行対象となります。なお、複数 DB(Oracle＋IM、Oracle の複数インスタンス)を同時にトランザクション制御することはできません。

更新 DB は CO 制御の環境定義で、TPBASE モニタのクラス毎に定義することができます。また、クラス定義に、Oracle＋IM 両方を更新対象と定義しておき、電文毎に Oracle を更新対象とするのか、IM を更新対象にするのかを決定することも可能です。

(1)　　コミット/ロールバックの暗示的呼び出し機能

AP がトランザクション(CO 連携)処理を正常終了するとコミットがおこなわれます。コミット終了後は該当トランザクションが正常終了して、次トランザクション受付が可能となります。

逆に AP がトランザクションを異常終了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。

(2)　　コミット/ロールバックの明示的呼び出し機能

AP が CO 処理中にコミット、ロールバックを実行したい場合、コミット API(diosacommit)、ロールバック API(diosarollback)を呼び出します。

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。

2.1.7　　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(＃1、＃2)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＃1出口の呼び出しを行います。

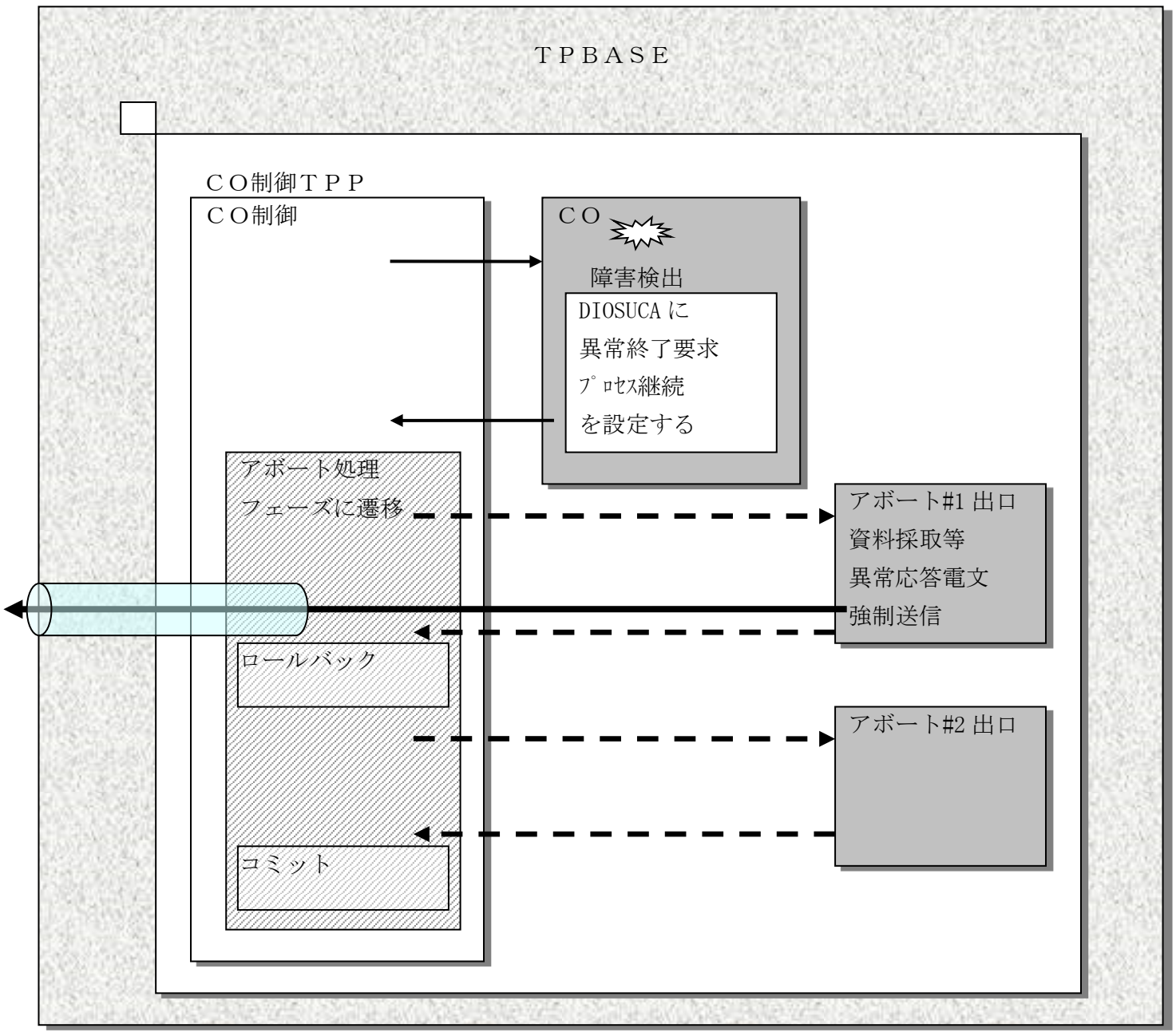
アボート出口には、アボート＃1出口とアボート＃2出口の2種類があります。

アボート＃1出口は障害解析用の資料採取を目的としたもので、アボート処理の最初に呼び出されます。アボート＃1出口呼び出し後にはロールバック処理が行われます。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)　　AP要求時のアボート処理

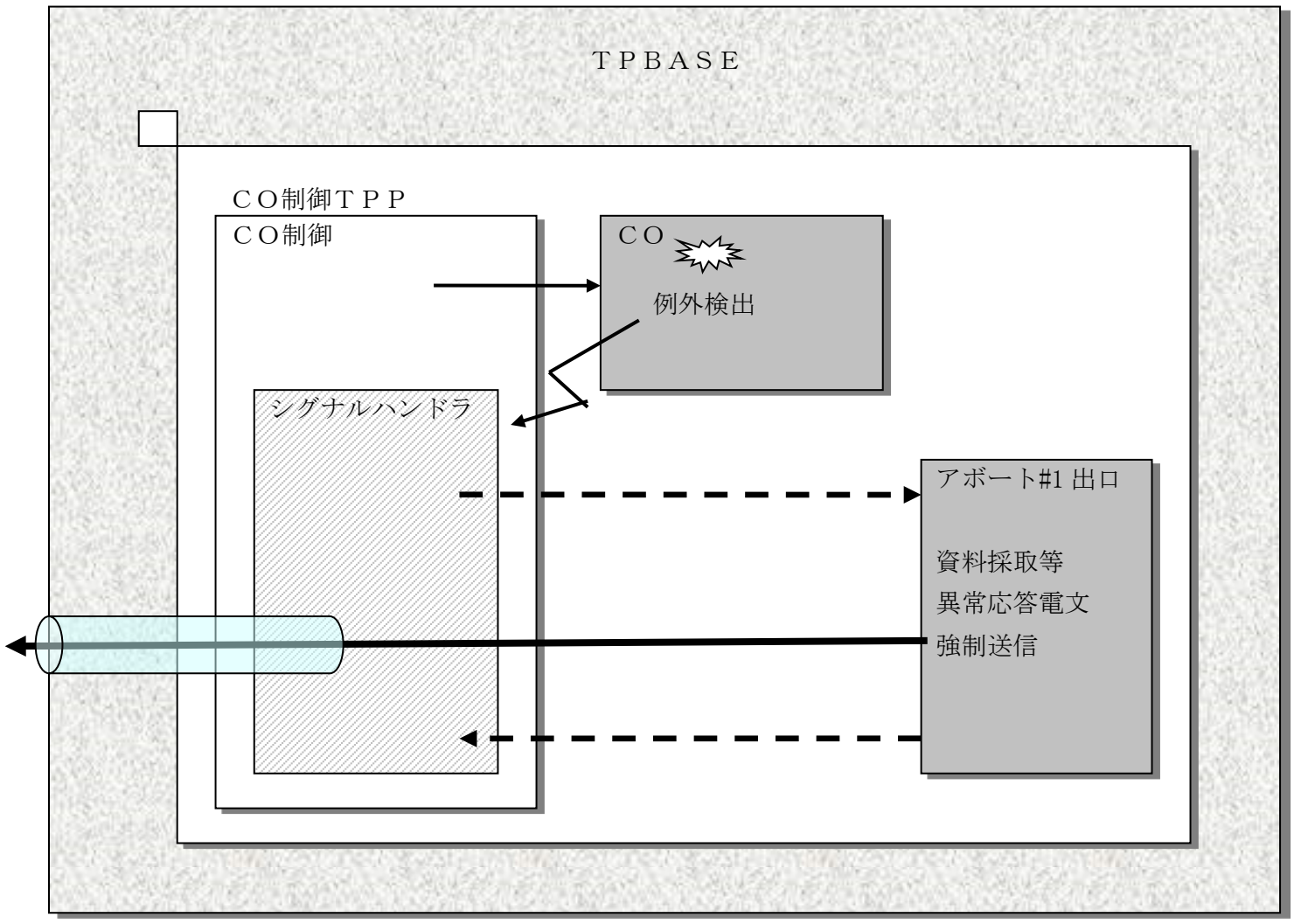
APから「異常終了要求」が返された時、CO制御はアボート処理を行います。異常終了後に該当プロセス停止有無を判定します。プロセス停止有無は、環境定義とCO終了時にAPから返される「異常終了要求」のコードにより決まります。



(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。

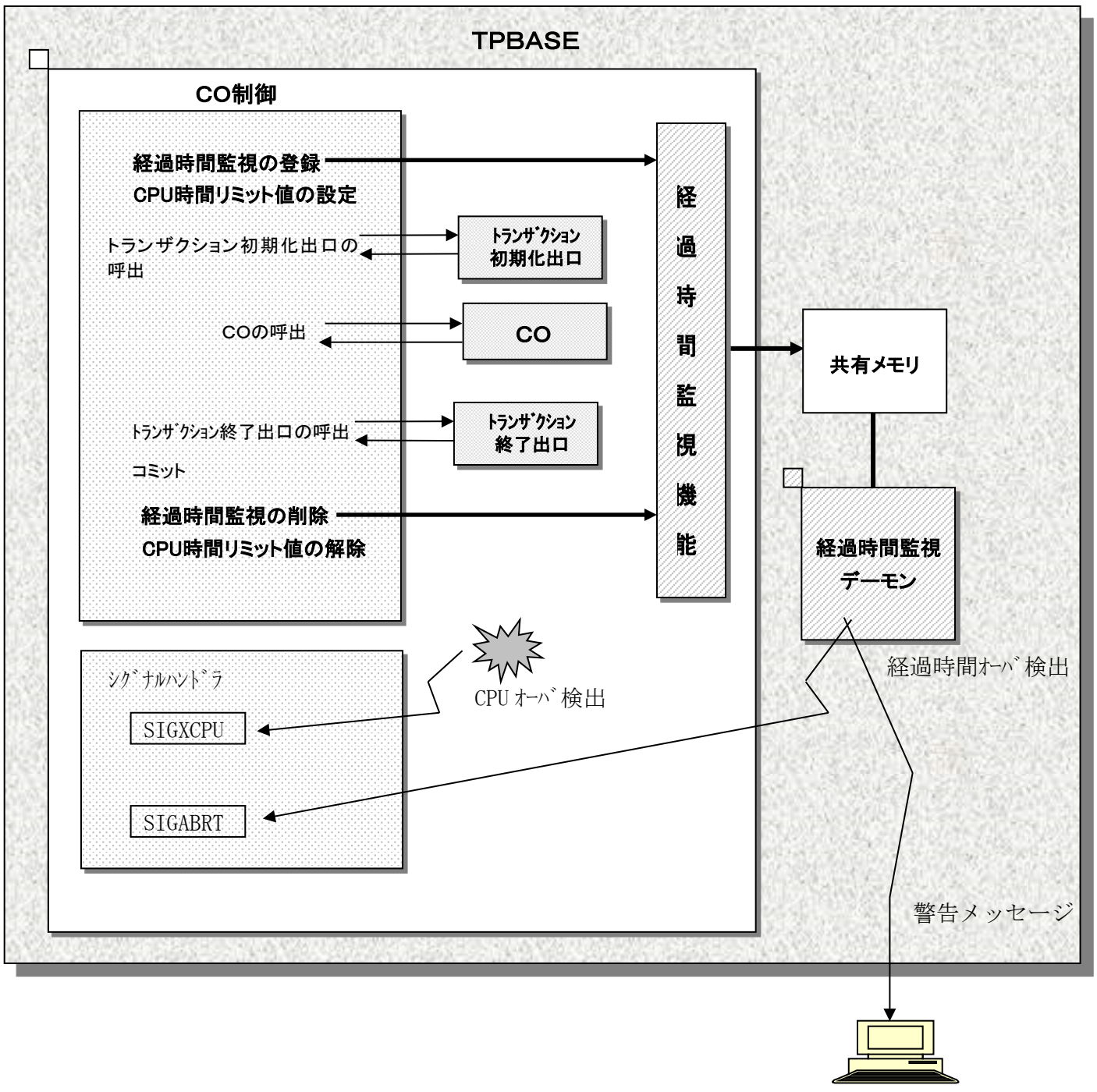

ＴＰＢＡＳＥ
ＣＯ制御ＴＰＰ
ＣＯ制御
ＣＯ
例外検出
シグナルハンドラ
アボート#1 出口
資料採取等
異常応答電文
強制送信

2.1.8　　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(dicocmod)を使い制限時間値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。

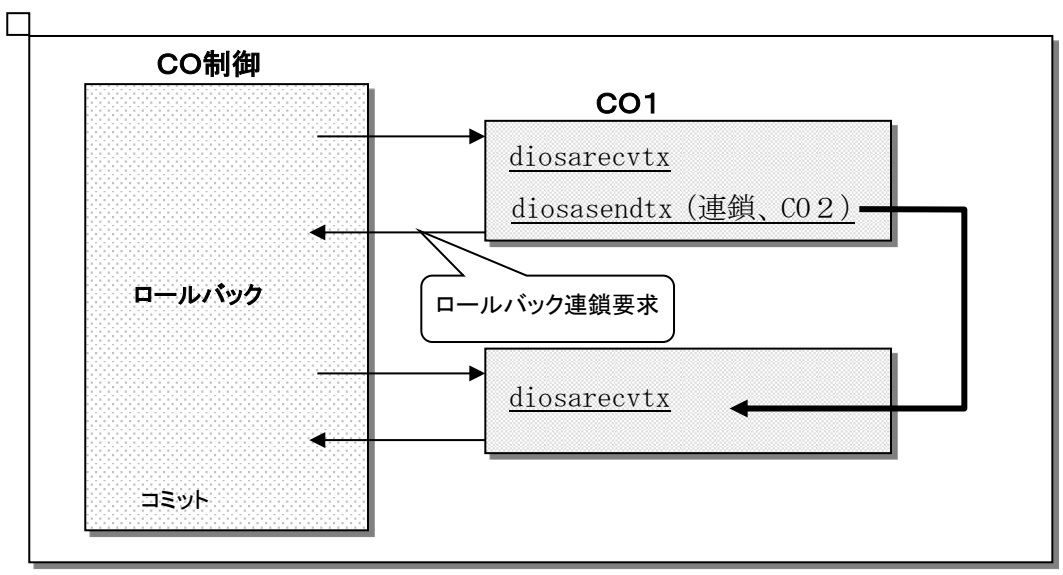


2.1.9　　ロールバック連鎖機能

AP から「ロールバック連鎖要求」が返された時、CO 制御はロールバックを行ってから連鎖 CO を呼び出します。AP は、送信 API により連鎖電文を登録しておく必要があります。CO 制御に戻る前に diosauca にロールバック連鎖要求を設定して CO を終了します。CO 制御がこの状態を参照してロールバック処理を行い、連鎖電文処理を継続します。

本機能は、今までの処理をキャンセルして新たに DB 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。

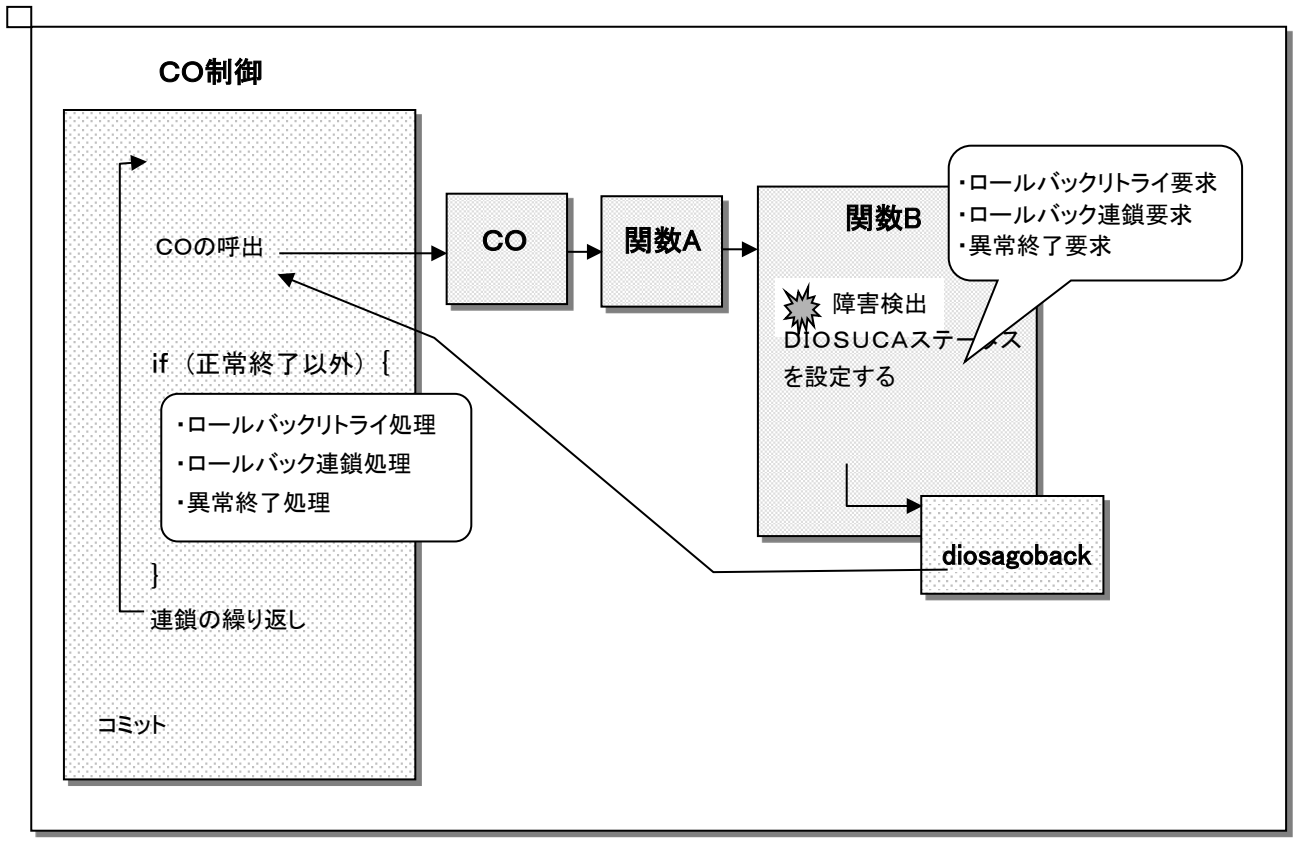
備考　ロールバック連鎖はアボート処理とは異なり、通常の連鎖処理と同様に正常終了扱いとなります。



2.1.10　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 に終了状態を設定しておきます)



2.1.11　ロールバックリトライ機能

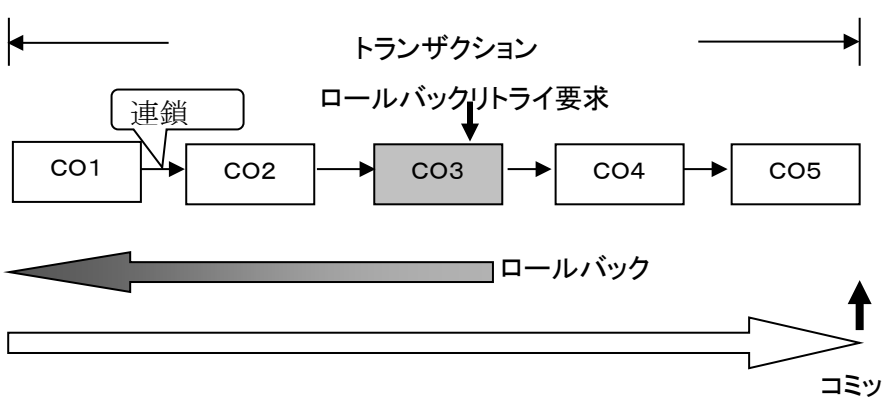
実行中処理を中断してロールバック後、処理を再開することができます。AP が業務処理で論理的エラーを検出した場合、diosauca の状態コード(Status)に「ロールバックリトライ要求」を設定して CO を終了します。CO 制御がこの状態コードを参照して、ロールバックリトライ処理を行います。

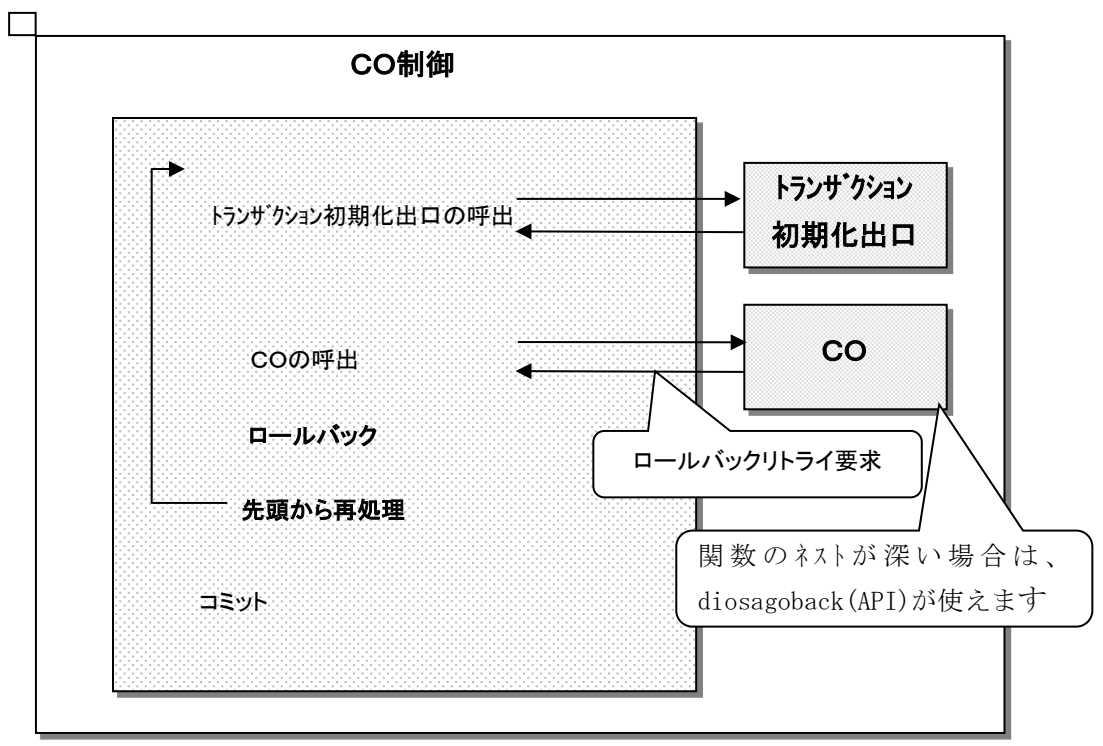
後述するデッドロックリトライも同様なリトライ処理を行います。

リトライ処理はロールバック実行後、トランザクション初期化出口、CO の順に呼び出しを再開します。リトライ処理では diosauca のトランザクション処理結果(ExitKey)に「ロールバックリトライ要求によるリトライ」か「デッドロックリトライ要求によるリトライ」かの識別子を渡しますので、トランザクション初期化出口、および CO は必要に応じてリトライ処理を切り分けることが可能です。

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。リトライ回数に関する情報は diosauca のトランザクション再実行回数上限値(RetryLim)、トランザクション再実行回数(RetryCount)で AP に通知されます。トランザクション再実行回数上限値(RetryLim)を超えた場合、アボート終了が呼び出されます。

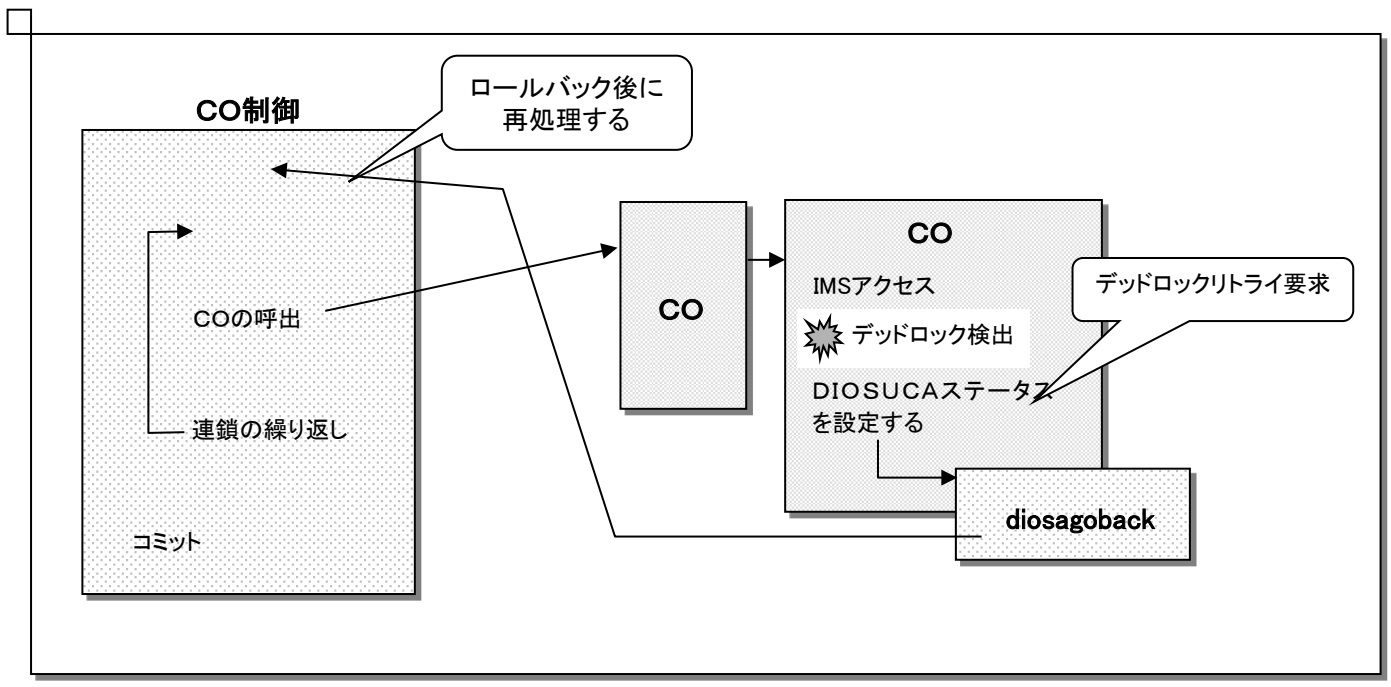
また、CO で diosacommit(API)によりトランザクション分割を行った場合、リトライ時に diosauca のコミット API(diosacommit)実行回数(CommitNum)を通知します。AP はこのコミット回数分処理をバイパスすることが可能です。





2.1.12　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。

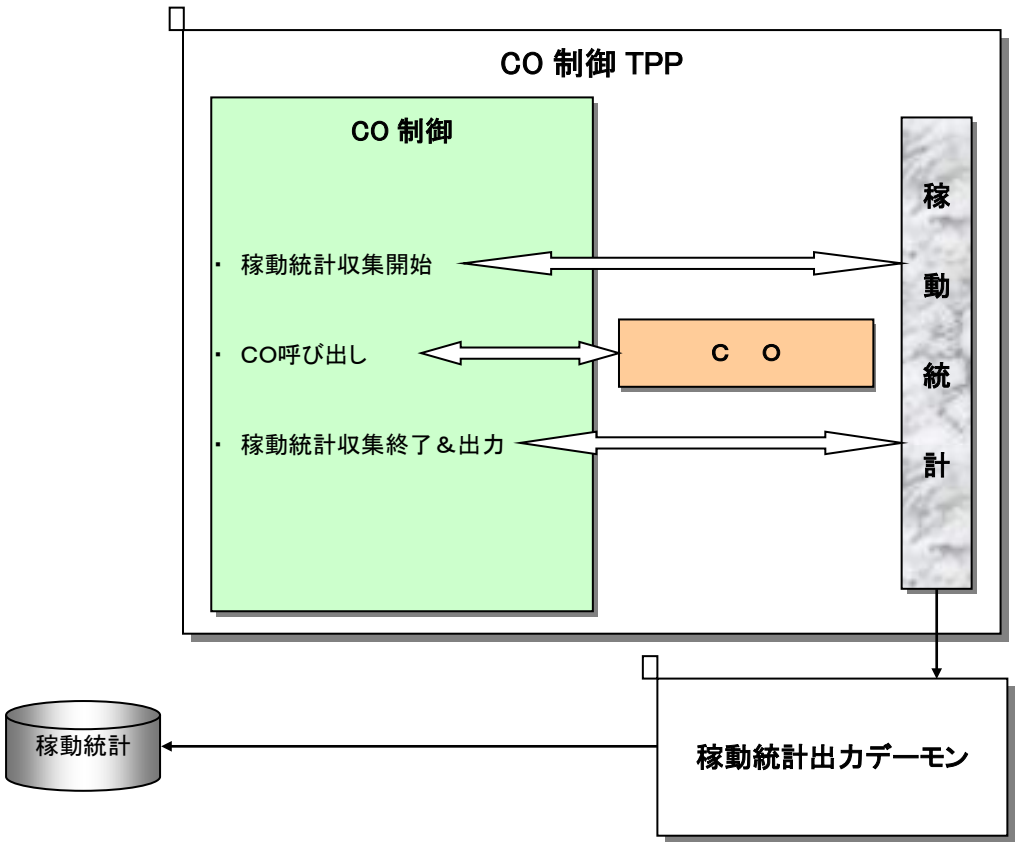


2.1.13　稼動統計情報収集機能

稼動統計機能と連携し、CO の稼動統計情報とトランザクション情報(コミット、ロールバックの実行毎情報)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 TPBASE モニタのトランザクション ID 毎に環境定義することができます。また、コマンド(dicocmod)を使い採取有無の変更をおこなうことができます。

詳細は、稼働統計機能を参照してください。



2.1.14　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。

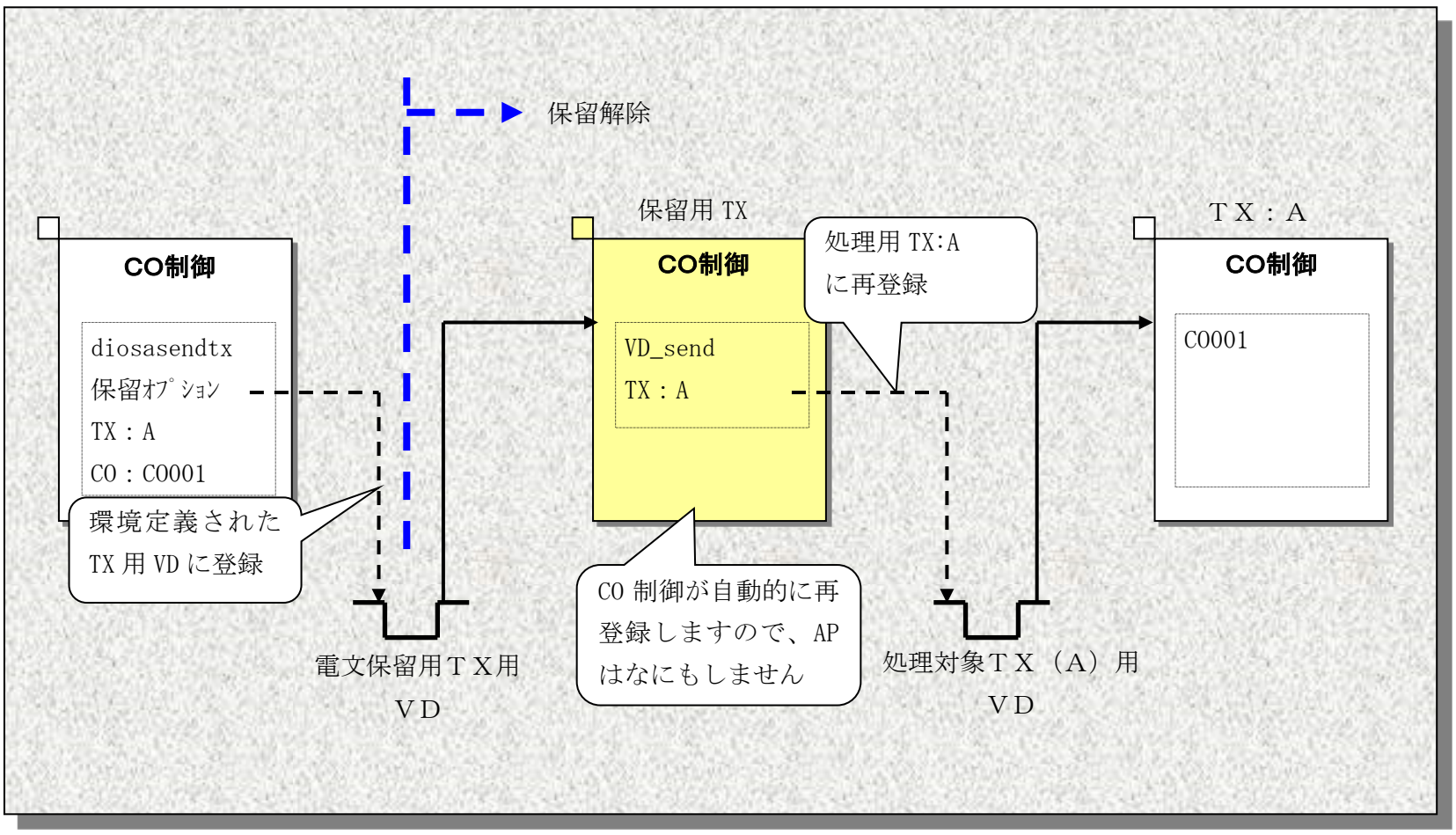
デッドロック、ロールバックリトライは直ちにリトライをしたい場合使う即時リトライ機能となり、電文保留機能はリトライ実行まで時間を要する場合に使う遅延リトライ機能となります。(以降の閉塞、閉塞解除という文言はTPBASE のトランザクション閉塞、閉塞解除の意味です)

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る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。



2.1.15　DB アクセス機能

利用者は更新したい DB(Oracle、IM)を環境定義することができます。TPBASE モニタのクラス毎に定義が可能です。

定義できる DB は以下となります。

① DB を使わない。

② IM のみを使う。

③ Oracle のみを使う。

④ IM、Oracle を使う。利用者が更新する DB を選択します。

上記定義のうち④を定義した場合は、利用者が更新 DB を決定する必要があります。更新 DB を決定する方法は２つ存在します。ひとつは電文送信時に決定する方法、もうひとつは電文受信時に受信電文解析出口で決定する方法です。電文送信時に更新 DB を IM としたいときは、電文送信情報に IM のメインキー情報、または MAPID 情報を指定します。更新 DB が Oracle の時は DB 情報を指定する必要はありません。受信電文解析出口で更新 DB を決定する場合は、出口に渡される構造体 t_diosa_analye を使います。詳細は API リファレンスマニュアルを参照してください。

「IM、Oracle を使う」場合、更新 DB はどちらかが選択されますが、選択されなかった DB の参照は自由です。Oracle の参照は利用者任意で行ってください。IM の参照は CO 制御が参照可能な状態とします。

IM と Oracle の両方を更新、または Oracle の複数インスタンスを更新する場合は、利用者の責任で行ってください。

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 AP 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。
- DB への接続、コミット制御。
- 利用者による初期化処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)　AP 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 CO/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、CO 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

CPU 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)　DB 自動制御機能

バッチ処理開始に DB(インメモリサーバまたは Oracle)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 CO を再度呼び出します。

2.2.2　　AP 呼び出し機能

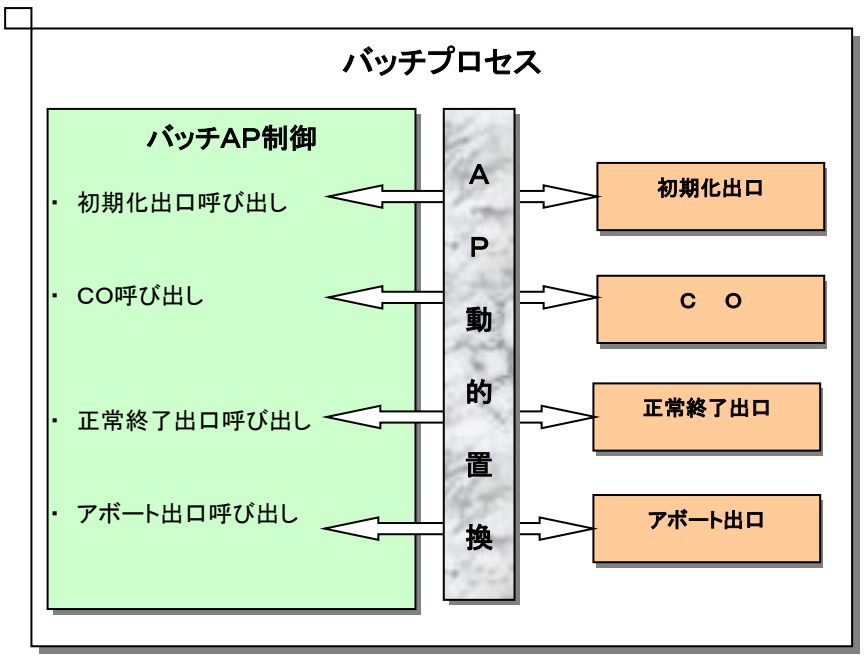
アプリケーション動的置換機能では関数名を解決して CO/利用者出口を動的に呼び出します。よって利用者はアプリケーションの物理的な配置を、アプリケーション動的置換機能の環境を設定する際に意識するだけでよく、バッチ AP 制御実行時にはパラメータに関数名を指定するだけで目的の CO/利用者出口を呼び出すことができます。

正常な流れでは「初期化出口→CO→正常終了出口」の順に呼び出しますが、CO/利用者出口から異常終了が返却された場合やバッチ AP 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 DB のロールバックは、バッチ AP 制御実行終了後に Oracle 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 AP 制御自身が明示的にロールバックを実行することはしません。SIGKILL 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。

CO/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。

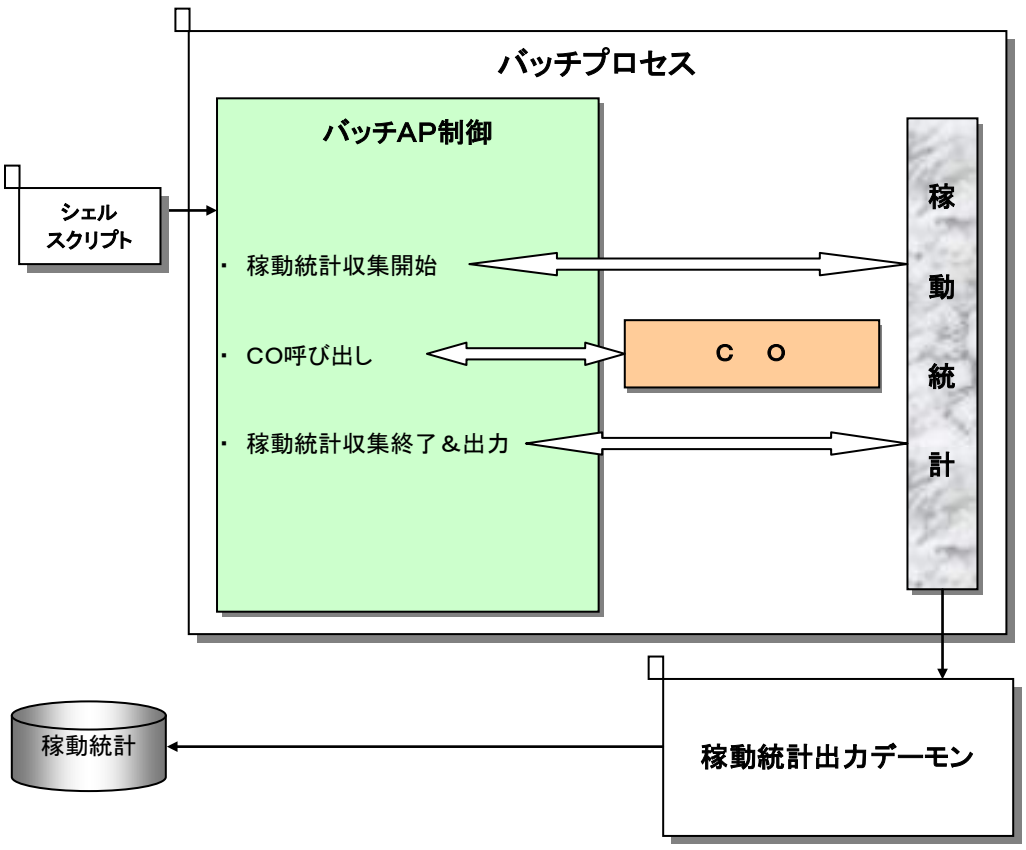
AP 呼び出し機能の概要図を以下に示します。



2.2.3　　稼動統計情報収集機能

稼動統計機能と連携し、CO 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。



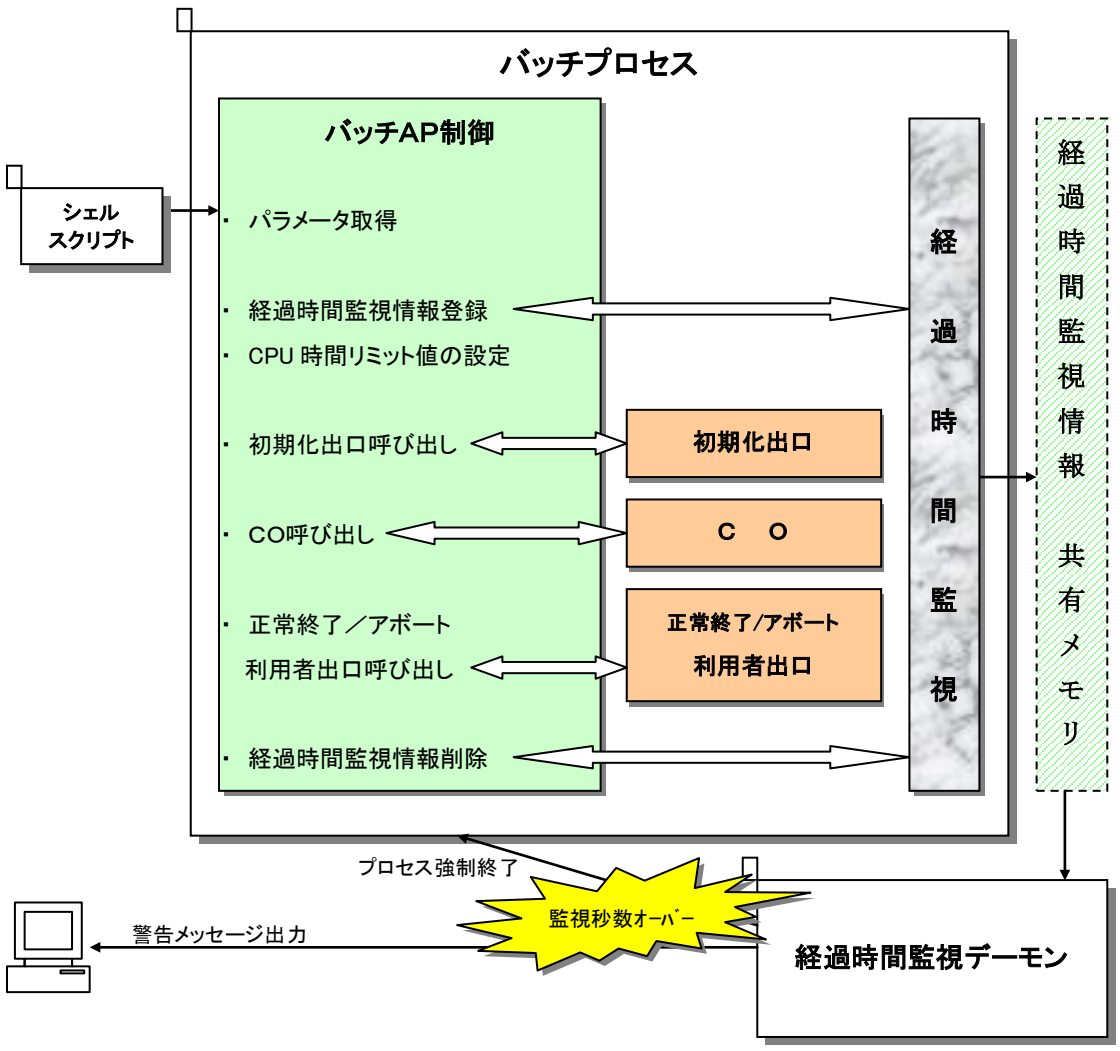
2.2.4　　ループ/ストール監視機能

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

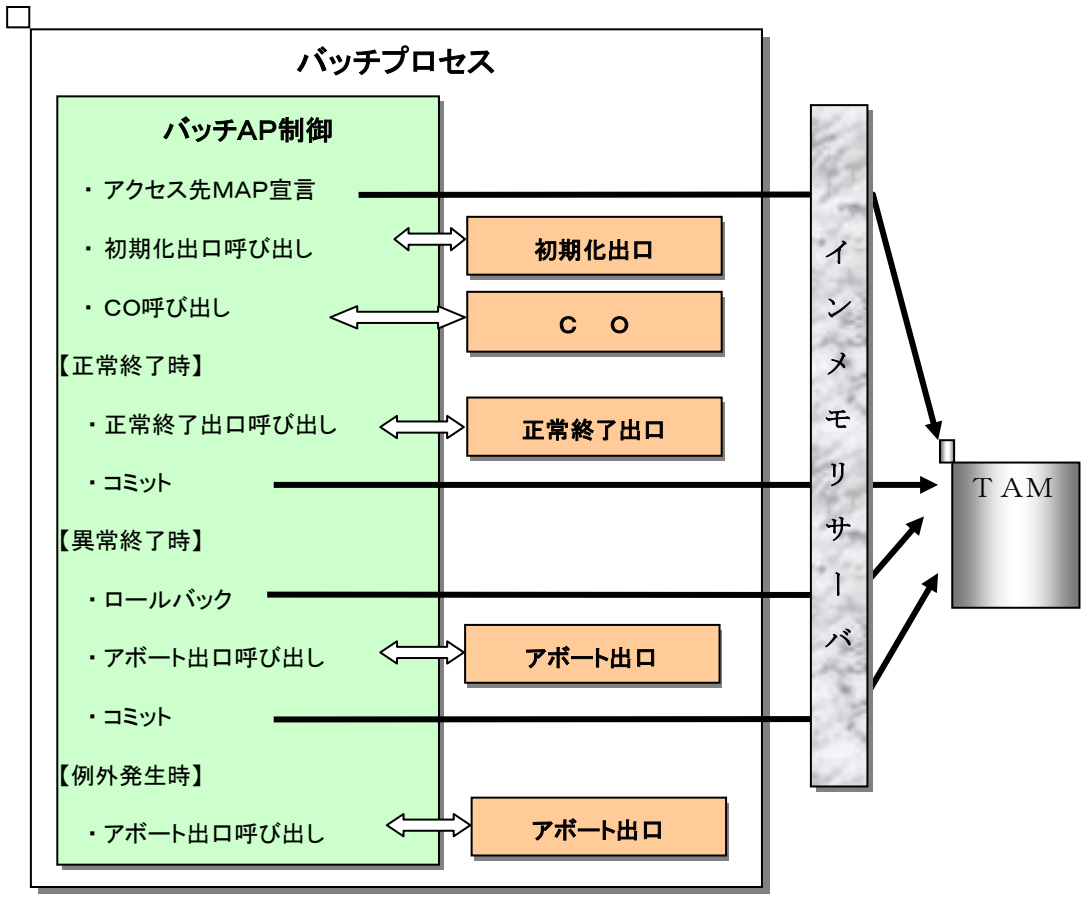
DB 自動制御機能には、TAM 自動制御と Oracle 自動制御の 2 種類を提供しています。

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。

(1) TAM 自動制御

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

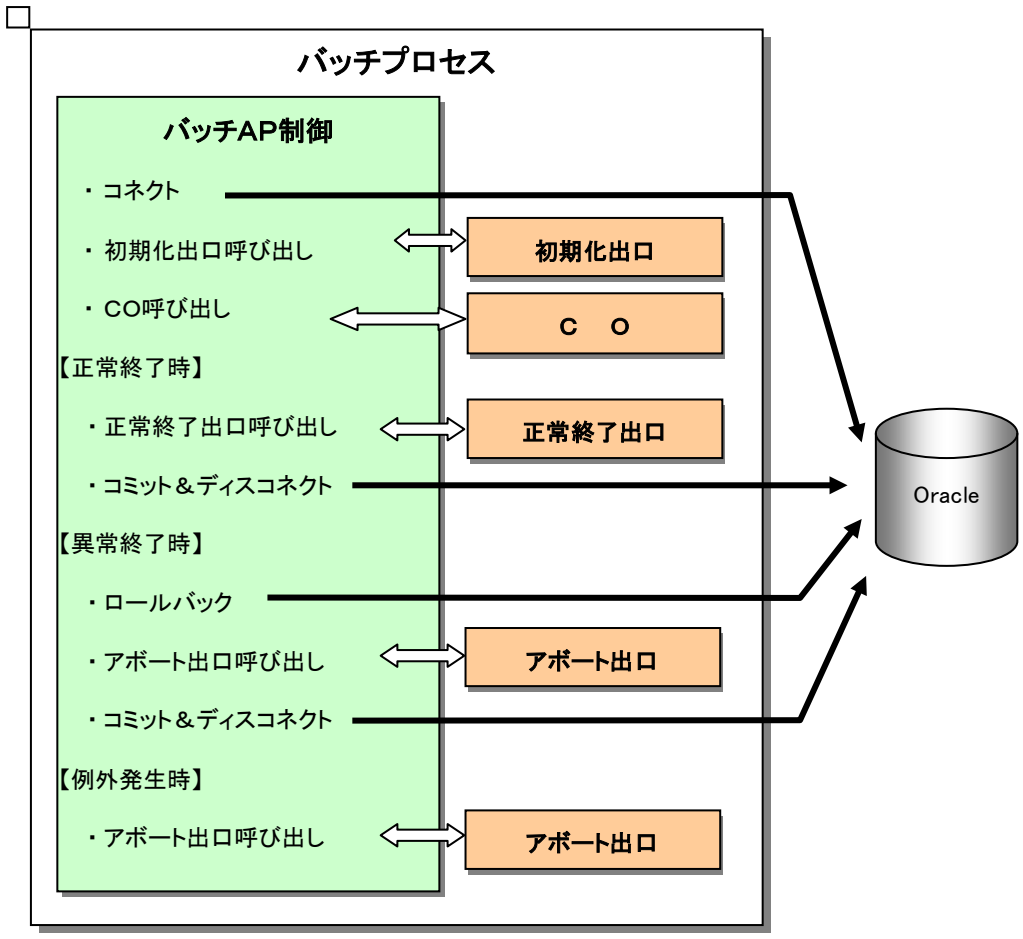
- プロセス起動時にインメモリサーバのオープンおよびアクセス先 MAP 宣言を行います。
- CO が正常終了時、コミットを行います。
- 正常終了出口の呼出し後にコミットを行います。
- CO/利用者出口からの要求による異常終了時には、アボート出口の呼び出し前にロールバックを行い、アボート出口の呼び出し後にコミットを行います。(アボート出口でも TAM アクセスが可能です。)
- シグナル発生時は例外処理としてメッセージ出力やアボート出口呼び出しを行いますが、明示的なロールバックを行いません。(バッチ AP 制御実行終了後にインメモリサーバにより自動的にロールバックされます。アボート出口で実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

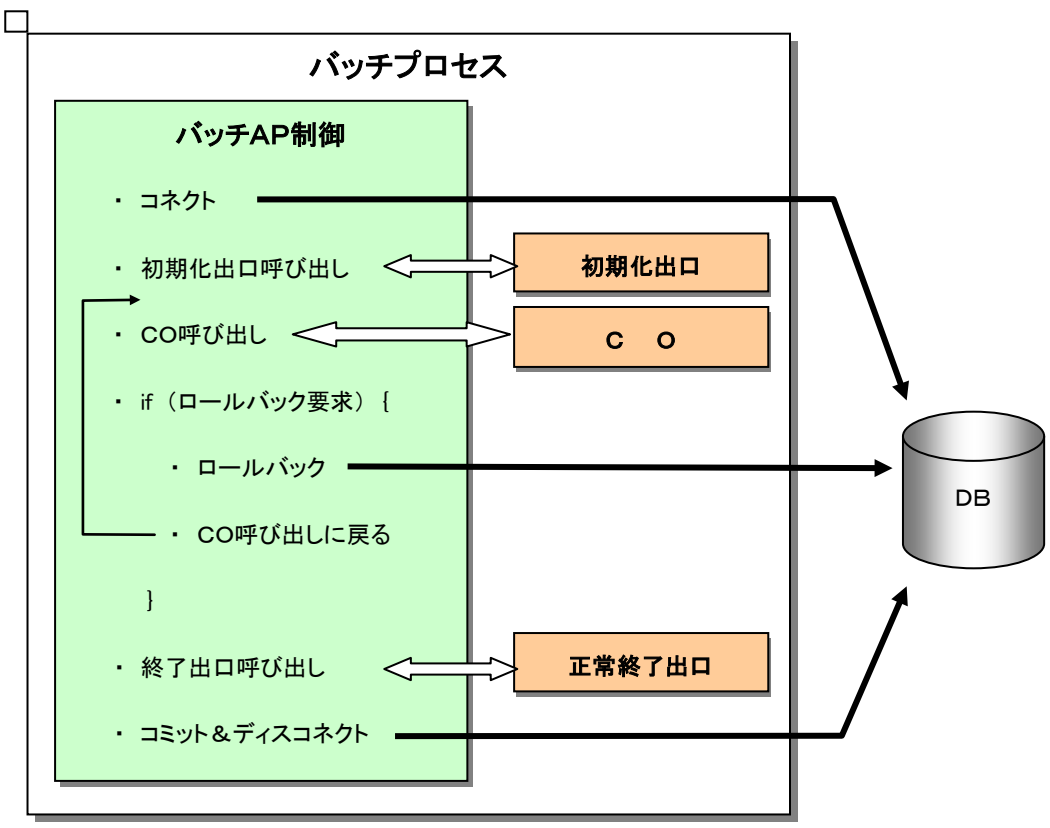
- プロセス起動時に Oracle へコネクトします。
- CO が正常終了時、コミットを行います。
- 正常終了出口の呼出し後にコミット＆ディスコネクトを行います。
- CO/利用者出口からの要求による異常終了時には、アボート出口の呼び出し前にロールバックを行い、アボート出口の呼び出し後にコミット＆ディスコネクトを行います。（アボート出口でも Oracle アクセスが可能です。）
- シグナル発生時は例外処理としてメッセージ出力やアボート出口呼び出しを行いますが、明示的なロールバックおよびディスコネクトを行いません。（バッチ AP 制御実行終了後に Oracle により自動的にロールバックさ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



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。

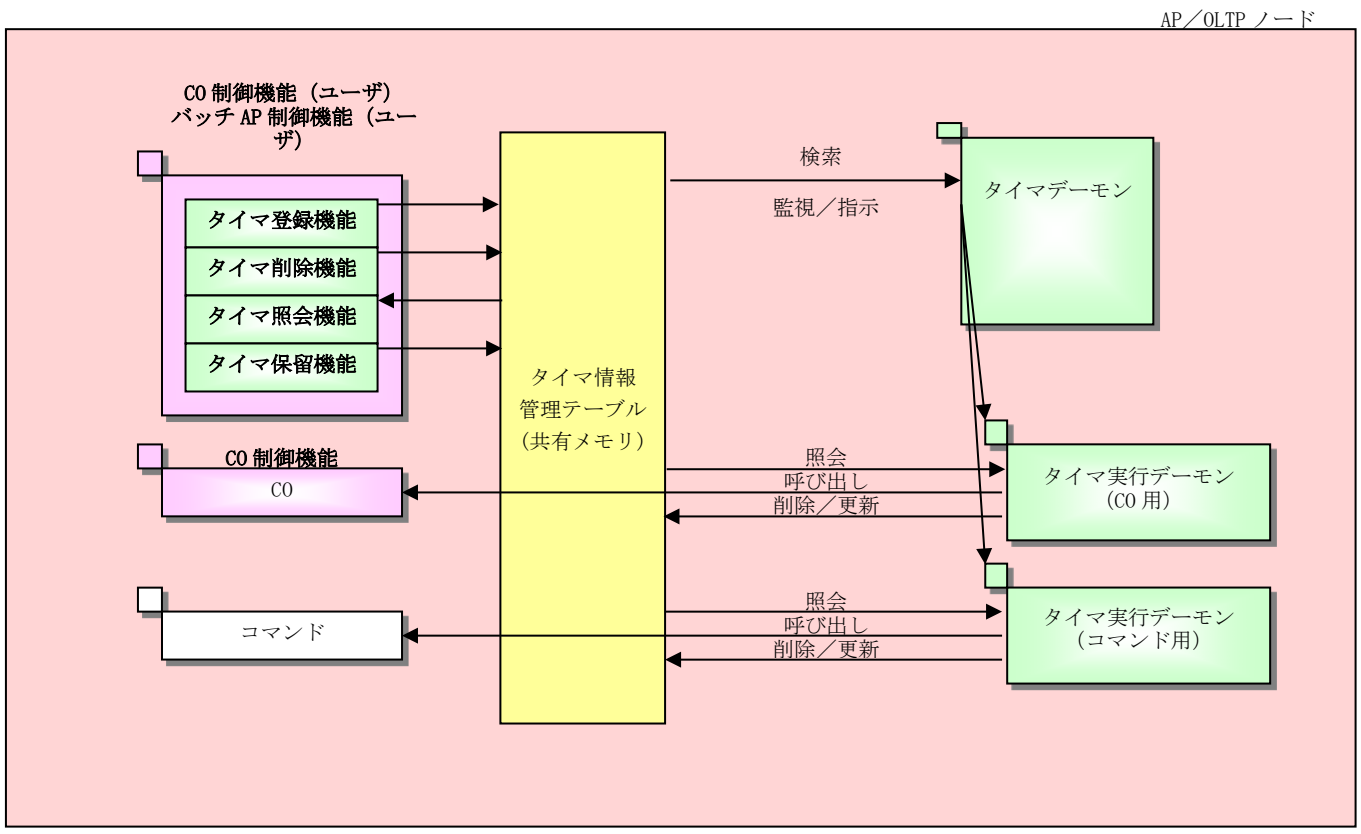


2.3　タイマ制御機能

タイマ制御機能では利用者任意のタイミングでCO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、CO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

- API、コマンド、SG からタイマ登録を行います。
- 登録時に指定するタイマ ID をキーにして管理します。

  同一タイマ ID によるタイマ登録をした場合は、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。API またはコマンドでタイマ ID 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。)
- タイマの最大登録数は環境変数にて指定します。

- 時刻形式によるタイマ実行動作

  タイマを実行する時刻形式として、インターバル指定、即時指定、時刻指定のいずれかが利用可能で、タイマ値として登録する時間情報(日付、時刻)、通知回数との組み合わせにより、タイマを実行する契機が選択可能です。

(a) インターバル指定

  インターバル指定は、タイマ値に設定した時間間隔(1 秒～23 時間 59 分 59 秒)で通知回数分(1 以上)のタイマを実行します。

(b) 即時指定

  即時指定は、インターバル指定と同じ動作を行いますが、タイマ登録した時点で 1 回目のタイマ実行を行います。2 回目以降は、インターバル指定と同じ扱いとなります。

(c) 時刻指定

  時刻指定は、タイマ値に設定した時刻にタイマを実行します。

  タイマ値に「日付(年月日)＋時刻(時分秒)」を指定した場合、指定した日時に 1 回だけタイマを実行します。この場合、登録時の日時より、過去の日時を登録することはできません。

  タイマ値に「時刻(時分秒)」のみを指定した場合、通知回数で指定した日数分、指定した時刻にタイマを実行します。なお、初回のタイマ実行日時は、「タイマ登録時刻＜タイマ値の時刻」の場合登録日当日となり、「タイマ登録時刻≧タイマ値の時刻」の場合登録日の翌日となります。

(2) タイマ削除機能

- コマンドまたは API 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、ワイルドカードが使用できます。

(3) タイマ照会機能

- コマンドまたは API 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 ID で指定し、ワイルドカードが使用できます。
- API による照会は、先頭からの順次検索とタイマ ID 指定のタイマエントリ照会が可能です。

(4)　　タイマ保留・保留解除機能

- コマンドまたはAPI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実行時刻を再算出します。

(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。
- 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、CO やユーザアプリケーションで動的に確保するデータ領域を一括で管理し、以下の機能を提供します。

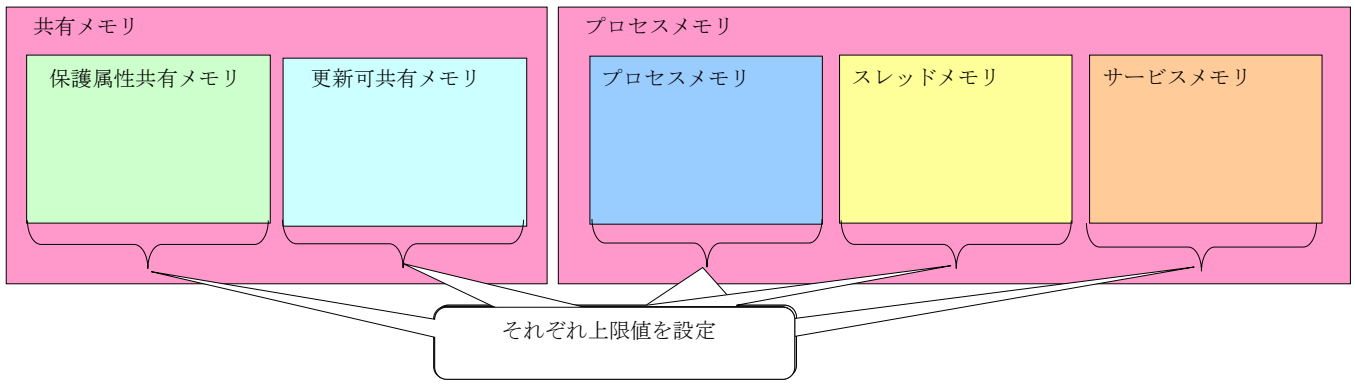
- 共有メモリおよびプロセスメモリの動的割り当て/解放をおこないます。
- 予めメモリ使用上限値を設定して、メモリの使い過ぎを防止することができます。
- CO 制御のトランザクション初期化時に前回のトランザクションで確保(解放漏れ)したプロセスメモリを自動解放することができます。
- 保護属性共有メモリを確保して、不正なメモリ更新はシグナルにより防止することができます。
- プロセスが異常終了した場合にメモリ情報をダンプファイルに出力し、出力したダンプファイルの内容を照会することができます。

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	ノード内共有	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	プロセス内共有	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
	スレッド内共有	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
	サービス内共有	一括解放対象	CO 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、CO 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、CO 制御やバッチ AP 制御から呼ばれる CO では、CO・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 API インタフェースを提供します。

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、ファイル型ロック制御機能、DB 型ロック制御機能を利用して制御をおこないます。

ロックの有効範囲とロック制御機能の対応を以下に示します。

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

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

・ウエイトオプション：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

- システム関数 fcntl()を利用してロック制御をおこないます。
- 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、ロック API は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

- データベースの排他制御支援パッケージ(DBMS_LOCK)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、ロック API はエラーを返却しません。
- DB 型ロック同士のデッドロック検出、アプリケーションによる DB 更新(表レコードのロック)との相互デッドロック検出ができます。
- 複数の DB インスタンスが存在する場合は、デフォルトリソースグループセットの DB インスタンスに対してロックを行います。

2.6　メッセージ出力機能

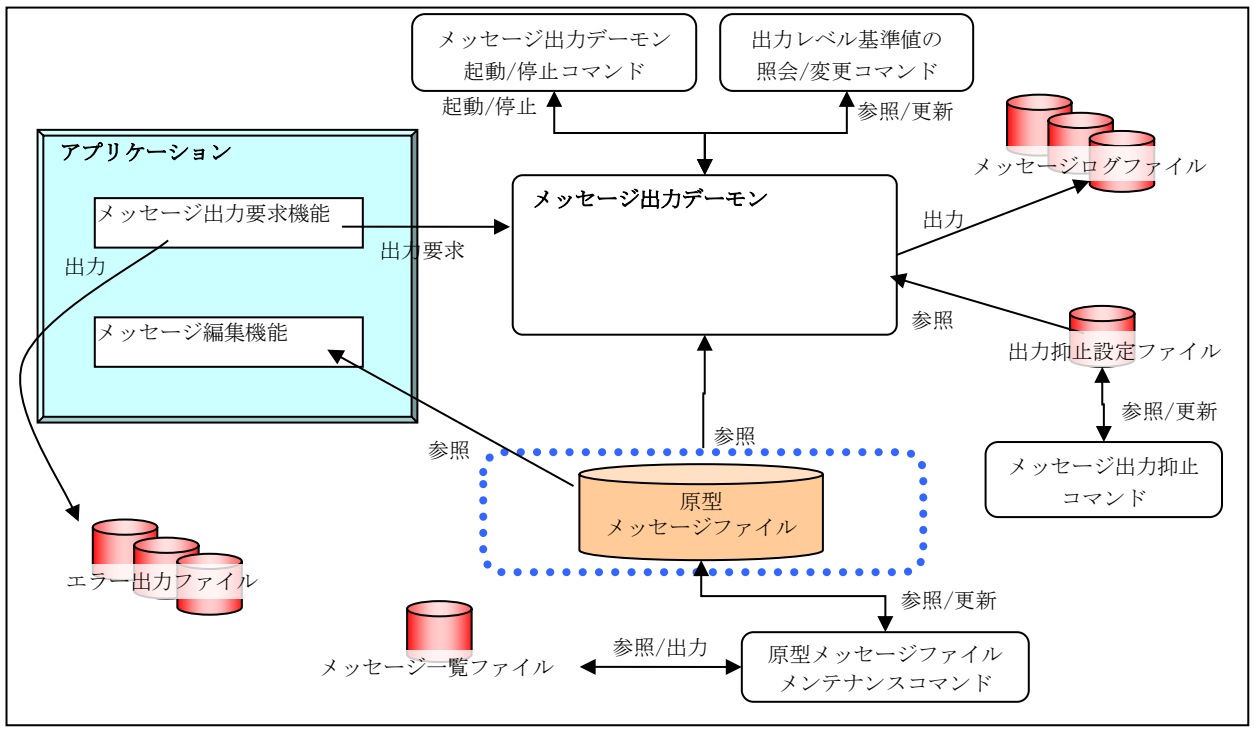
メッセージ出力機能は、ユーザアプリケーションがユーザ固有の重要メッセージをログファイルに出力するためのインタフェースを提供します。

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、メッセージごとにメッセージレベル(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(メッセージ出力レベル照会コマンド)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(メッセージ出力レベル変更コマンド)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、指定された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)

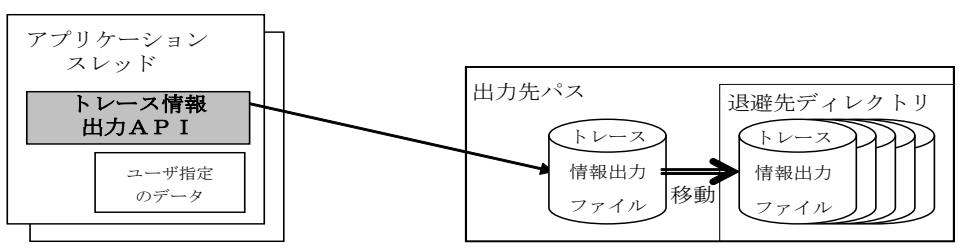
トレース情報をファイルへ直接出力する API を提供します。ただし、API 実行時に指定する出力レベルが環境設定で指定された出力レベルより大きい場合は、トレース出力は抑止されます。

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＞

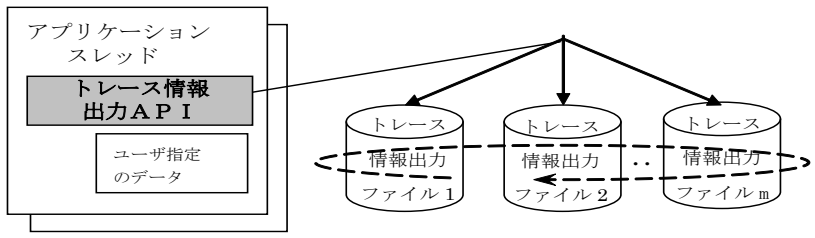
環境設定で指定されたサイズに達すると退避用ディレクトリにファイルを移動します。



＜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。

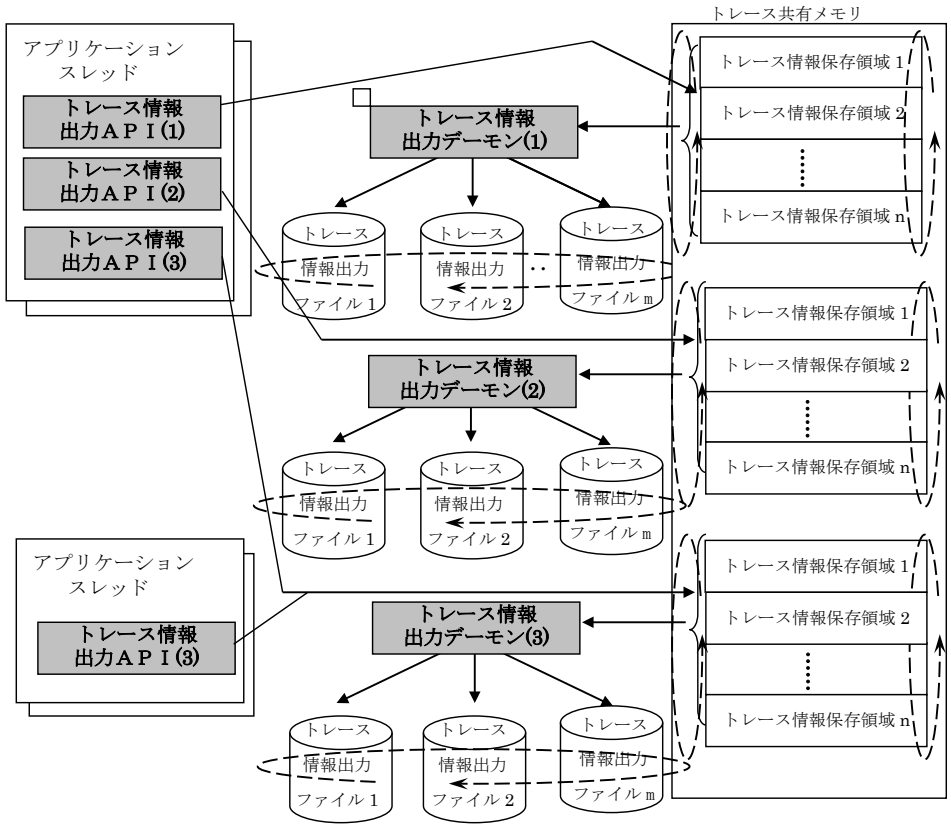
トレース情報を全てのプロセスから、同じ又は複数のトレースファイルに出力することが可能です。

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

修正前後の複数世代のライブラリを総称するための概念です。

置換コマンド等で指定するのも論理ライブラリ名です。

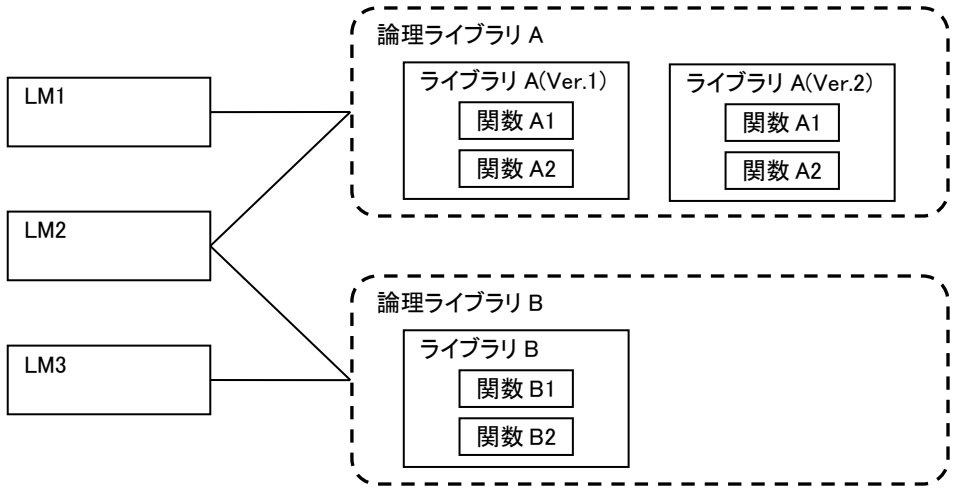
実ライブラリファイルと同じ名前で定義することも可能です。

(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

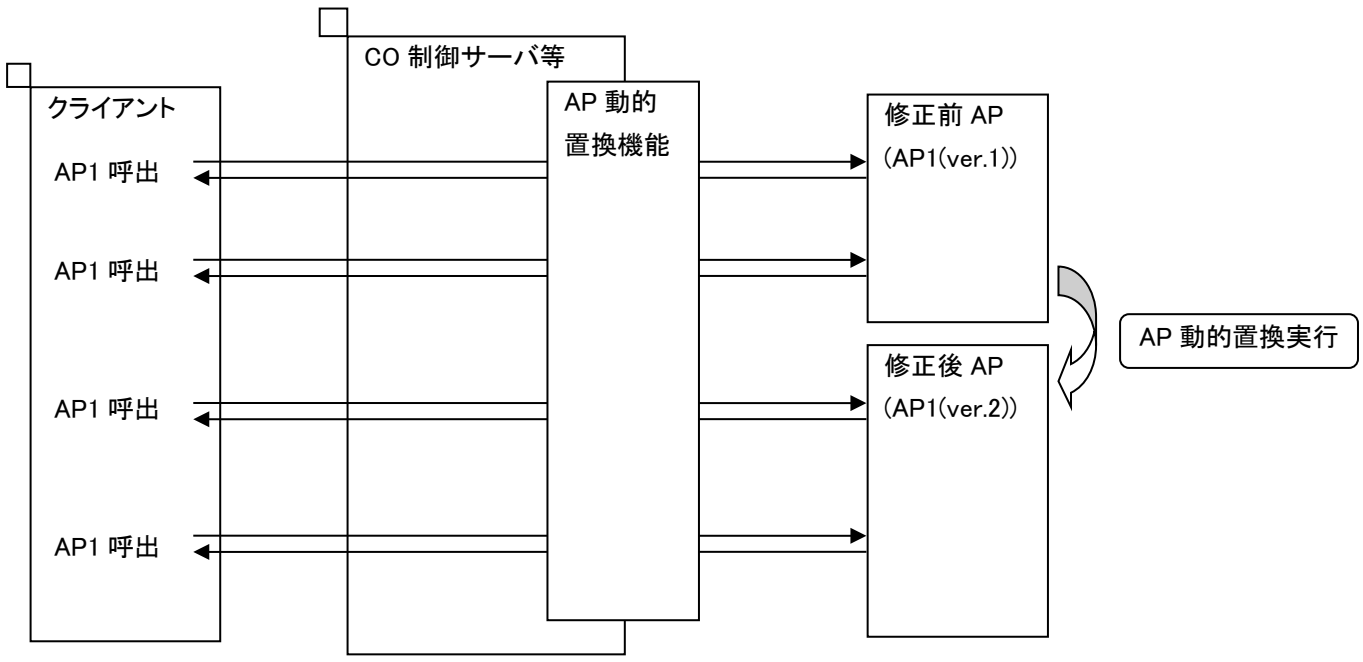
実際に呼び出される関数です。



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、DIOSA_LIBNAME_NODR 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。 CO 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)

(iii) 環境変数 DIOSA_LIBNAME_NODR に指定されたライブラリ

環境変数 DIOSA_LIBNAME_NODR に指定されたライブラリは、ライブラリ名内に%F を含まないものについては起動時にロードされます。%F を含むものについては、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)。

1 度ロードされると、AP 動的置換コマンドの実行や、環境変数 DIOSA_LIBUNLOAD の指定に関わらず、プロセス終了までアンロードされることはありません。

(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。

トランザクション開始からの経過時間が経過監視時間を超えたアプリケーションを検出した場合、対象アプリケーションの情報をメッセージ出力するか、対象アプリケーションに対してシグナル送信するか、いずれかの動作をおこないます。

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。

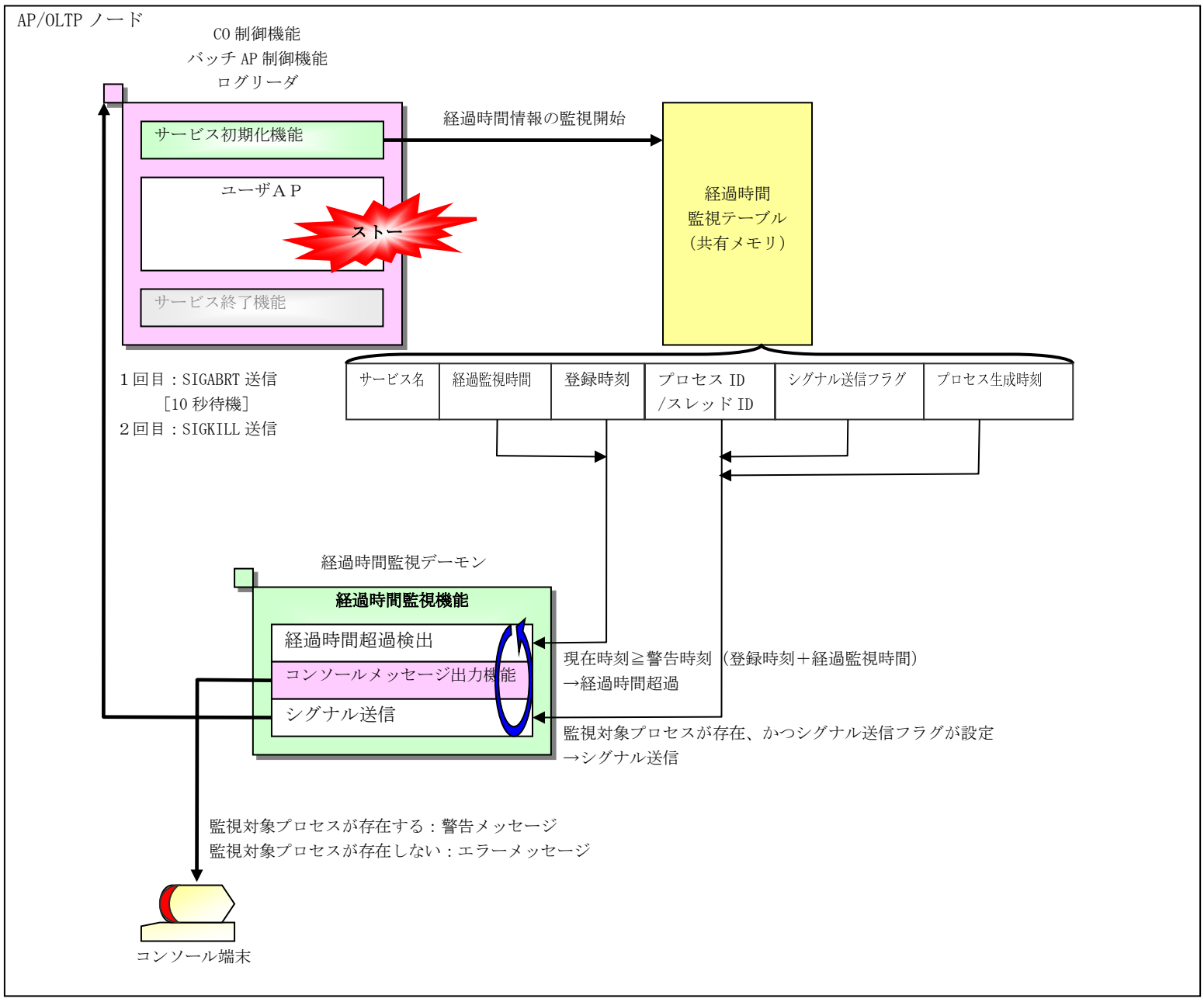
経過監視時間および検出時の動作については、CO 制御およびログリーダは環境定義(COCENV 節の ELPTIME パラメータ、DELAYED 節の ELPTIME パラメータ)、バッチ AP 制御は起動パラメータで指定できます。

(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

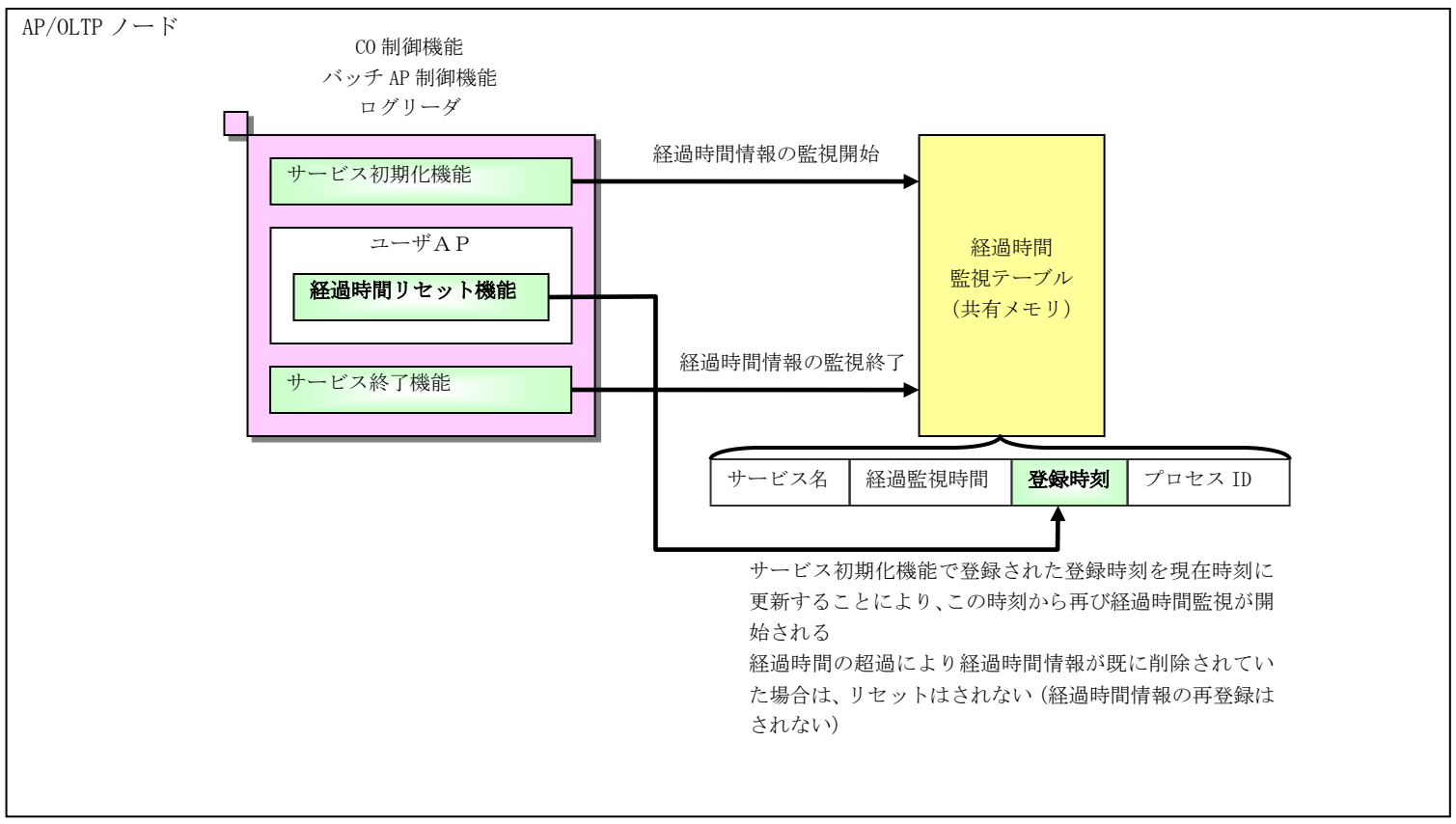
- 経過監視時間超過時に監視対象プロセスが存在し、シグナル送信フラグが設定されていた場合、監視対象プロセスを停止させます。
- 監視対象プロセスに SIGABRT シグナルを送信した後も監視対象プロセスが存在する場合、10 秒後に SIGKILL シグナルを送信します。


AP/OLTP ノード
CO 制御機能
バッチ AP 制御機能
ログリーダ
サービス初期化機能
ユーザＡＰ
ストー
サービス終了機能
経過時間情報の監視開始
経過時間
監視テーブル
（共有メモリ）
サービス名
経過監視時間
登録時刻
プロセス ID
/スレッド ID
シグナル送信フラグ
プロセス生成時刻
1回目：SIGABRT 送信
［10 秒待機］
2回目：SIGKILL 送信
経過時間監視デーモン
経過時間監視機能
経過時間超過検出
コンソールメッセージ出力機能
シグナル送信
現在時刻≧警告時刻（登録時刻＋経過監視時間）
→経過時間超過
監視対象プロセスが存在、かつシグナル送信フラグが設定
→シグナル送信
監視対象プロセスが存在する：警告メッセージ
監視対象プロセスが存在しない：エラーメッセージ
コンソール端末

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、経過時間監視デー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。



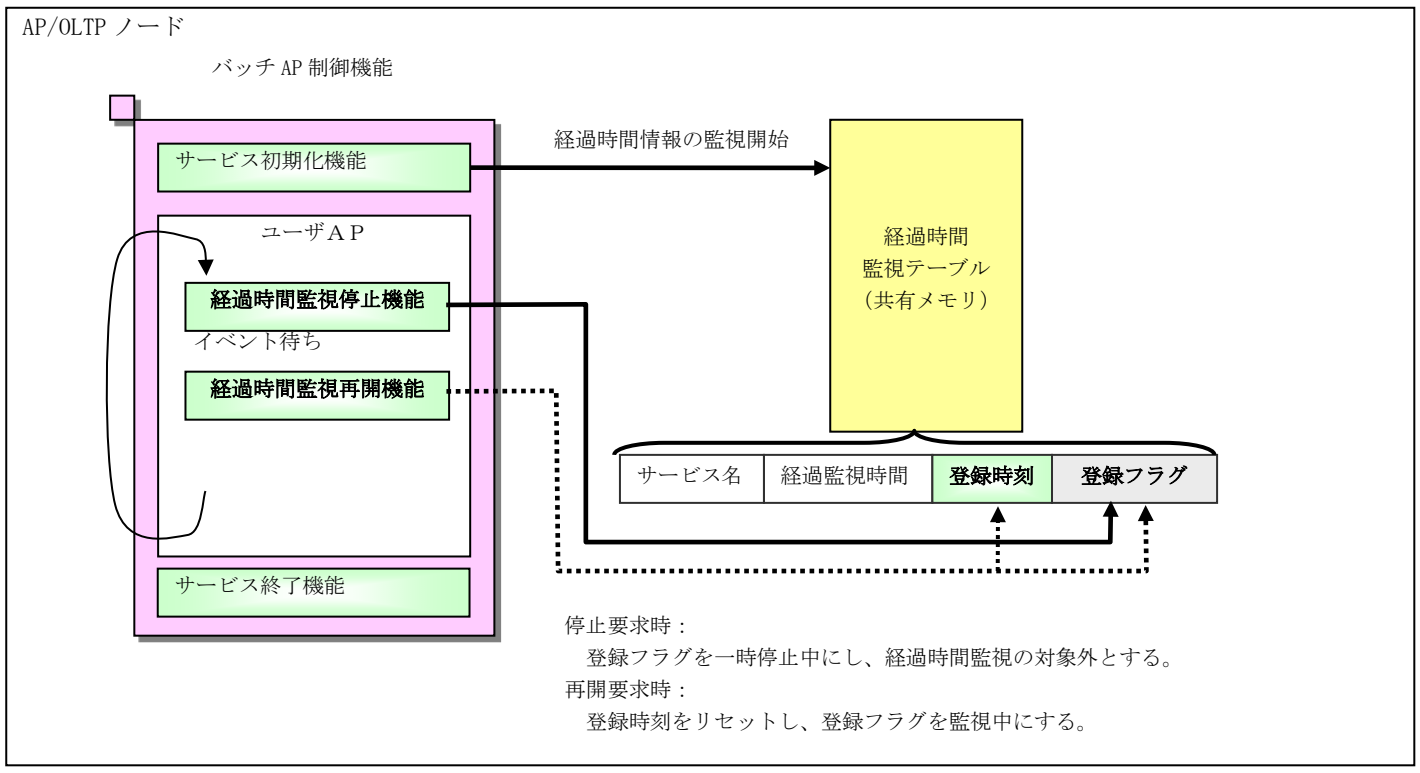
2.9.3　　経過時間監視停止/再開機能

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、CO の稼動情報を蓄積し、収集・編集する機能を提供します。CO 実行毎に開始時刻、経過時間、CPU 時間、インメモリサーバ API 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、各ノードで出力される稼動統計ファイルの内容を一括参照し、稼動情報の分析を容易に行う事を可能とするため、各ノードから稼動統計ファイルを収集する機能、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

CO 制御およびバッチアプリケーション制御と連携し、CO 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 UNIX 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、CO 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)　CO の稼動統計情報

CO 単位の稼動情報であり、1 つの CO 実行にかかる処理時間や CPU 時間を採取するのが主な目的です。CO 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 の関数名
- 電文開始時刻
- CO の開始時刻
- CO の処理時間(マイクロ秒単位)
- CO のシステム CPU 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

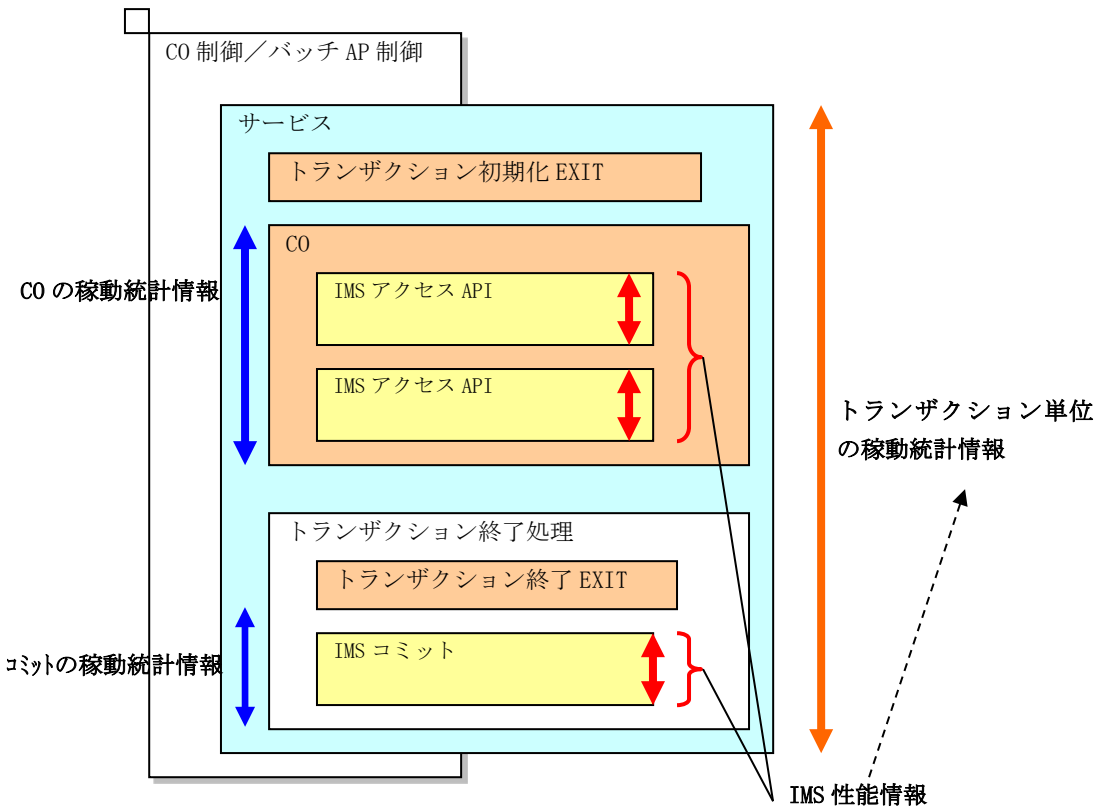
(d)　稼動統計情報の採取区間

CO の稼動統計情報は、1 つの CO 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)

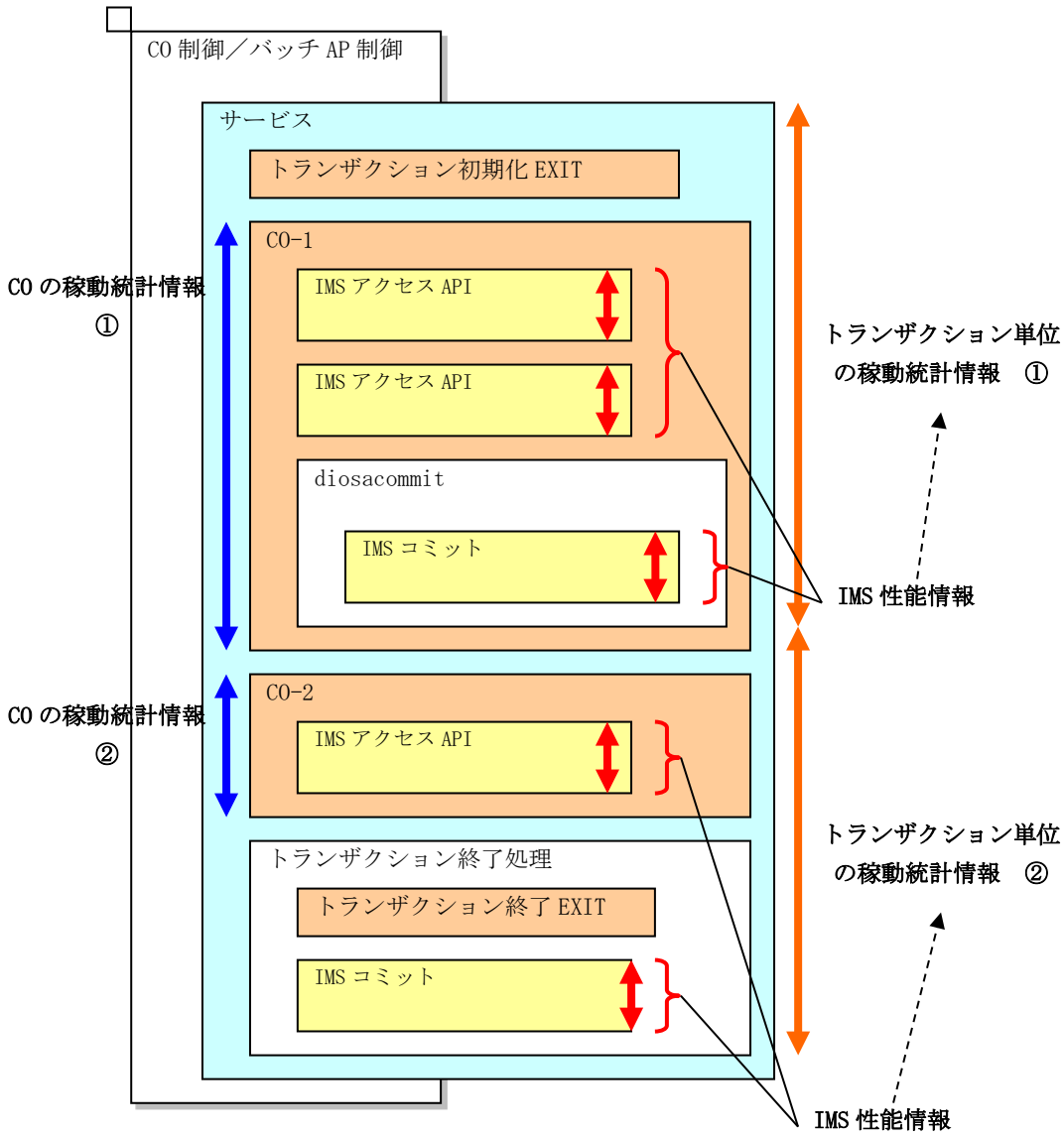
下記のような処理の場合、CO の稼動統計情報 1 件、コミット稼動統計情報 1 件とトランザクション単位の稼動統計情報が 1 件出力されます。

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。

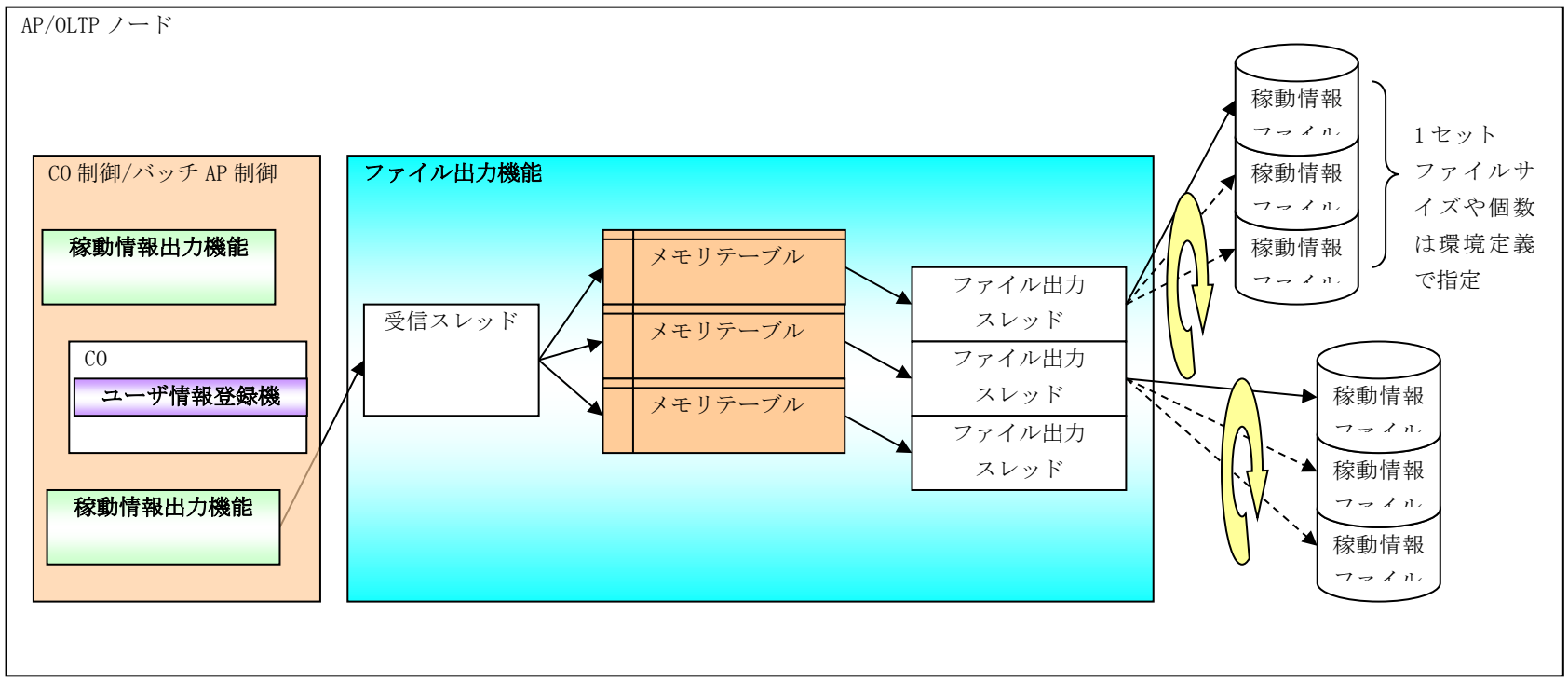
(3)　　ファイル出力機能

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。

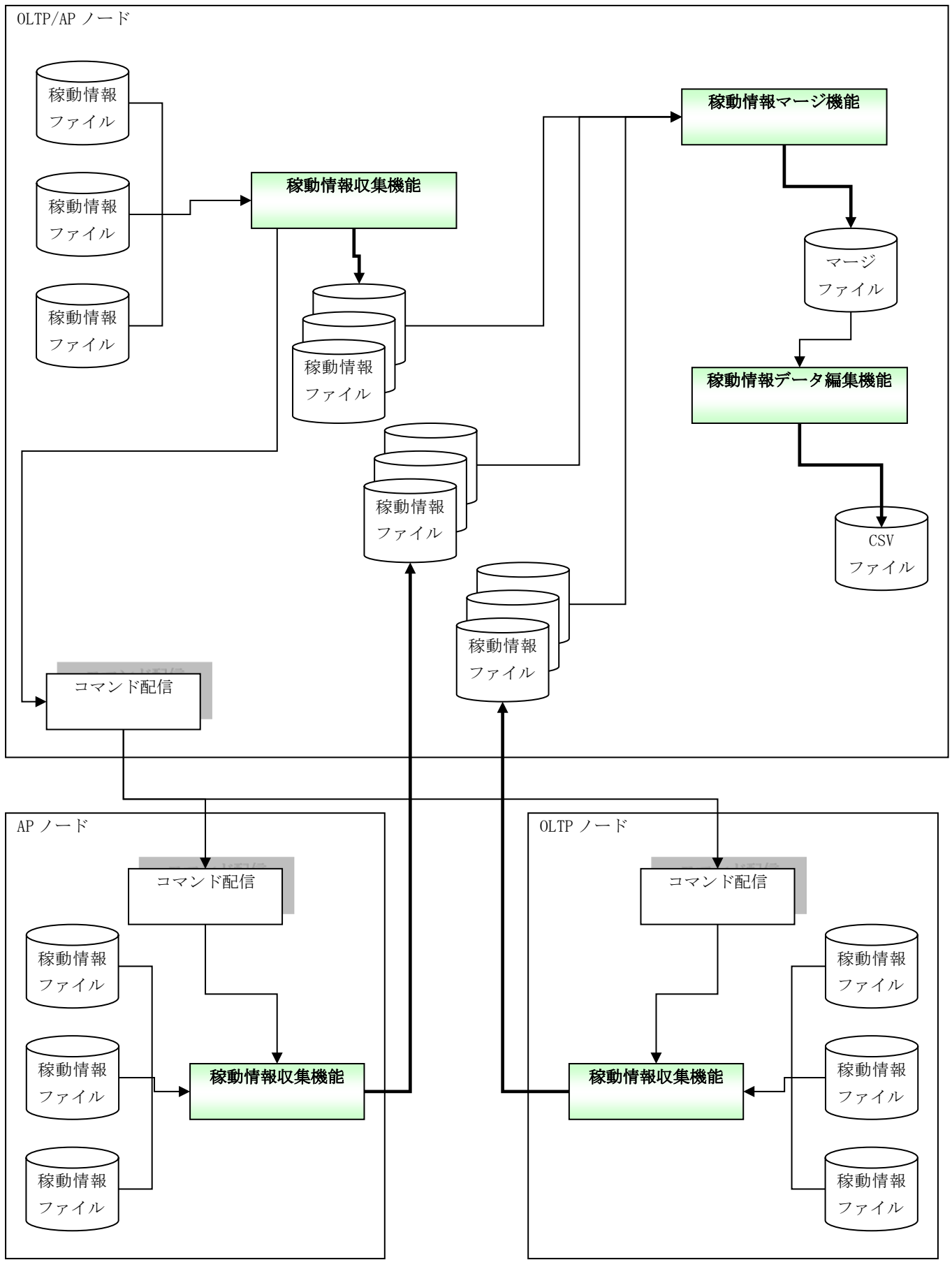
(2)　　稼動統計マージ機能

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、CSV 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、FTP が使用できる場合、手順②は不要となります。FTP 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 OLTP/AP ノードで稼動統計転送コマンドが呼び出され、FTP 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。

※FTP 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 OLTP/AP 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　論理ノード間通信管理機能

論理ノード間通信管理機能は、CO が送信 API(diosasendtx)を呼び出した際の、宛先となる同論理システム内の論理ノードへの電文送信可否を決定します。この時に用いられる論理的な通信パスのことを論理ノード間パスと呼び、電文送信可否は論理ノード間パスの活性・非活性によって決まります。

論理ノード間パスは、宛先となる論理ノードと TPBASE の組み合わせで表現され、1 つ以上の端末で構成されます。論理ノード間パスで結ばれる両 TPBASE は、互いに宛先となるリスナを端末として定義する必要があります(片方向のみの論理ノード間パスは構築できません)。論理ノード間パスは、それを構成する端末のうち 1 つ以上が接続済の状態であれば、活性とみなされます。ただし、論理ノード間パスが活性状態であっても、宛先論理ノードが閉塞・予閉塞状態の場合は電文を送信できません。

2.11.1　端末状態照会機能

状態照会コマンド(dinodepathref)の実行や、CO が状態照会 API(diosagetnodepathstatus)を呼び出すことで、自論理ノードが関係する論理ノード間パスの活性・非活性状態や、論理ノード間パスを構成する端末の状態を照会できます。

2.11.2　端末統計情報照会機能

統計情報照会コマンド(dinodepathrefstats)を使って、自論理ノードが関係する論理ノード間パス、あるいはそれを構成する端末を通じて送信された電文の数と滞留している電文の数を統計情報として照会できます。

2.11.3　端末制御機能

端末制御コマンド(dinodepathctrl)を使い、端末の接続や切断を行えます。対象の端末を名前で指定する以外にも、論理ノード間パスを構成する端末をまとめて指定したり、自論理ノードが関係する論理ノード間パスを構成する端末全てをまとめて指定できます。

2.11.4　端末状態監視機能

端末制御コマンド(dinodepathctrl)の実行やネットワークの障害等を認識し、それによる端末の状態(接続済/切断済/障害発生)決定や復旧のための接続コマンド実行を行います。

端末の接続は、以下の要因で発生します。

- 端末制御コマンド(接続指示)の実行
- 相手 TPBASE 側(端末側)からの接続
- TPBASE 起動時の端末の自動接続設定

端末の切断は、以下の要因で発生します。

- 端末制御コマンド(切断指示)の実行
- 相手 TPBASE 側(端末側)からの切断

端末の障害は、以下の要因で発生します。

- リスナプロセスの停止
- リスナプロセス障害に伴う電文送信失敗
- 接続先の不在(相手ホスト障害や定義ミスを含む)

端末の再接続のための接続コマンド実行は、ネットワーク障害の検出により発生します。復旧できない場合(TPBASEの端末再接続試行回数指定値の超過時)は、端末は障害状態とみなされます。

ネットワーク障害はTCPキープアライブやTPBASEのアライブチェック機能によって検出されます。

2.12　論理システム間通信管理機能

論理システム間通信機能は、AP ノード、OLTP ノードにおいて他論理システムとの通信を可能にします。

IP アドレスとポート番号の組をアクセスポイントと呼び、一つの論理システム内に複数のアクセスポイントを使用可能とします。

電文の送受信はアクセスポイントを指定して行います。

2.12.1　　常時接続

常時接続は TPBASE の機能を利用し、常にパスが接続された状態で電文の送受信を行います。

常時接続のアクセスポイントには 1 パスのアクセスポイントと 2 パスのアクセスポイントがあり、1 パスのアクセスポイントでは同一の端末で電文の送信と受信の両方が行われ、2 パスのアクセスポイントでは送信用端末と受信用端末を分けて電文の送受信を行います。

常時接続では下位プロトコルに TCP/IP、OLF/TP-UT、MQ をサポートします。

(1)　　パス接続切断機能

論理システム間の常時接続で使用するパスの接続・切断を行うコマンドと API を提供します。

接続・切断はアクセスポイント単位あるいはアクセスポイントを構成する端末単位で行うことができ、アクセスポイント単位での接続・切断は、アクセスポイントを構成する端末すべてが対象となります。

(2)　　パス状態監視機能

常時接続の端末が相手論理システム側で切断要求が発生した場合や、障害により切断された場合、端末が切断されたことを検知します。

アクセスポイントを構成する全ての端末が切断された場合、アクセスポイントを切断状態とし、電文の送信を抑止します。

(3)　　パス接続切断通知機能

論理システム（アクセスポイント）との常時接続のパスが接続または切断された場合、環境定義で指定された CO(パス接続切断同期通知 CO)を呼び出し、端末の状態が変化したことを伝える機能を提供します。

(4)　　パス自動再接続機能

常時接続の端末が障害により切断された場合、自動的に端末再接続を試行する機能を提供します。

試行のリトライ回数と間隔は TPBASE の端末定義ファイルの AUTCNT と AUTTIME で指定します。

パス自動再接続機能により再接続に成功した場合、通信経路の瞬断と判断し、ユーザへの通知は行われません。

再接続がリトライオーバーした場合、端末は切断状態となります。

2.12.2　　都度接続

都度接続は都度接続デーモンにより、送受信のたびにパスの接続を行い電文の送受信を行います。電文の送受信が完了するとパスは自動的に切断されます。

都度接続のアクセスポイントでは電文送信要求が発生した場合、自動的にパスの接続が行われるため、事前にパスの接続要求を行う必要がありません。

(1)　応答電文受信機能

応答電文受信機能は、電文送信 API で応答あり指定をした場合は、宛先アクセスポイントへ電文を送信後も接続状態を保持し、応答電文が返信されるのを待ちます。応答電文を受信した場合、CO 制御 TPP へ電文を送信します。

(2)　送受信エラー通知機能

送受信エラー通知機能は、宛先アクセスポイントとの電文送受信でエラーが発生した場合に、環境定義で指定された出口(送受信エラー出口)にエラー発生を通知します。

2.12.3　電文種別判定出口

論理システム間通信管理機能では複数の電文プロトコルに対応するため、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。

アクセスポイント毎にプロトコル種別を定義することができ、プロトコル毎に異なる電文種別決定出口を呼び分けることができます。

2.12.4　状態照会機能

アクセスポイントの状態を照会するコマンドを提供します。

2.12.5　統計情報照会機能

アクセスポイントの統計情報を照会するコマンドを提供します。

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。

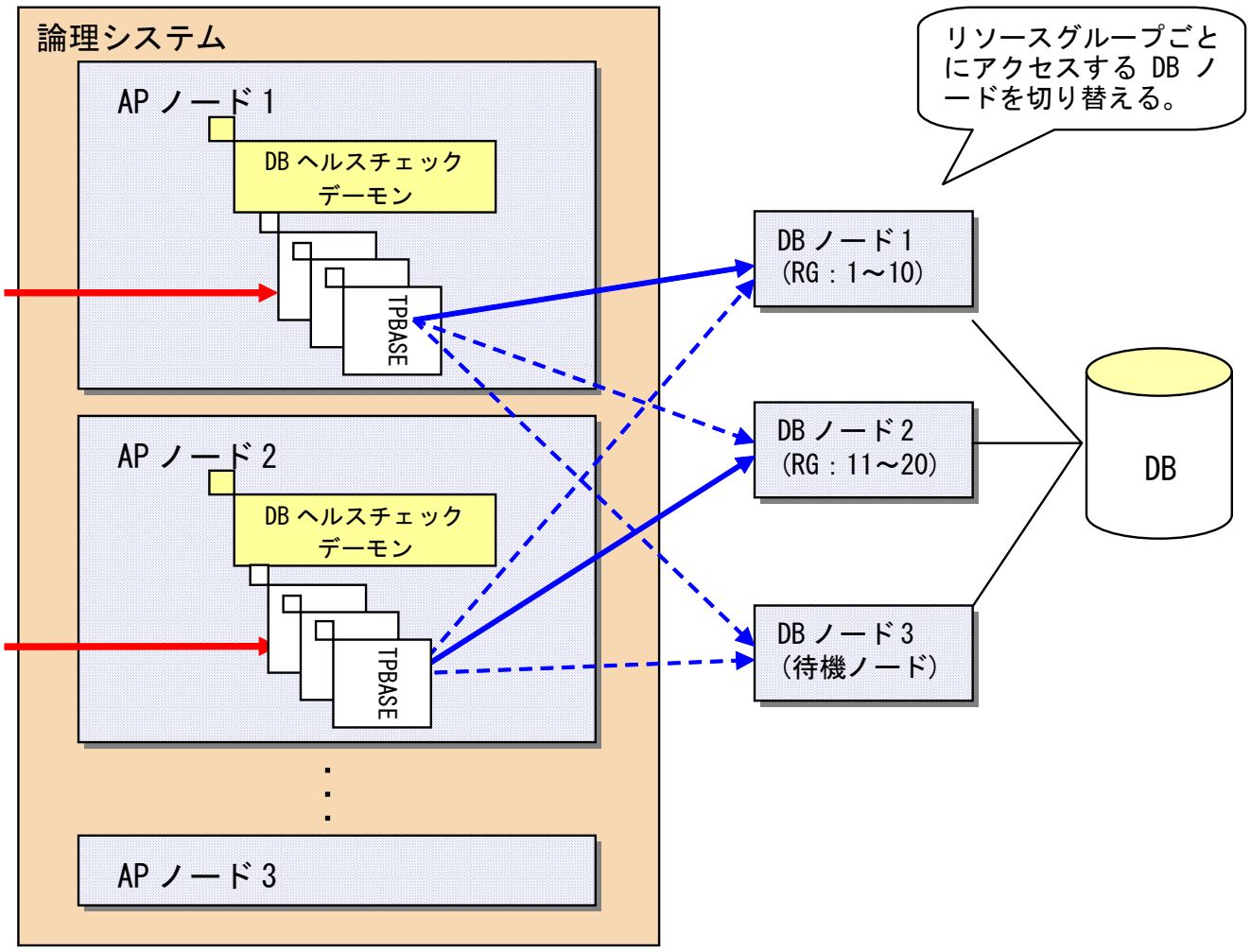


図 2-1 DB マルチコネクション概要

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情報は、環境定義で定義します。

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(キャシュフュージョン)が多発し、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、本機能が稼動中の 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。

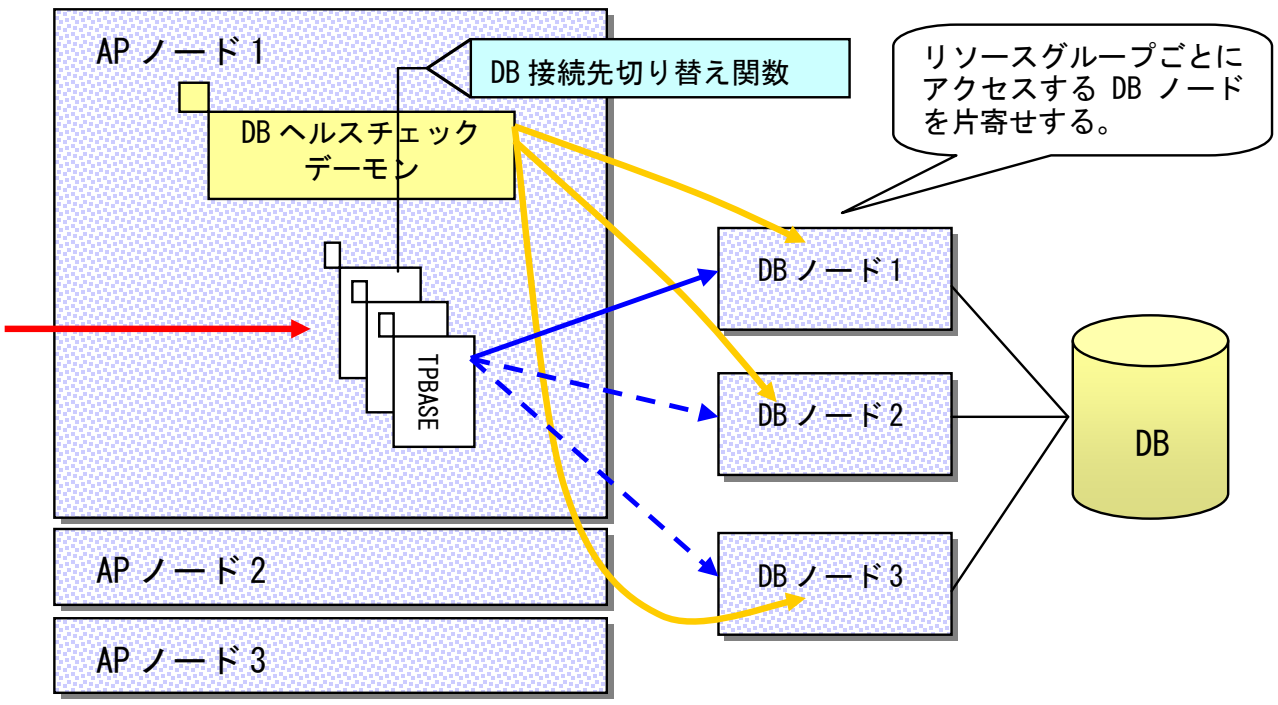


図 2-2 Oracle の DB インスタンス振り分け機能

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動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」を設定しておくと、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、DB ノードを閉塞するときに実行する閉塞状態変更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 Oracle 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 2-3 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。

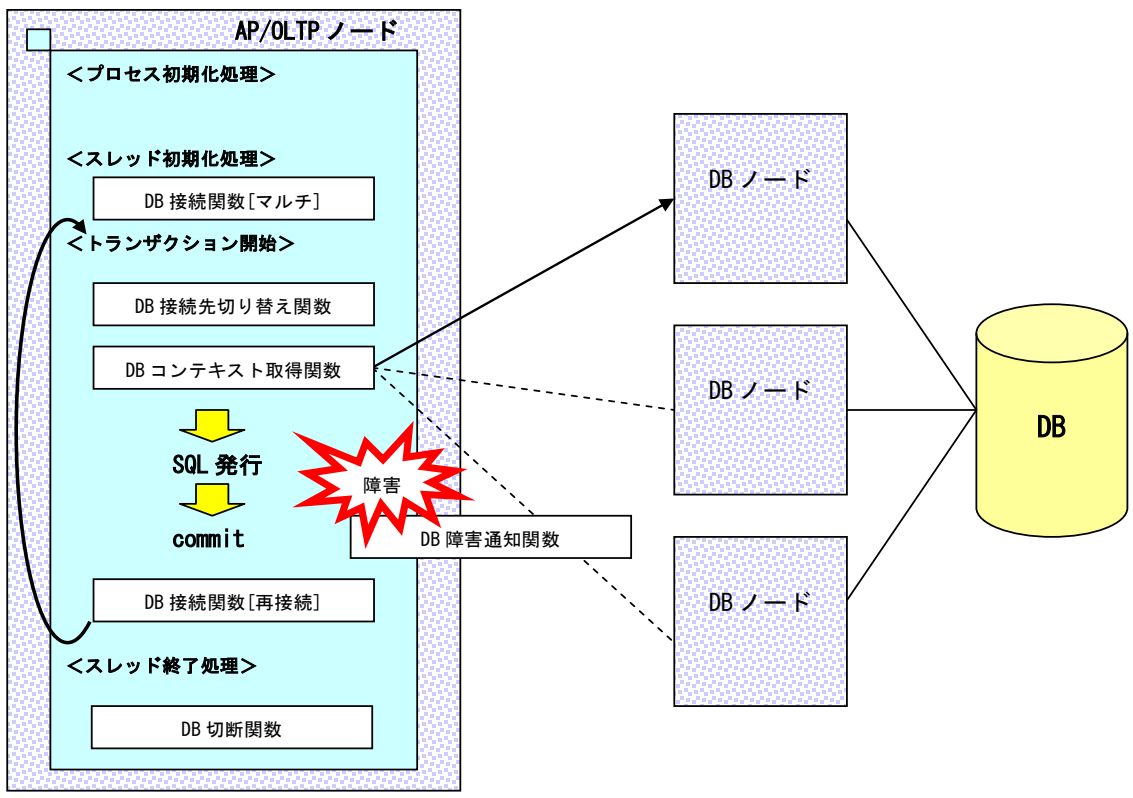


図 2-3 Oracle 常時接続(マルチコネクション)時の API 使用方法

Oracle へ常時接続(シングルコネクション)したい場合は、図 2-4 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。

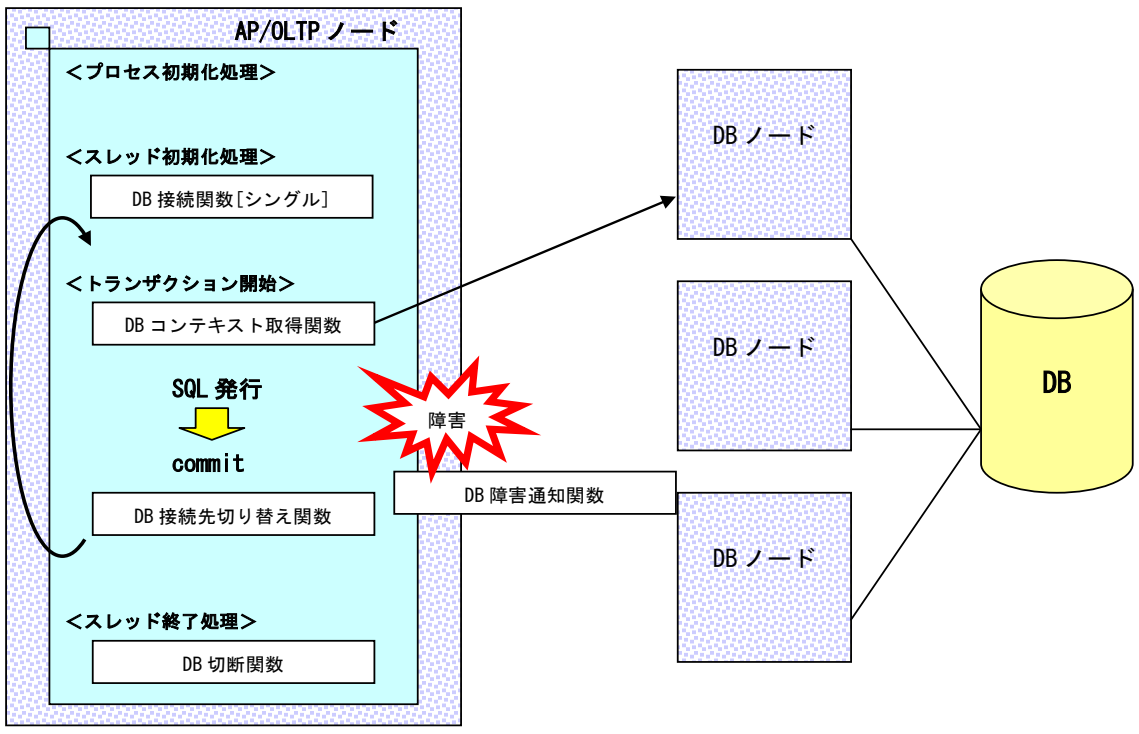


図 2-4 Oracle 常時接続(シングルコネクション)時の API 使用方法

Oracle へ随時接続したい場合は、図 2-5 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。

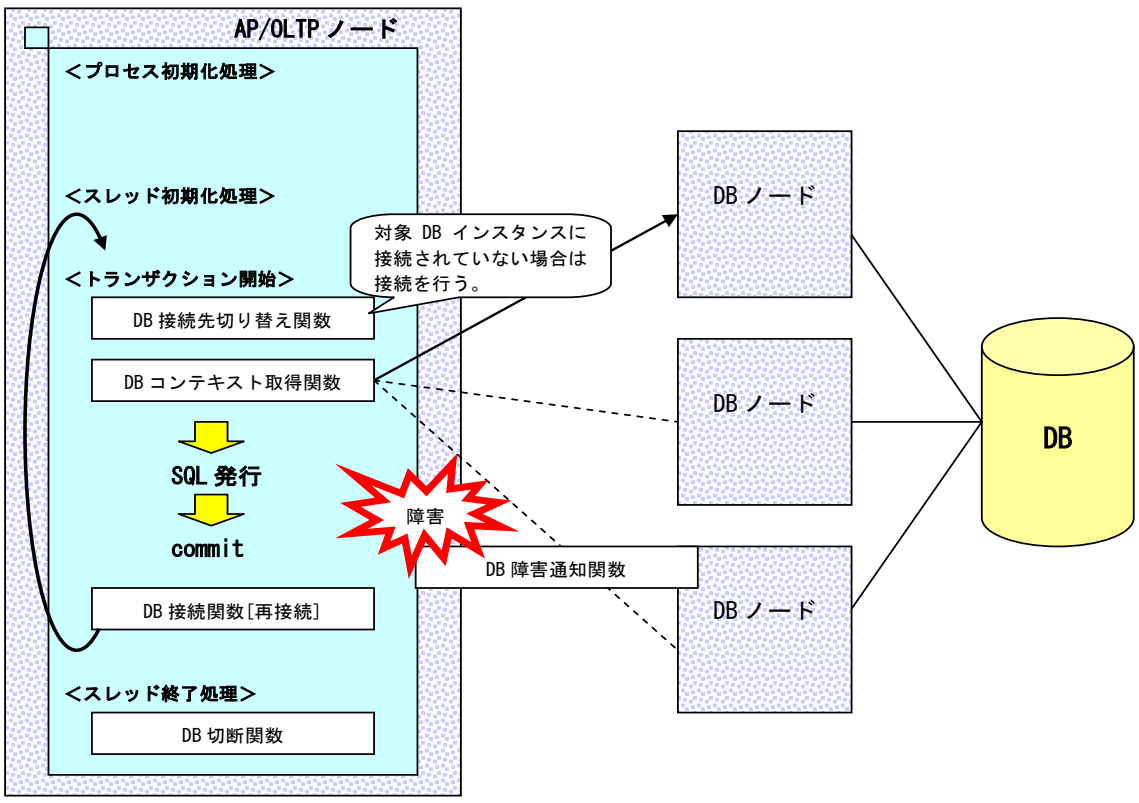


図 2-5 Oracle 随時接続時の API 使用方法

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 AP 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

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、次回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

データベースとのコネクションを切断します。

※　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コネクションを切断します。

2.14　閉塞管理機能

閉塞管理機能は、ノードや CO 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 CO 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、仕掛かり中の処理を完了させるため、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。

- CO 閉塞

  指定された CO 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、CO 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。CO 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(最新の閉塞状態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、最新の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即時に行えます。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(AP ノード群、DB ノード群、OLTP 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、UNIX コマンド、DIOSA/XTP コマンドが指定できます。

実行結果(stdout、stderr)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 API およびコマンドが提供されています。

(1)　　C 関数

diosacmdsend　　：コマンド配信を要求します。

diosacmdconf　　：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

dicmdsend　　：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(SYSMAP 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 CMDSEND 節の SRVGRP 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(DIOSAMAP 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)　AP ノード

指定論理システム配下の AP ノードを配信対象とします。

(c)　OLTP ノード

指定論理システム配下の OLTP ノードを配信対象とします。

(d)　DB ノード

指定論理システム配下の DB 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)　ALL 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)　ANY 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

API 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(CMDSENDINFO 節の BLOCKCHK)の定義値に NO を指定している場合は、API 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。

配信先		自ノード配信対象有無	論理システム1						論理システム2						論理システム3					
			ノード11 AP	ノード12 AP	ノード13 OL	ノード14 OL	ノード15 OL	ノード16 DB	ノード21 AP	ノード22 AP	ノード23 OL	ノード24 OL	ノード25 OL	ノード26 DB	ノード31 AP	ノード32 AP	ノード33 OL	ノード34 OL	ノード35 OL	ノード36 DB
ノード21	－	－							○											
サーバグループ01	ALL型	－		○							○				○					
サーバグループ02		対象	○	○							○				○					
サーバグループ02		除外		○							○				○					
サーバグループ01	ANY型	－		△							△				△					
サーバグループ02		対象	△	△							△				△					
サーバグループ02		除外		△							△				△					
論理システム1	ALL型	－							○	○	○	○	○	○						
		対象	○	○	○	○	○	○												
		除外		○	○	○	○	○												
	ANY型	－							△	△	△	△	△	△						
		対象	△	△	△	△	△	△												
		除外		△	△	△	△	△												
論理システム1-AP	ALL型	－							○	○	○									
		対象	○	○																
		除外		○																
	ANY型	－							△	△	△									
		対象	○	○																
		除外		○																
論理システム1-OLTP	ALL型	－									○	○	○							
		対象			○	○	○													
		除外			○		○													
	ANY型	－									△	△	△							
		対象			△	△	△													
		除外			△		△													
論理システム1-DB	ALL型	－												○						
		対象						○												
		除外																		
	ANY型	－												△						
		対象						△												
		除外																		
指定なし	－	－	○																	

用例　AP ： APサーバ　　○ ： コマンドを配信するノード
　　OL ： OLTPサーバ　　△ ： コマンドの配信対象ノード
　　DB ： DBサーバ　　この中からラウンドロビン
　　方式で1ノードを決定する

自ノード(配信元ノード):網掛けセルとする
サーバグループ01:ノード12、ノード22、ノード31とする
サーバグループ02:ノード11、ノード12、ノード22、ノード31とする
※ 上記例の論理システム1、ALL-DB型で自ノードを除外した指定の場合、配信先ノードが存在しないためエラーとなる。
※ コマンド配信先にシステム内全ノード宛が存在しないが、サーバグループを登録することでサーバグループ内全ノード宛(ALL配信)、及びサーバグループ内の任意の1ノード(ANY配信)に配信が可能である。

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

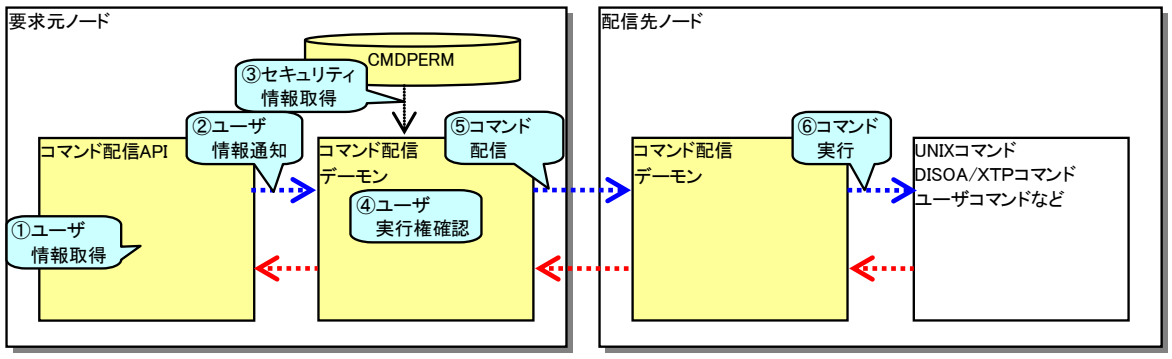
```
dicmdsend –t dixxxcmd
```

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 CMDSEND 節の CMDPERM 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

```
%CMDPERM
    DFLTPERM = NOEXEC     ←未登録コマンドの実行不可
    %CMDGRP
        NAME = CMDGRP01
        %CMDTEXT TEXT = dicmdxx01;
        ・・・
        ・・・
;
%CMDGRP
    NAME = CMDGRP02
    %CMDTEXT TEXT = dicmdxx02;
    ・・・
    ・・・
;
%EXPERM
    USERTYPE   = USER
    NAME       = user1
    PERMISSION = EXEC
    %CMD CMDGRP = CMDGRP01;
;
%EXPERM
    USERTYPE   = USERGRP
    NAME       = group1
    PERMISSION = EXEC
    %CMD CMDGRP = CMDGRP02;
;
%EXPERM
```

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

この例では、ユーザ“user1”、ユーザグループ“group1、group2”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。

user1

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。

group1 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。

group2 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

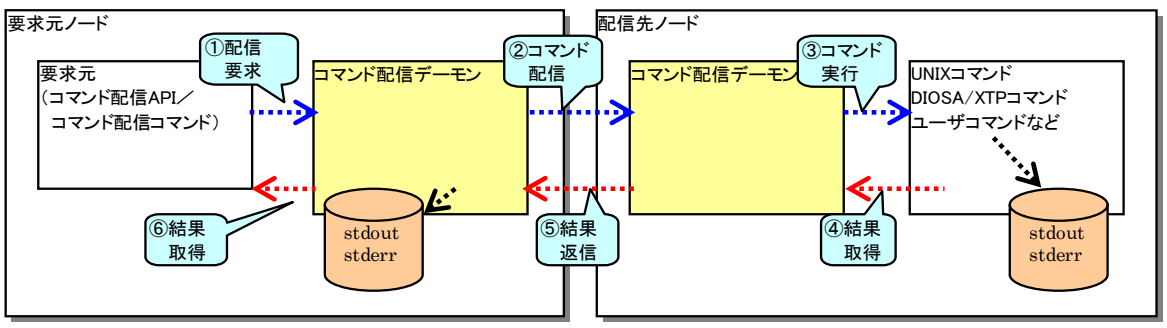
上記に含まれないユーザについては、全てのコマンドが実行不可となります。

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。

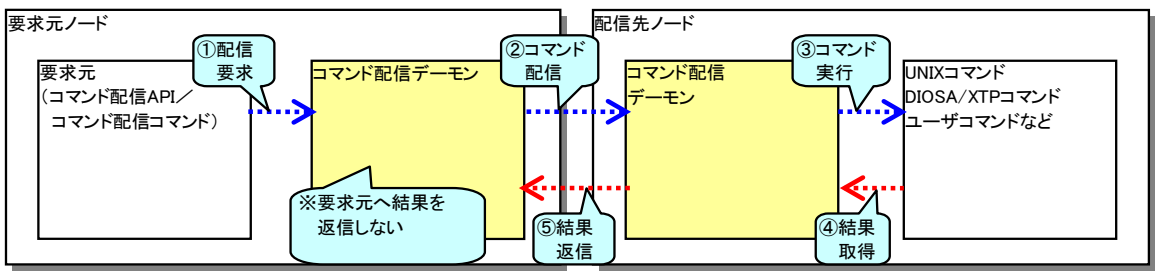


この場合、要求元では結果が返信されるまで、待合せを行います。

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

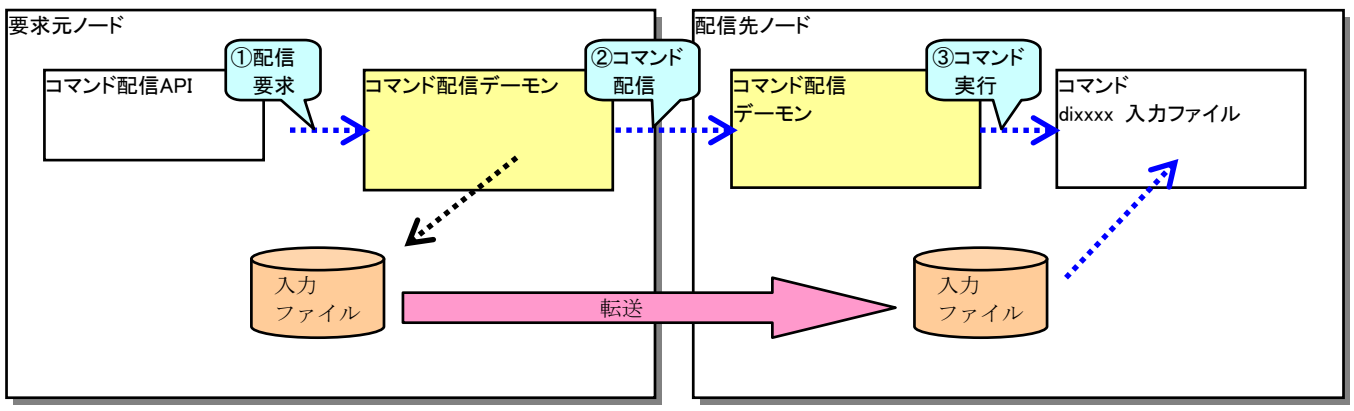
コマンド配信のみを行い、結果の確認が必要ない場合に使用します。



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。



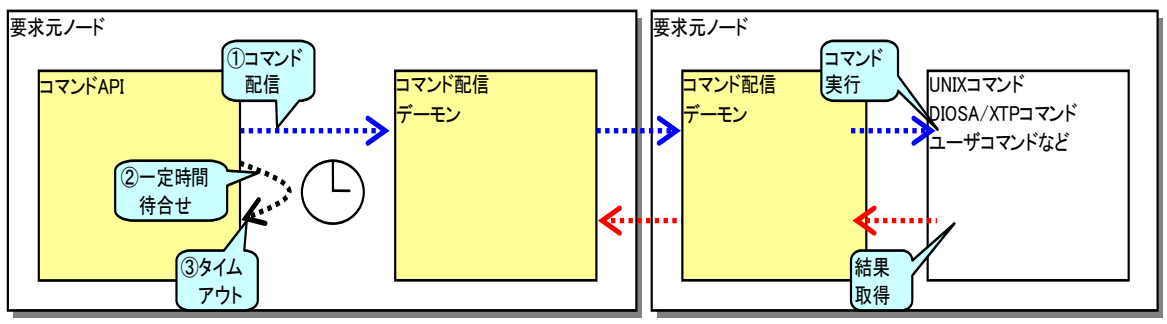
配信元の入力ファイルおよび配信先に転送した入力ファイルは、コマンドの実行が完了した後、削除することができます。

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考えられます。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。

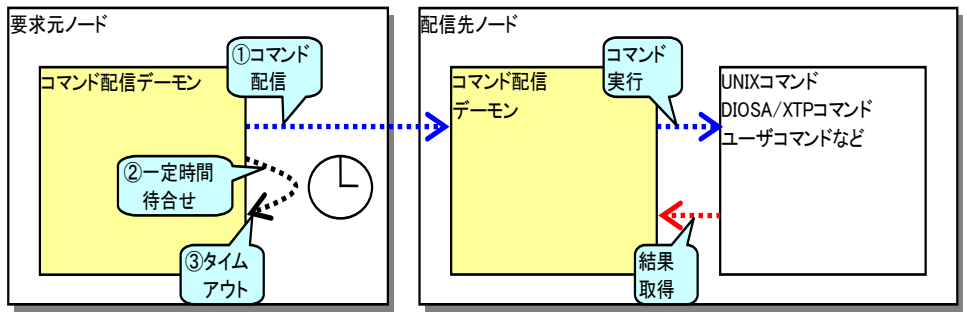
①コマンド、及び API のパラメータ値

②環境変数 DIOSA__CDDAPITIMEOUT 値

③環境定義 CMDSEND 節の CMDSENINFO 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ーモンは、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。

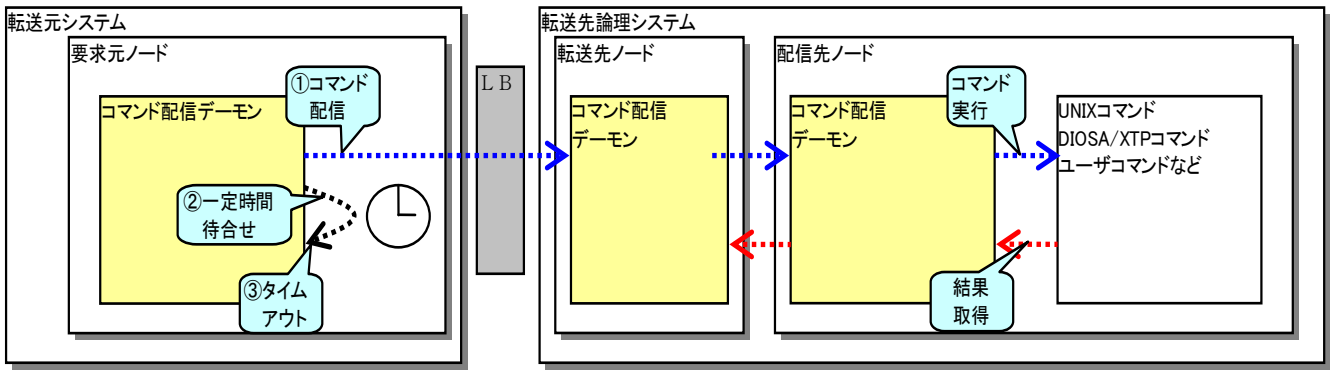
①コマンド、及び API のパラメータ値

②環境変数 DIOSA__CDDEXECTIMEOUT 値

③環境定義 CMDSEND 節の CMDSENINFO 項の EXECTIMEOUT 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ーモンは、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。

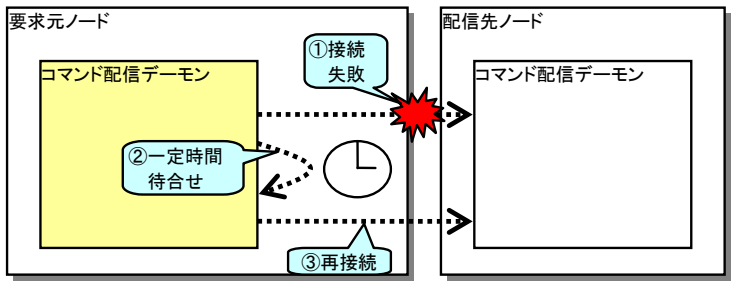
リトライ処理では、一定インターバル待合せ後に、再配信を行います。さらに、リトライ処理は指定されたリトライ回数のみ行います。

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 WAIT 状態となるので注意が必要です。

(1)　　ALL 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 ALL 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(connect)に失敗
- 配信先ノードへの送信(send)に失敗

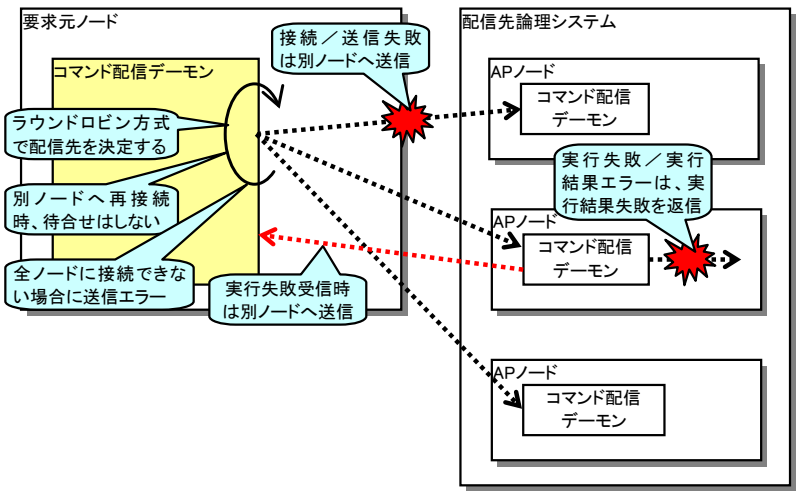


(2)　　ANY 型配信の場合

論理システム、サーバグループ ANY 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。

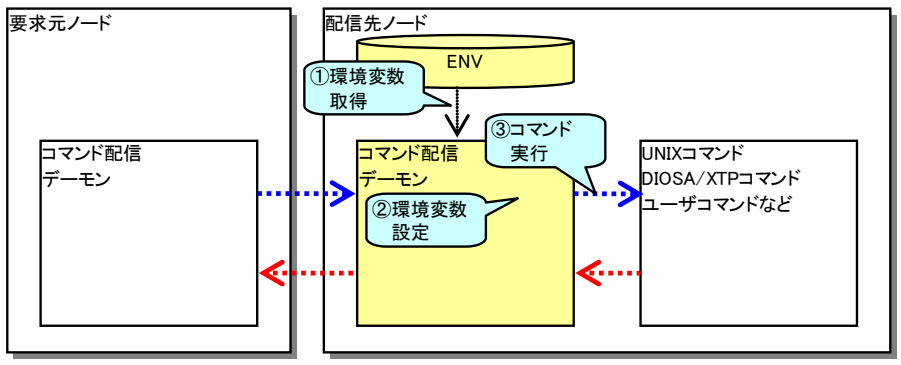
- 配信先ノード(配信先論理システム)が閉塞状態
- 配信先ノードへの接続(connect)に失敗
- 配信先ノードへの送信(send)に失敗
- 配信先ノードでコマンド実行に失敗

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 CMDSEND 節の ENV 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 CMDSEND 節の CMDSENDINFO 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。

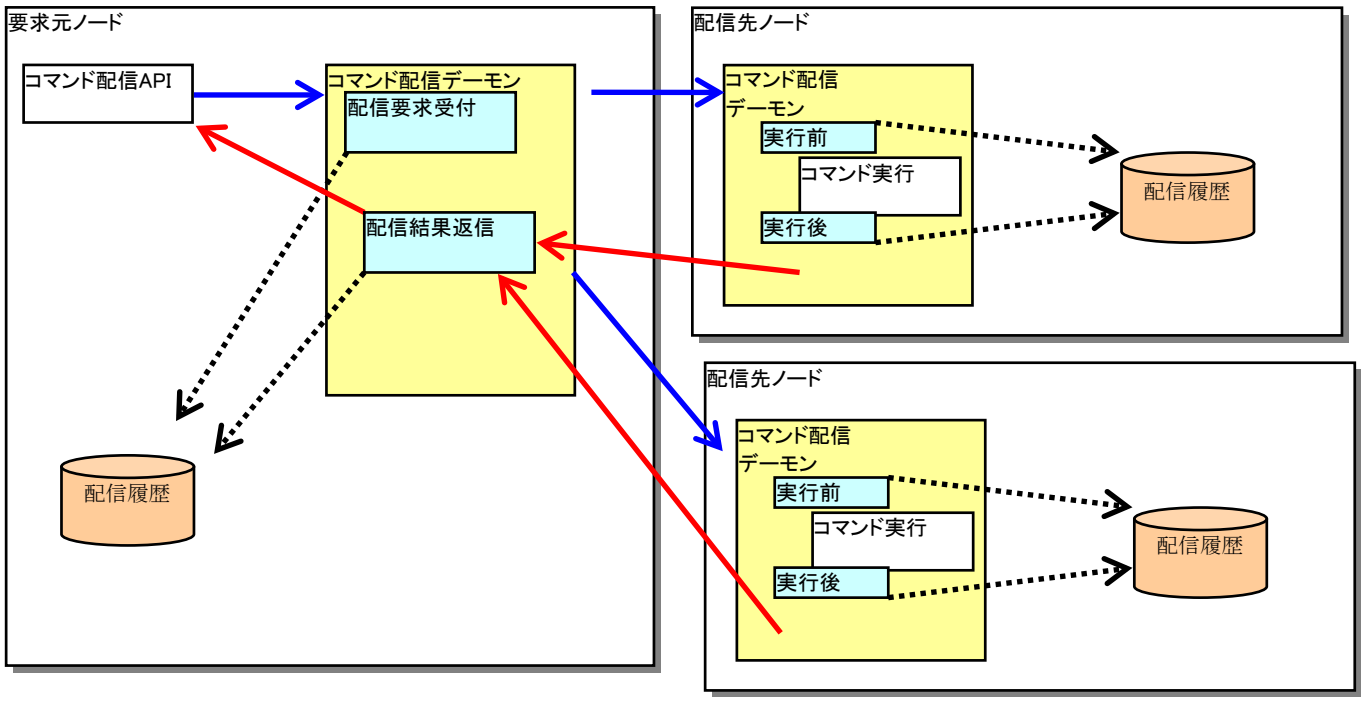
- 日時
- ユーザ名
- コマンド
- 配信先指定/実行ノード名
- 実行結果(ステータス)　など

(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。

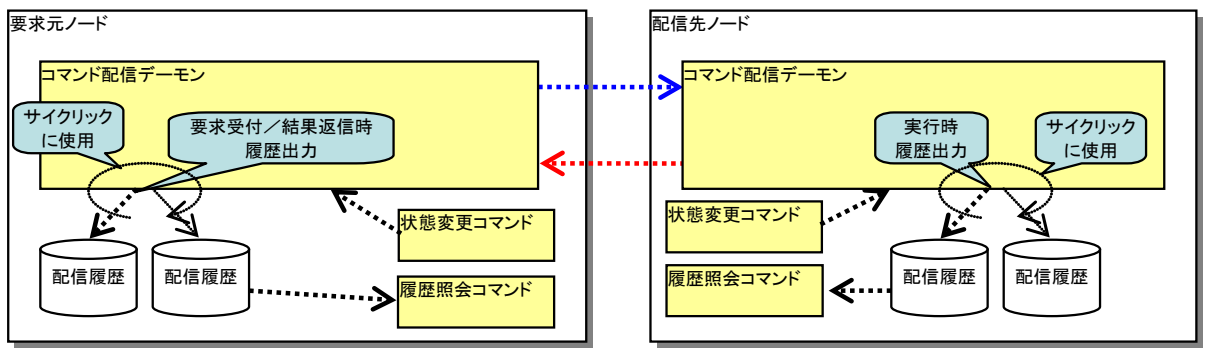
- コマンド配信要求受付時
- コマンド配信結果返信時(配信先ノード数分)
- コマンド実行前
- コマンド実行後(実行結果)



(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 CO 実行時の履歴情報を抽出することが可能です。

2.16　電文保証機能

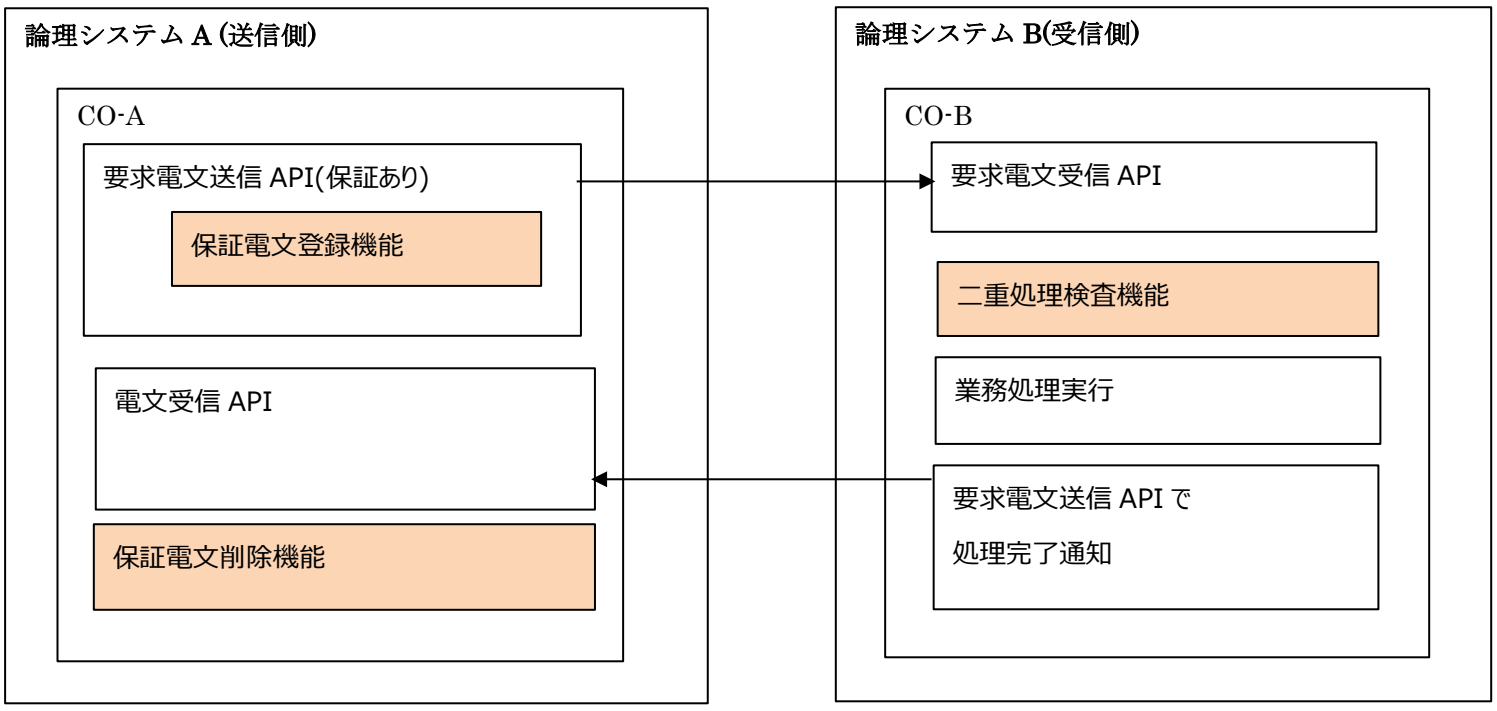
電文保証機能は、他論理システムまたは他 OLTP ノードに送信した電文が、送信先で確実に処理されることを保証するための機能です。電文間で処理順序の制御が必要な場合は、先行する電文が処理完了してから後続の電文を送信するための順序性保証機能を提供します。

2.16.1　　到達保証機能

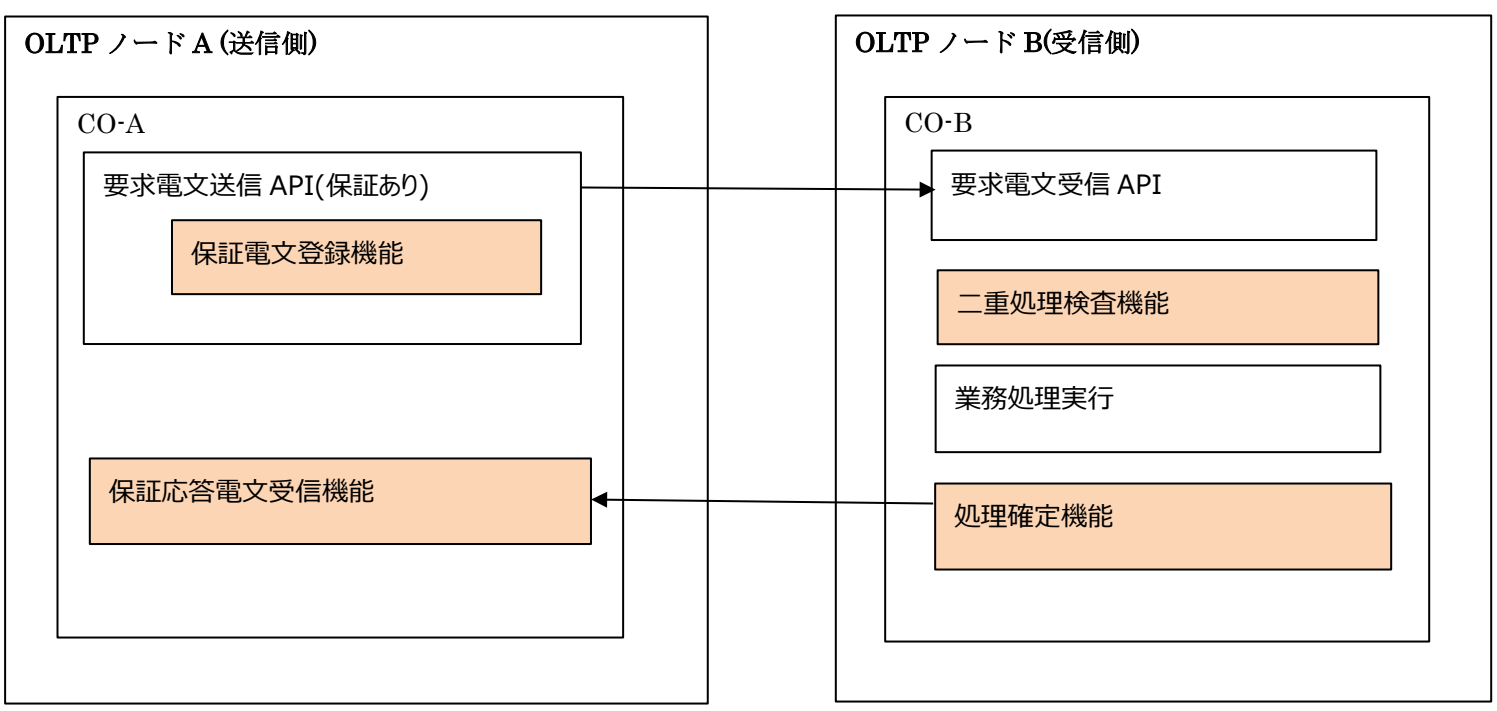
送信した電文が送信先で、確実に処理実施するための到達保証機能を提供します。

論理システム内、論理システム間での電文保証機能の使用イメージは以下の通りです。

■ 論理システム間での電文保証



■ 論理システム内での電文保証



(1)　　**保証電文登機能 ： API 提供**

diosasendtx で電文送信時に利用者が到達保証または順序性保証を指定した場合に、
到達保証・順序性保証の対象と
なります。本機能は、到達保証・順序性保証の対象となった電文を DB（TAM または Oracle）に
保存・管理します。本機能により登録された電文が保証電文再送機能の再送対象となります。

(2)　　**二重処理検査機能 ： API 提供**

本機能は、到達保証・順序性保証対象の電文を受信した場合に、当該電文に対する業務が既に実施済みであるかを通知します。

到達保証・順序性保証対象の電文は、電文の到達確認の遅れや処理の遅延により、到達保証・順序性保証対象の電文の送信先において一度受信した電文を再度受信する可能性があります。そのため、業務処理を開始する前に、本機能を利用して、受信した電文に対する業務処理がまだ行われていないことを確認してください。

(3)　　**処理確定機能（論理システム内限定）： API 提供**

受信した電文に対する業務処理が完了したことを、電文の送信元に通知する機能です。

アプリケーションは電文に対する業務処理が完了した後に、本機能を利用して、業務処理が完了したことを送信元に通知してください。なお、保証電文削除機能を使用する場合は本機能による通知は不要です。

(4)　　**保証応答電文受信機能（論理システム内限定）： 制御 TPP**

処理確定機能からの通知を受信し、業務処理が完了した電文情報を削除する。
本機能により該当電文の再送を停止します。

(5)　　**保証電文削除機能 ： API 提供**

業務処理が完了した電文情報を削除する。本機能により該当電文の再送を停止します。

2.16.2　順序性保証機能

業務データの整合性を保つために、送信電文の到達保証に加えて電文の送信順序を保証する機能です。

同一順序性保証グループかつ、同一 DB（TAM は MAPID、Oracle は RGSET）の範囲で送信毎の順序性を保証します。

(1)　　**保証電文登機能 ： API 提供**

diosasendtx で到達保証機能の機能に加え、同一順序性保証グループかつ、同一 DB で
送信中の電文が存在する場合は、送信中の電文の到達を確認するまで当該電文の送信を保留します。

(2)　　**二重処理検査機能 ： API 提供**

到達保証機能と同等です。

(3)　　**処理確定機能（論理システム内限定）： API 提供**

到達保証機能と同等です。

(4)　　**保証応答電文受信機能（論理システム内限定）： API 提供**

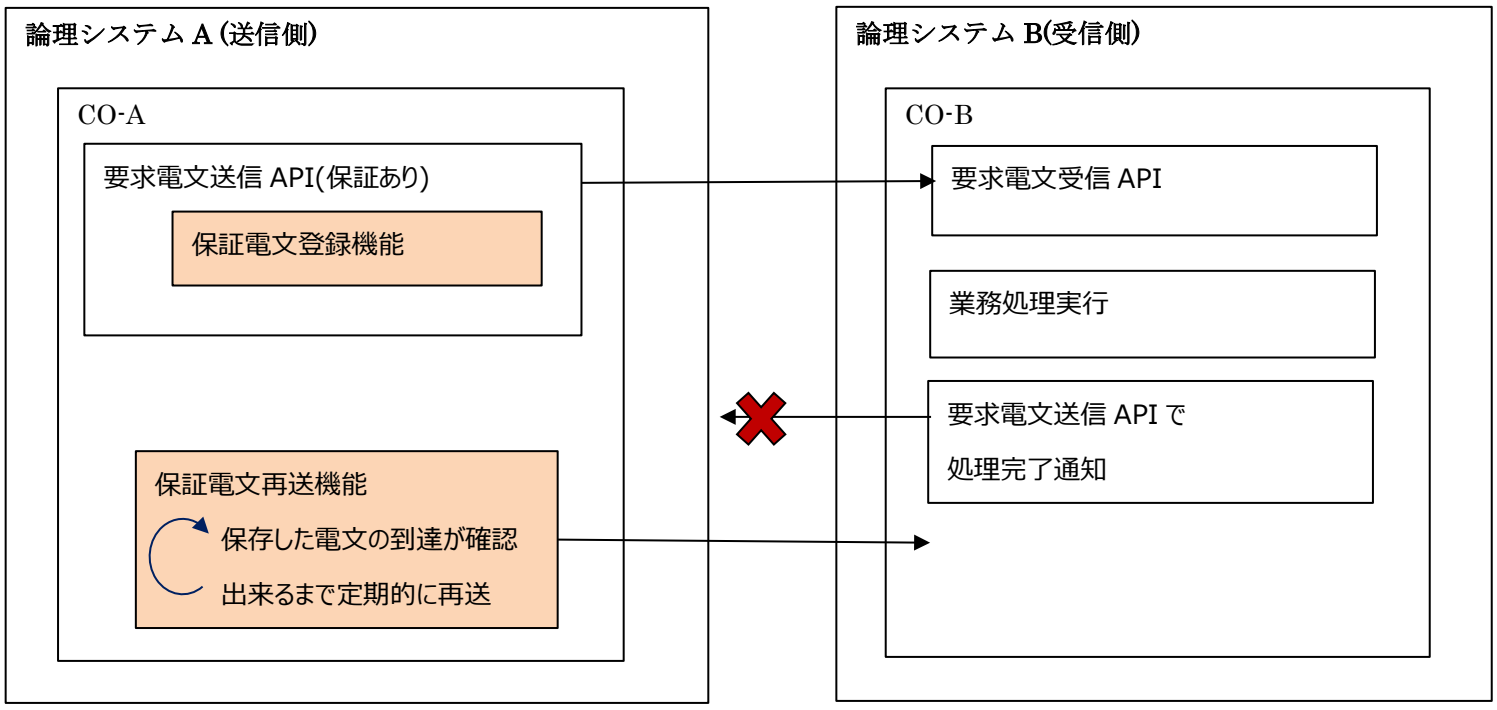
到達保証機能に加えて、順序性保証の次電文送信を行います。次電文を送信する直前で、
保証電文送信有無判定出口(利用者出口)を呼び出し、その判定結果に応じた電文送信を行います。

(5)　　**保証電文削除機能 ： API 提供**

到達保証機能に加えて、順序性保証の次電文送信を行います。次電文を送信する直前で、
保証電文送信有無判定出口(利用者出口)を呼び出し、その判定結果に応じた電文送信を行います。

2.16.3　保証電文再送機能

保証電文再送機能(制御 TPP)の処理イメージを以下に示します。



保証電文登録機能で保存した電文の処理完了を確認するまで、自動的に再送を行います。本機能は、環境定義(APMGRNT 節 MSGGNT 項 RSNDCHKINVL) で指定された時間間隔で、再送対象の電文を送信します。
再送対象の電文は、前回の送信から電文毎に設定した再送間隔時間を超過したものです。
電文と送信する直前に保証電文送信有無判定出口(利用者出口)を呼び出し、その判定結果に応じた
電文送信を行います。
なお、最大送信回数を超過した電文は、再送を行わずリトライオーバーCO (APMGRNT 節 OVERSEND 項) に
電文を渡します。利用者はリトライオーバーCO 上で保証電文削除機能により再送を停止して下さい。
なお、リトライオーバーCO が未定義の場合は、最大送信回数を超過した電文の再送を自動で停止します。

2.16.4　保証電文送信有無判定出口

各電文保証の機能が保証電文を送信する前に、利用者の関数を呼び出して、電文内容の編集や送信するかどう

かの判定が可能です。送信有無の判定は以下の４つから選択可能です。

判定	動作概要
自動判定	送信回数に従って電文保証機能が判断します。 (最大送信回数以下なら送信、超えていたらリトライオーバー)
保留	次のインターバル時間まで送信延期します。(送信回数を増やしません。)
スキップ	次のインターバル時間まで送信延期します。送信回数を増やします。)
削除	送信を取りやめて保証電文を削除します。 順序性保証の次電文送信は行いません。

2.16.5　受信電文情報削除機能

保証電文の受信側で、二重処理検査機能が検査用に使う受信電文管理情報の削除を行います。

本機能は、環境定義（APMGRNT 節 MSGGNT 項 DELCHKINVL）で指定された時間間隔で、削除対象の受信電文管理情報を削除します。

削除対象の受信電文管理情報は、保存時間（APMGRNT 節 MSGGNT 項 RGNTTIME）を超過したものです。

2.16.6　次回送信タイミング変更機能

保証電文登録機能により保存した電文の次回送信タイミングを変更できます。変更対象は次回送信のみで、次々回以降の送信は従来の再送間隔で行われます。

2.16.7　保証電文削除機能

保証電文の電文情報を削除します。通常は保証電文削除機能や再送処理のリトライオーバーを契機に自動的に削除して再送を停止しますが、処理が完了していない電文を強制的に削除したい場合に使用します。本機能は全削除、宛先アクセスポイント指定、メッセージ ID 指定の 3 つの削除モードを提供します。

2.16.8　宛先論理システム切替機能

保証電文の宛先論理システムが拠点切り替えを実施した際にも、保証電文の送信を継続可能とするため、登録済み保証電文の宛先アクセスポイントを切り替えます。

2.16.9　定義変更機能

電文保証機能の以下の環境定義を運用中に変更します。変更した環境定義の値は、即時に各プロセスに反映されます。

- 再送時間間隔(diosasendtx で指定されなかった場合の規定値)
- 最大送信回数(diosasendtx で指定されなかった場合の規定値)
- 受信電文情報保持時間
- 順序性保証グループ情報の保持時間)

2.16.10　照会機能

動作中の電文保証機能の定義情報や、滞留している保証電文の情報を照会することができます。

以下の 3 つの照会モードを提供します。

- 定義情報照
- 滞留電文照会
- 保証電文情報照会

2.16.11　DB 初期化機能

電文保証機能が DB（TAM または Oracle）に保存しているデータの初期化を行います。

本機能を実行すると、実行以前の保証電文に対して、到達保証と順序性保証が行われなくなるため、運用中に使用しないで下さい。

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、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 制御利用者出口の開発

利用者出口は、アーギュメントとして diosauca(t_diosa_uca)が渡されます。diosauca は CO 制御が用意します。

出口には、以下のものがあります。

全ての出口は CO 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

・受信電文解析出口

・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

・プロセス終了出口

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

(1)　　プロセス初期化出口

CO 制御 TPP プロセス開始時に一度だけ呼び出されます。

AP 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。

```
#include <diosa.h>

void トランザクション初期化出口( t_diosa_uca *pstUca )
{

   利用者任意の処理

    pstUca->Status     = DIOSA_ST_DONE;
    return;
}
```

(2)　　受信電文解析出口

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、圧縮電文の解凍、CO 名の決定、更新 DB の決定を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。

・CO 名の決定論理

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 名を指定すると、その CO 名は diosarecvtx で受け取ることができます。

(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。

・更新DBの決定論理

基本的には、環境定義でTPBASEモニタのクラス単位に定義することができます。この他、電文送信時と受信電文解析出口で決定することも可能です。

環境定義には以下が定義可能です。

(a)　DBを使わない

(b)　IMのみをを使う

(c)　Oracleのみを使う

(d)　IM、Oracleを使う。

利用者は電文送信する際、送信先DBを指定することができます。指定できるDBはIMのみとなります。IM情報のメインキー、またはMAPIDを指定してTAMのマスタノードへ電文送信を行います。メインキー、MAPID指定で送信した先の更新DB定義が「IMのみを使う」または「IM、Oracleを使う」と定義されていなくてはなりません。

電文送信時にIM情報を指定しない場合は、電文受信側の環境定義が有効となります。「IM、Oracleを使う」という定義では既定値としてOracleが選択されます。

受信側のCO制御に受信電文解析出口を用意することで、電文受信側で更新DBを決定、更新することも可能です。受信電文解析出口では、IM、Oracleどちらを更新DBにするか(「IM、Oracleを使う」と定義されている場合)を選択できます。また、IMを更新DBとする場合、メインキー、MAPIDが指定、変更可能となります。Oracleを更新DBとする場合、Orackelインスタンスに対応するRGSET名(環境定義、$DBCTRL)を指定、変更することができます(RGSET名未指定は、$DBCTRLのFEFAULTRGSETを採用)。ただし環境定義の更新DBそのものを変更することはできません。(IMのみを使う定義をOracleのみを使うに変更する等)

「IM、Oracleを使う」場合、更新DBはどちらかが選択されますが、選択されなかったDBの参照は自由です。Oracleの参照は利用者任意で行ってください。IMの参照はCO制御が参照可能な状態とします。

IMとOracle、またはOracleの複数インスタンスを更新する場合は、利用者の責任で行ってください。

```
#include <diosa.h>

void 受信電文解析出口( t_diosa_uca       *pstUca
                       t_diosa_analyze   *pstAnalyze  ))
{

    /* ＤＢ情報の変更 */
    switch( pstAnalyze->DbInfo.Type ){
    case DIOSA_DB_NO:
        break;
    case DIOSA_DB_MAINKEY:
    case DIOSA_DB_IM:
        pstAnalyze->DbInfo.Type  = DIOSA_DB_MAPID;
        pstAnalyze->DbInfo.MapId = 99;
        break;
    case DIOSA_DB_RGSET:
        strcpy( pstAnalyze->DbInfo.RgSet, “ORACLE_INSTANCE01” );
        break;
    default:
        break;
    }

    /* ＣＯ名を指定する */
    if( ‘¥0’ == pstAnalyze->CoName[0] ){
        strcpy(pstAnalyze->CoName, “DEFAULT_CONAME” );
    }

    /* 受信電文を解凍する  */
    nRet = diosamsgbufalloc( 2048, &msgbuf);
    if( nRet != DIOSA_DONE ){
        エラー処理;
        pstUca->Status     = DIOSA_ST_ABORT;
        pstUca->UserStatus = nRet;
        return;
    }
    利用者の電文解凍論理
    pstAnalyze->EditFlag = DIOSA_ON;
    pstAnalyze->EditSize = 2048;
    pstAnalyze->EditBuf  = msgbuf;

    pstUca->Status     = DIOSA_ST_DONE;
    return;
}
```

(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

AP がアボート要求を返却し、また、CO 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)　プロセス終了出口

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。

3.1.3 CO制御とのインタフェース

(1) CO呼び出しインタフェース

CO呼び出しは、アーギュメントとしてdiosauca(t_diosa_uca)が渡されます。diosaucaはCO制御が用意します。

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

特殊なケースとして、受信電文解析出口にはdiosaucaと受信電文解析用構造体(t_diosa_analyze)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。

```
#include <diosa.h>

void ＣＯ名( t_diosa_uca *pstUca )
{
    t_diosa_dcuca       stRecvDcUca;            /* DCUCA構造体(受信用)          */
    int                 nRet;                   /* 戻り値                       */
    char                *pcRecvBuf;             /* 受信電文ポインタ             */

    /* diosarecvtx */
    nRet = diosarecvtx( &stRecvDcUca, &pcRecvBuf );
    if( nRet != DIOSA_DONE ){
        エラー処理;
        pstUca->Status     = DIOSA_ST_ABORT;
        pstUca->UserStatus = nRet;
        return;
    }

    ＣＯ毎の業務処理

    pstUca->Status = DIOSA_ST_DONE;
    return;
}
```

(2) COからの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。

(3) 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(Status)」の項目、および「利用者コード(UserStatus)」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

終了インタフェースと diosagoback

```
#include <diosa.h>

int ＣＯから呼び出された関数名( int  *p_nFunc )
{
    t_diosa_uca          *pstUca;
    ～

    if( エラー発生 ){
        エラー処理;
        nRet = diosaucaget( &pasUca );          ← diosauca のアドレスを取得する
        pstUca->Status     = DIOSA_ST_ABORTSTOP; ← diosauca の状態を更新する
        pstUca->UserStatus = 9999;              ← 必要に応じて利用者ステータスを設定する
        nRet = diosagoback( );                  ← 正常動作は制御が戻されることはない
    }

    pstUca->Status = DIOSA_ST_DONE;
    return 0;
}
```

(1)　　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。利用者との情報交換には diosadcuca(t_diosa_dcuca)を使います。本構造体は利用者が確保します。

(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

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 の電文タイプを論理システム間に、宛先アクセスポイント名を指定とすることにより可能となります。

```
#include <diosa.h>

void CO名( t_diosa_uca   *pstUca )
{
    t_diosa_msgbuf      *msgbuf = NULL;          /* 電文格納領域ポインタ          */
    t_diosa_dcuca       stRecvDcUca;             /* DCUCA 構造体(受信用)          */
    t_diosa_dcuca       stSendDcUca;             /* DCUCA 構造体(送信用)          */
    int                 nRet;                    /* 戻り値                        */
    char                *pcRecvBuf;              /* 受信電文ポインタ              */

    /* diosarecvtx */
    nRet = diosarecvtx( &stRecvDcUca, &pcRecvBuf );
    if( nRet != DIOSA_DONE ){
        エラー処理;
        pstUca->Status     = DIOSA_ST_ABORT;
        pstUca->UserStatus = nRet;
        return;
    }

    /* diosamsgbufalloc */
    nRet = diosamsgbufalloc( 2048, &msgbuf);
    if( nRet != DIOSA_DONE ){
        エラー処理;
        pstUca->Status     = DIOSA_ST_ABORT;
        pstUca->UserStatus = nRet;
        return;
    }

    /* diosasendtx */
    memset( &stSendDcUca, '¥0', sizeof(t_diosa_dcuca ));
    stSendDcUca.Db.Type   = DIOSA_DB_NO;
    stSendDcUca.MsgGnt    = DIOSA_MSGGNT_NO;
    stSendDcUca.MsgSize   = 1024;                          ←  電文長
    stSendDcUca.MsgType   = DIOSA_MSG_LS;                  ←  電文タイプ(論理システム間)
    strcpy( stSendDcUca.LSys.AcsPoint, "ACSPOINT001" );    ←  アクセスポイント名

    nRet = diosasendtx( &stSendDcUca, msgbuf );
    if( nRet != DIOSA_DONE ){
        エラー処理;
        pstUca->Status     = DIOSA_ST_ABORT;
        pstUca->UserStatus = nRet;
        return;
    }

    pstUca->Status = DIOSA_ST_DONE;
    return;
}
```

(ii) 論理システム内電文送信(派生)

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。

派生には以下の電文タイプがあります。

・CO 名指定は、DIOSA_MSG_CO

・中継 CO 名指定は、DIOSA_MSG_RELAYCO

・TXID 指定は、DIOSA_MSG_TX

・中継 TXID 指定は、DIOSA_MSG_RELAYTX

```
#include <diosa.h>

void CO名( t_diosa_uca   *pstUca )
{
    t_diosa_msgbuf      *msgbuf = NULL;          /* 電文格納領域ポインタ            */
    t_diosa_dcuca       stSendDcUca;             /* DCUCA構造体(送信用)             */
    int                 nRet;                    /* 戻り値                          */

    /* diosasendtx */
    memset( &stSendDcUca, '¥0', sizeof(t_diosa_dcuca ));
    stSendDcUca.Db.Type   = DIOSA_DB_NO;
    stSendDcUca.MsgGnt    = DIOSA_MSGGNT_NO;
    stSendDcUca.MsgSize   = 1024;                          ←  電文長
    stSendDcUca.MsgType   = DIOSA_MSG_CO;                  ←  電文タイプ(CO名指定)
    strcpy( stSendDcUca.CoName, "CO_0001" );               ←  CO名

    nRet = diosasendtx( &stSendDcUca, msgbuf );
    if( nRet != DIOSA_DONE ){
        エラー処理;
        pstUca->Status     = DIOSA_ST_ABORT;
        pstUca->UserStatus = nRet;
        return;
    }

    pstUca->Status = DIOSA_ST_DONE;
    return;
}
```

(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 の電文タイプ(MsgType)を連鎖(DIOSA_MSG_CHAIN)とすることにより連鎖要求となります。

(iv) 保留

処理中の電文を一旦保留します。diosadcuca の電文タイプ(MsgType)を保留(DIOSA_MSG_REST)とすることにより保留要求となります。

(v)　通常送信と強制送信

通常送信と強制送信は diosadcuca の送信モード(SendMode)パラメータを使います。

通常送信は、DIOSA_SEND_FORCE(0)

強制送信は、DIOSA_SEND_FORCE(1)

```
#include <diosa.h>

void CO名( t_diosa_uca   *pstUca )
{
    t_diosa_msgbuf      *msgbuf = NULL;          /* 電文格納領域ポインタ            */
    t_diosa_dcuca       stSendDcUca;             /* DCUCA構造体(送信用)             */
    int                 nRet;                    /* 戻り値                          */

    /* diosasendtx */
    memset( &stSendDcUca, '¥0', sizeof(t_diosa_dcuca ));
    stSendDcUca.Db.Type   = DIOSA_DB_NO;
    stSendDcUca.MsgGnt    = DIOSA_MSGGNT_NO;
    stSendDcUca.MsgSize   = 1024;                          ←  電文長
    stSendDcUca.MsgType   = DIOSA_MSG_CO;                  ←  電文タイプ(CO名指定)
    stSendDcUca.SendMode  = DIOSA_SEND_FORCE;              ←  送信モード(強制)
    strcpy( stSendDcUca.CoName, "CO_0001" );               ←  CO名

    nRet = diosasendtx( &stSendDcUca, msgbuf );
    if( nRet != DIOSA_DONE ){
        エラー処理;
        pstUca->Status     = DIOSA_ST_ABORT;
        pstUca->UserStatus = nRet;
        return;
    }

    pstUca->Status = DIOSA_ST_DONE;
    return;
}
```

(vi) ラウンドロビンと優先度送信

ラウンドロビンと優先度送信は diosadcuca のラウンドロビン(Roundrobin)パラメータを使います。

優先度送信は、DIOSA_NO(0)

ラウンドロビン送信は、DIOSA_YES(1)

```
#include <diosa.h>

void CO名( t_diosa_uca   *pstUca )
{
    t_diosa_msgbuf      *msgbuf = NULL;          /* 電文格納領域ポインタ            */
    t_diosa_dcuca       stSendDcUca;             /* DCUCA構造体(送信用)             */
    int                 nRet;                    /* 戻り値                          */

    /* diosasendtx */
    memset( &stSendDcUca, '¥0', sizeof(t_diosa_dcuca ));
    stSendDcUca.Db.Type   = DIOSA_DB_NO;
    stSendDcUca.MsgGnt    = DIOSA_MSGGNT_NO;
    stSendDcUca.MsgSize   = 1024;                          ←  電文長
    stSendDcUca.MsgType   = DIOSA_MSG_CO;                  ←  電文タイプ(CO名指定)
    stSendDcUca.Roundrobin= DIOSA_YES;                     ←  ラウンドロビン送信
    strcpy( stSendDcUca.CoName, "CO_0001" );               ←  CO名

    nRet = diosasendtx( &stSendDcUca, msgbuf );
    if( nRet != DIOSA_DONE ){
        エラー処理;
        pstUca->Status     = DIOSA_ST_ABORT;
        pstUca->UserStatus = nRet;
        return;
    }

    pstUca->Status = DIOSA_ST_DONE;
    return;
}
```

(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、電文タイプ、キー情報等の情報は、diosadcuca に返却します。

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、電文送信 API(diosasendtx)を使っておこないます。電文送信 API(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電文タイプ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。

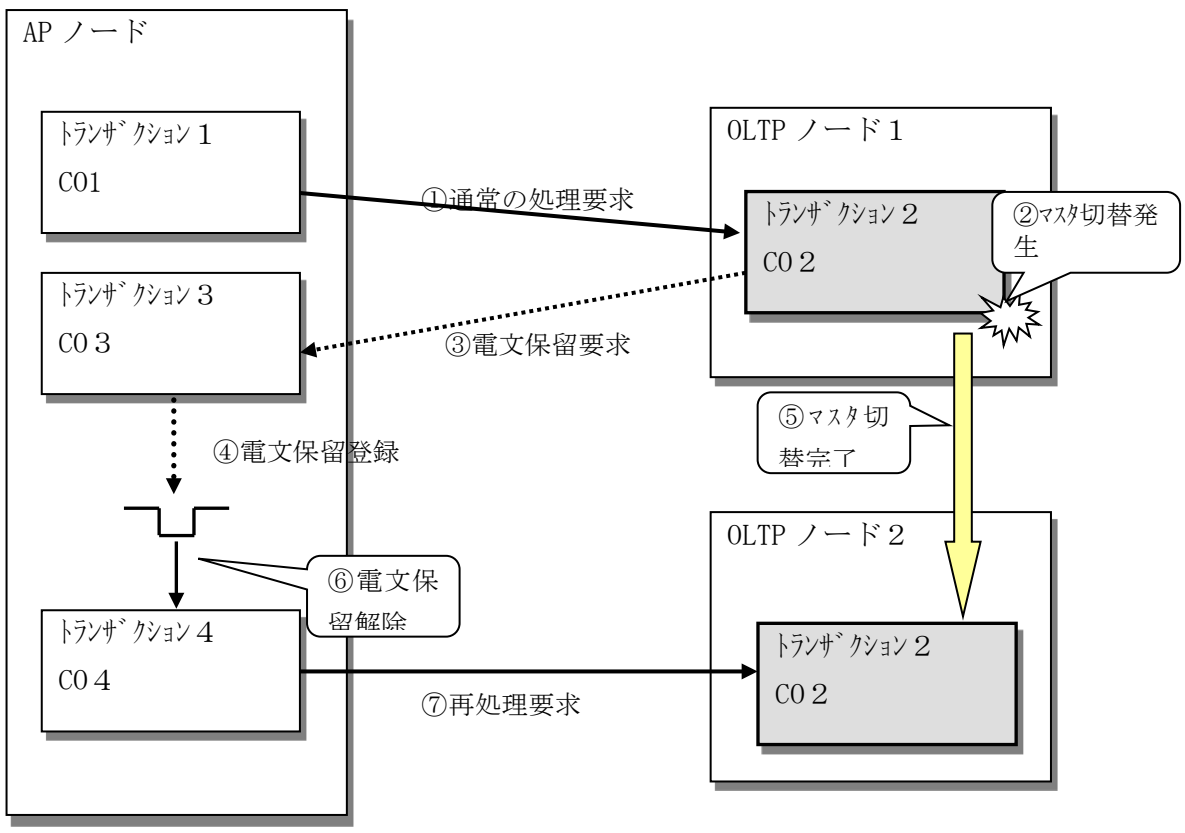
④ CO の途中、またはコミット時に電文保留要件が発生した場合、保留電文となるトランザクション開始時に受信したオリジナル電文を取得することができます。トランザクション開始時電文の取得 API(diosagetsrctx)を使います。

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。電文保留制御コマンド(dicoctxblock)を使います。

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 AP 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

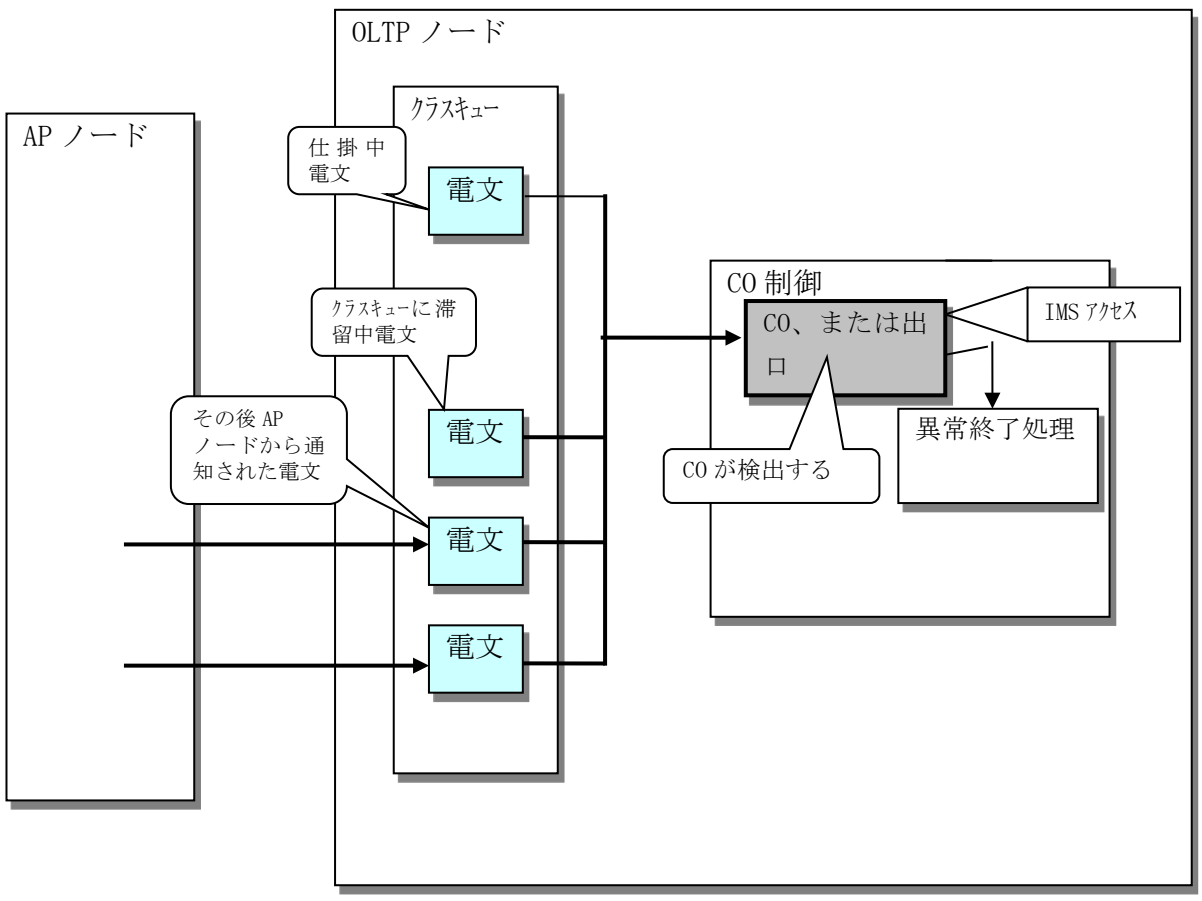
処理中トランザクションの CO、または出口(インメモリサーバへのアクセスが可能な出口　詳細は「DIOSA/XTP　API リファレンス　付録 A　API 一覧を参照」)で実行された TAM アクセス API の戻り値で知ることができます。詳しくは「DIOSA/XTP　API リファレンス　2 章　メモリキャッシュ」を参照してください。

コミット API(diosacommit)でマスタ切り替えに遭遇した場合、DIOSA_SWITCH、DIOSA_ESWITCH が返却されます。

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。

(b)　CO 制御が TAM マスタ切り替えを検出

CO 制御は暗黙のコミット実行時に切り替えを検出することができます。切り替えを検出すると自動的にアボート処理に遷移します。


OLTP ノード
AP ノード
ｸﾗｽｷｭｰ
仕掛中電文
電文
ｸﾗｽｷｭｰに滞留中電文
電文
その後 AP ノードから通知された電文
電文
電文
CO 制御
CO、または出口
IMS ｱｸｾｽ
異常終了処理
CO が検出する

(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、AP が異常終了要求を返却した。また、CO 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 CO制御がロールバックを行います。

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。保留する電文をAPノードに強制送信(diosasendtx)することができます。

④ CO制御がコミットを行います。

(b)　アボート＃1出口のTAMマスタ切り替え検出

CO制御がマスタ切り替えを検出した場合は、diosauca-ExitKeyでDIOSA_EK_SWITCH(IMマスタ切替)、またはDIOSA_EK_ESWITCH(IMマスタ障害切替)を通知します。利用者はこの状態を確認して保留処理を行ってください。

利用者がマスタ切り替えを検出した場合、diosauca-ExitKeyにはDIOSA_EK_APREQ(APからの異常終了要求)が通知されます。利用者は、COからの異常終了要求の時、diosauca-UserStatusにマスタ切り替え発生を通知するための ID を設定しておいてください。本情報はアボート＃1 出口呼び出し時、diosauca-UserKeitKeyで通知されます。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。

(c)　ロールバックのTAMマスタ切り替え検出

CO制御がロールバックを行います。

ロールバックがマスタ切り替えに遭遇した場合、該当トランザクションは終了します。

(d)　アボート＃2出口のTAMマスタ切り替え検出

アボート＃1出口と同じです。

保留する電文は、通常送信(コミット同期)、強制送信とも送信可能ですが、通常送信の場合、アボート＃2出口が異常終了を要求すると、送信がキャンセルされてしまうため、強制送信をお勧めします。

なお、例外発生時は本出口の呼び出しは行われませんので、保留電文、エラー応答の強制送信はアボート＃1出口で共通化することができます。

アボート＃2出口が異常終了要求した場合、ロールバック処理を実行後、アボート処理を終了します。

(e)　コミットのTAMマスタ切り替え検出

CO制御がコミットをおこないます。

コミットがマスタ切り替えに遭遇した、または障害が発生した場合、CO制御はもう一度ロールバックを実行してトランザクションを終了します。

(3)　　オリジナル受信電文を取得する

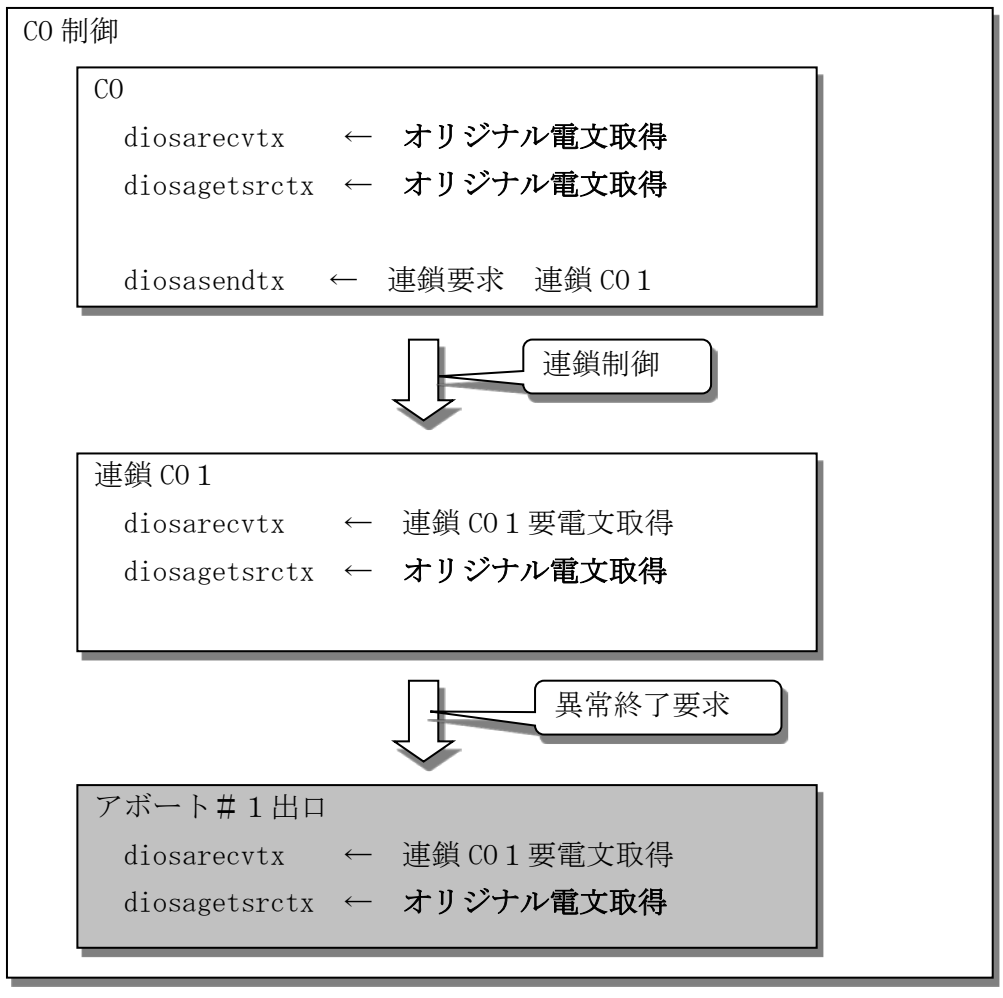
保留する電文の取得と電文情報について記載します。

diosarecvtx(電文受信 API)は最新の電文と電文情報を返却します。CO 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。CO 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 CO、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                                  ＃クラスの既定値定義
         MSGMAX   =   3 2                        ＃電文最大長
         RESTTXID  =  RESTORETXID                ＃電文保留用トランザクション ID
     ;
     %CLASS
         NAME      =  クラス名
         RESTTXID  =  CORESTTXID001              ＃電文保留用トランザクション ID
     ;
 ;
```

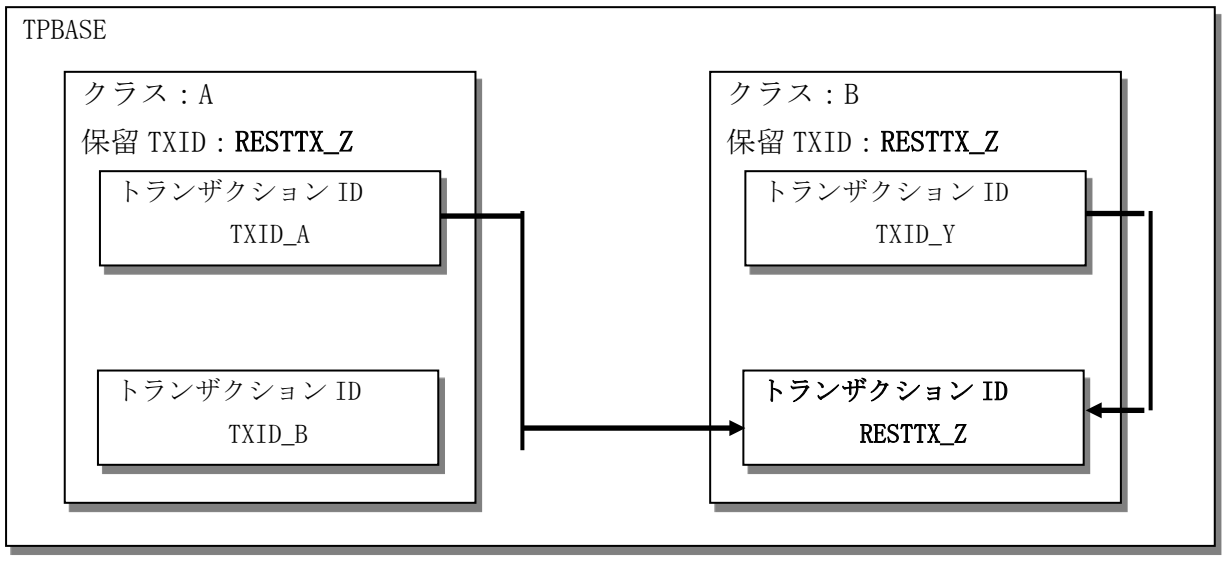
(b)　(2) 電文保留用トランザクション ID の定義

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        ＃電文保留用トランザクション ID
     ;
     %TRANS
         NAME      =  CORESTTXID001        ＃トランザクション ID(保留用とする)
         CONAME    =  CONAME001            ＃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          ＃電文保留用トランザクション ID
     ;
     %TRANS
         NAME       =  TXID_A              ＃トランザクション ID
         CONAME     =  CONAME001         ＃CO 名
     ;
     %TRANS
         NAME       =  TXID_B              ＃トランザクション ID
         CONAME     =  CONAME001         ＃CO 名
     ;

     %CLASS
         NAME       =  B
         RESTTXID   =  RESTTX_Z          ＃電文保留用トランザクション ID
     ;
     %TRANS
         NAME       =  TXID_Y              ＃トランザクション ID
         CONAME     =  CONAME001         ＃CO 名
     ;
     %TRANS
         NAME       =  RESTTX_Z          ＃トランザクション ID(保留用とする)
         CONAME     =  CONAME001         ＃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 AP ノード、TPBASE 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 ID、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)　CO 名指定、または TXID 指定

(b)　宛先 TPBASE モニタを固定したい場合は、論理ノード名、TPBASE モニタ名指定可

AP ノード上で電文保留登録する。(図中③)

(c)　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))

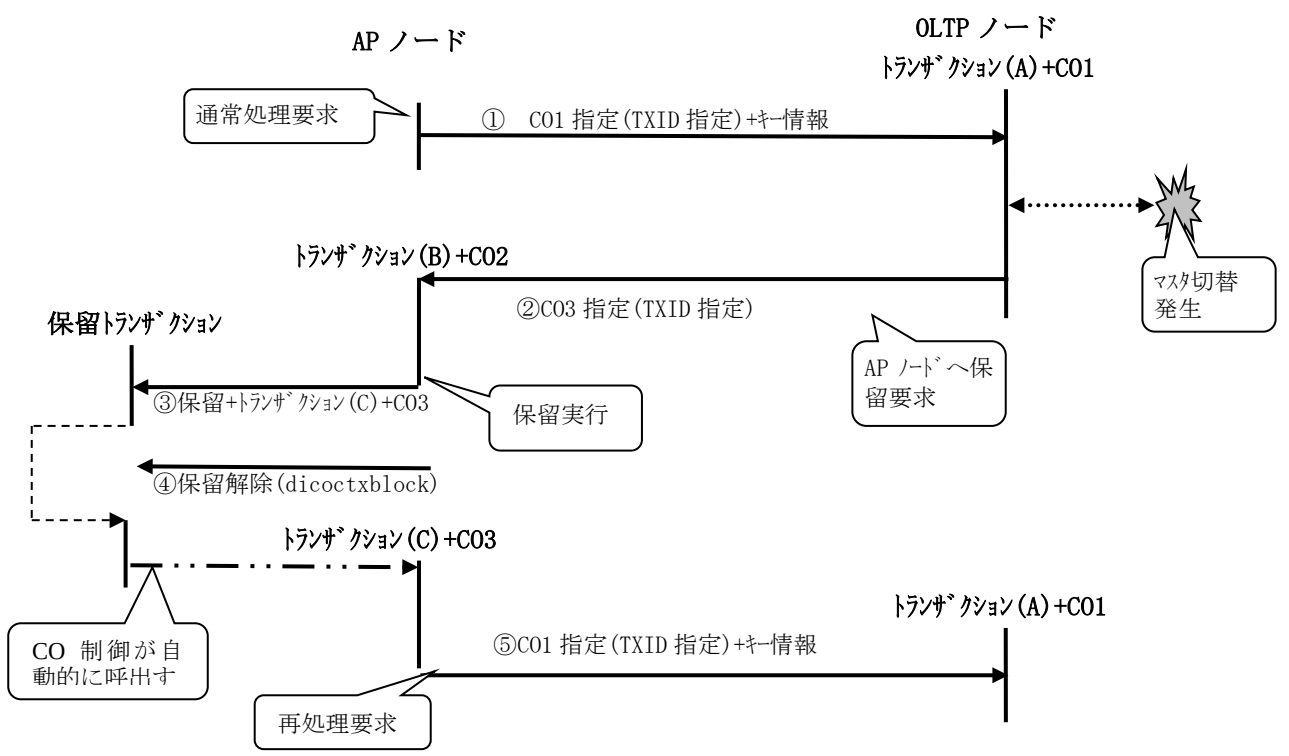
(d)　電文保留解除後実行する CO 名(CO3)

電文保留解除後、再処理を実行する。(図中⑤)

(e)　キー情報、③の電文受信 API(diosarecvtx)で取得したキー情報を電文送信 API(diosasendtx)に設定する。

(f)　再処理する CO 名指定、または TXID(トランザクション(A))

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 DB 情報(diosadcuca- Db)を設定します。このキー情報を与えることで diosa は IM の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 が持ち回る情報で、電文受信 API(diosarecvtx)で参照することが可能です。保留解除後の再処理の際、電文送信 API(diosasendtx)の DB 情報(diosadcuca- Db)として IM のキー情報を再設定することにより、マスタノードを再選択後、電文送信ができます。

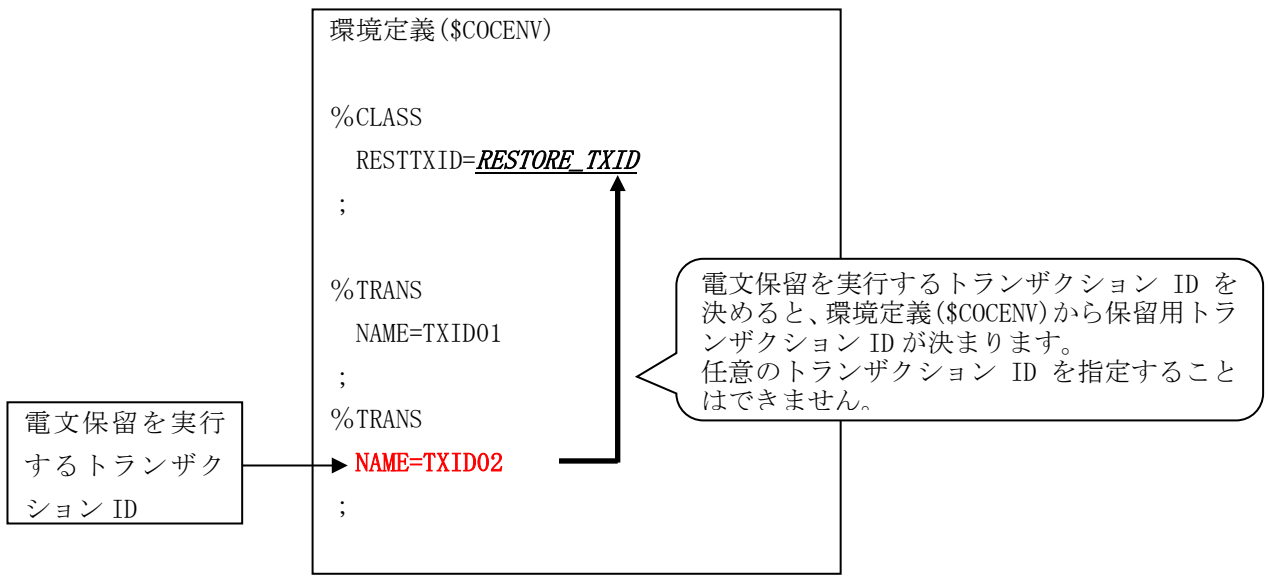
(7)　　電文保留実行情報の決定

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 CO 名です。

AP ノードと TPBASE モニタが決定すると、今度は TPBASE 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 CO が呼び出されます。この CO が実際に保留トランザクションに電文を登録します。

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



(8)　　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。CO 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)

dicoctxblock 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。dicoctxblock 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 dicoctxblock 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　　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。

CO 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 CO 制御サーバとは別の CO 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。

3.1.6　　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。

3.1.7　　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。

3.1.8　　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。

3.1.9　　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"
```

3.1.10　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)　CO タイマ登録

CO 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)、CO 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。

※CO 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 diosacommit 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;
     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 ) );

     /* CO タイマ登録 インターバル型:15 秒×10 回 */
     snprintf( TmcUca.TimerId, 17, "sample_TMC_01" );
     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, "sample_CO99" );
     snprintf ( CosetUca.TxId, 7, "TR0101" );
     CosetUca.Textalen = ( short )strlen( Data );

     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(日付あり)

時刻形式に DIOSA_CLOCK を設定し、タイマ値に日付、時刻を設定した場合、指定日時に 1 回だけ実行されるタイマを登録することができます。

利用例

```
#include <diosa.h>

int SAMPLESET(){

    int              Ret;
    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 ) );

    /* CO タイマ登録 時刻指定(日付あり)型 */
    snprintf( TmcUca.TimerId, 17, "sample_TMC_02" );
    TmcUca.Mode     = DIOSA_CLOCK;
    TmcUca.KeyCheck = DIOSA_OFF;

    TmcUca.Time.Year   = 2020;
    TmcUca.Time.Month  = 12;
    TmcUca.Time.Day    = 31;
    TmcUca.Time.Hour   = 12;
    TmcUca.Time.Minute = 0;
    TmcUca.Time.Second = 0;

    snprintf ( Data, 200, "CO_TEST_MSG" );

    snprintf ( CosetUca.CoName, 31, "sample_CO99" );
    snprintf ( CosetUca.TxId, 7, "TR0101" );
    CosetUca.Textalen = ( short )strlen( Data );

    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;
}
```

(c)　時刻指定(日付なし)

時刻形式に DIOSA_CLOCK を設定し、タイマ値に時刻のみを設定した場合、指定時刻に毎日実行されるタイマを登録することができます。

利用例

```
#include <diosa.h>

int SAMPLESET(){

    int              Ret;
    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 ) );
```

```
        /* CO タイマ登録 時刻指定(日付なし)型 */
        snprintf( TmcUca.TimerId, 17, "sample_TMC_03" );
        TmcUca.Mode     = DIOSA_CLOCK;
        TmcUca.Count    = 10;
        TmcUca.KeyCheck = DIOSA_OFF;

        TmcUca.Time.Year   = 0;
        TmcUca.Time.Month  = 0;
        TmcUca.Time.Day    = 0;
        TmcUca.Time.Hour   = 23;
        TmcUca.Time.Minute = 30;
        TmcUca.Time.Second = 30;

        snprintf ( Data, 200, "CO_TEST_MSG" );

        snprintf ( CosetUca.CoName, 31, "sample_CO99" );
        snprintf ( CosetUca.TxId, 7, "TR0101" );
        CosetUca.Textalen = ( short )strlen( Data );

        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、diosatmcreset 関数を利用します。diosatmcreset 関数を利用する際には、タイマの制御情報(t_diosa_tmcuca)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、diosatmcreset 関数に加えて diosacommit 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 CO が受信する電文形式

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

このとき CO 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

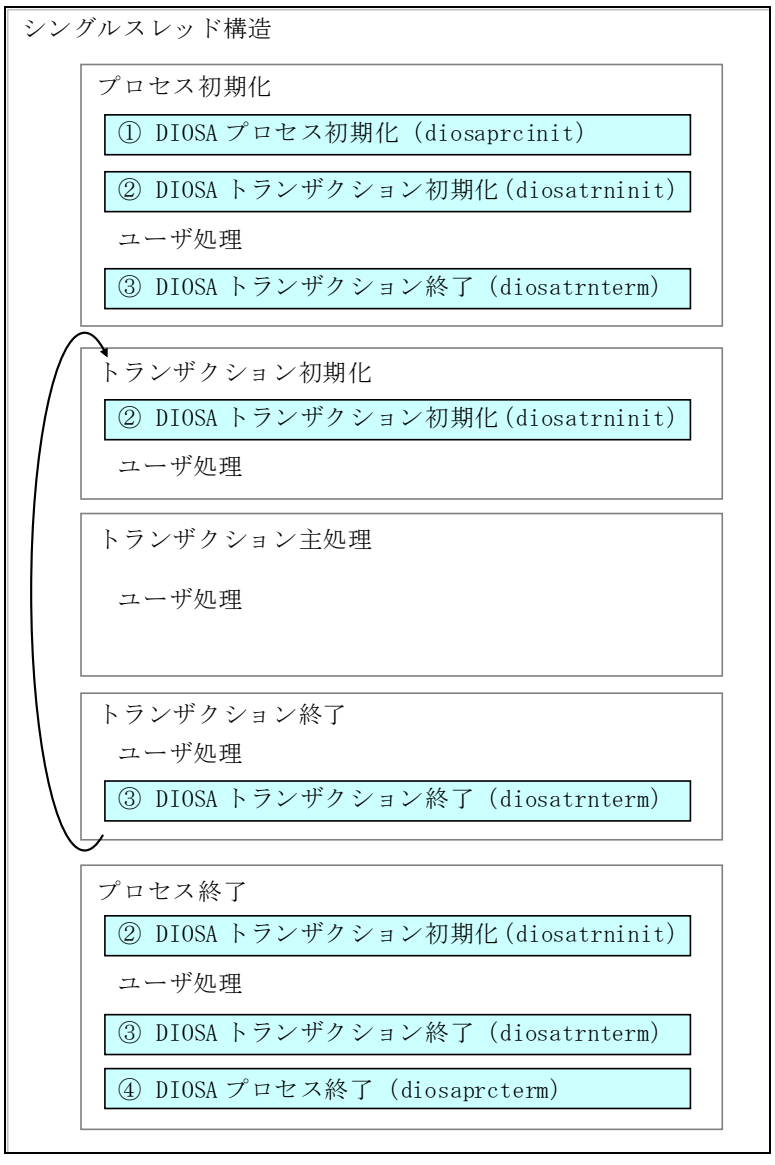
本章では、ユーザアプリケーションの作り方について説明します。

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。

DIOSA の各種 API は、本 DIOSA トランザクション初期化から DIOSA 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 DIOSA トランザクション終了

DIOSA 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 API を実行します。なお、本 DIOSA トランザクション終了後に DIOSA の各種 API を使用する場合は、再度 DIOSA 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 DIOSA トランザクション終了を実行してください。

④ DIOSA 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 DIOSA 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、DIOSA プロセス終了処理も呼び出さないようにしてください。

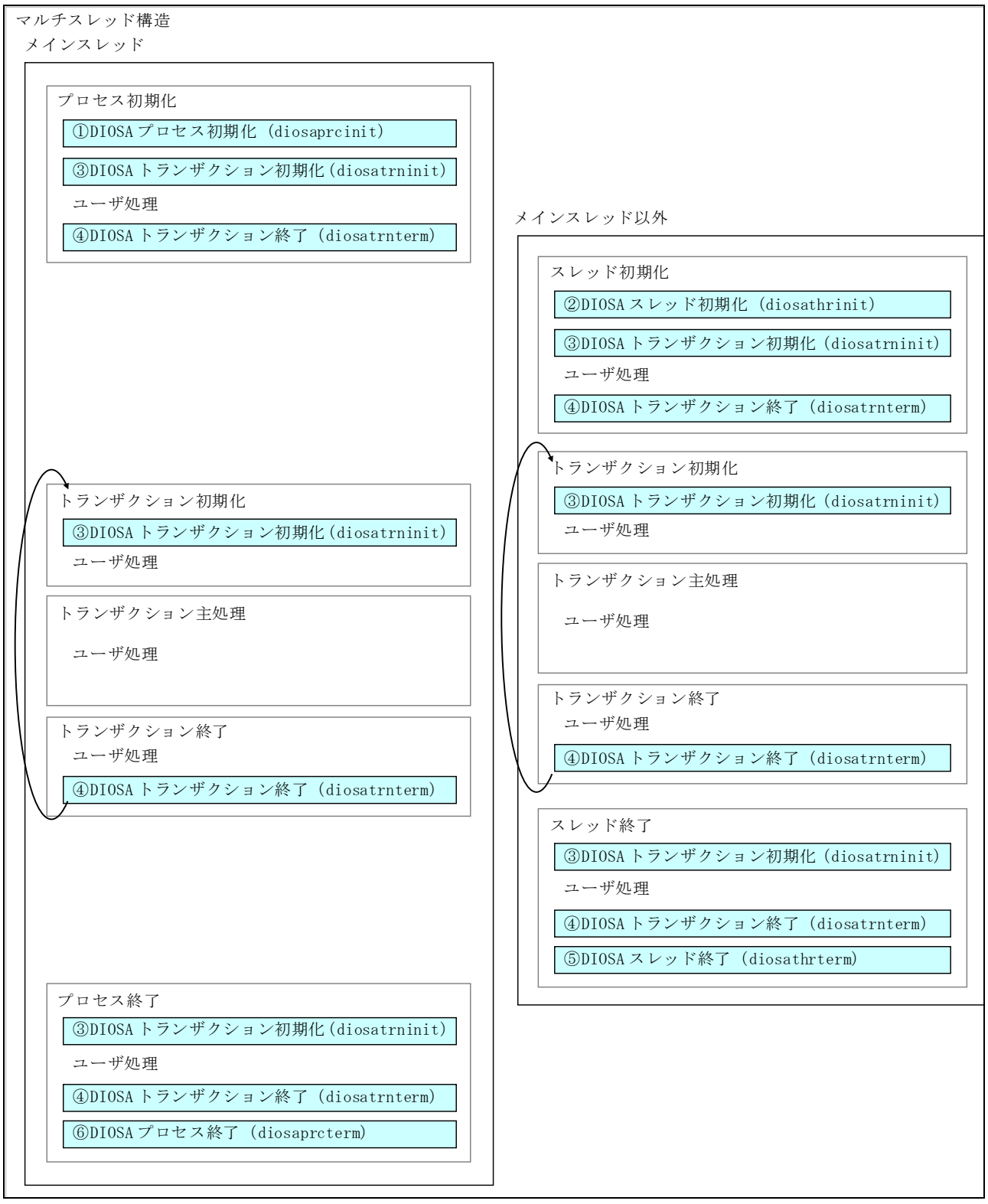
プロセス終了後、再度プロセス初期化処理を呼び出して処理を継続することはできません。

(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 DIOSA プロセス初期化が行われている必要があります。

③ DIOSA トランザクション初期化

DIOSA プロセス初期化や DIOSA 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 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。

DIOSA の各種 API は、本 DIOSA トランザクション初期化から DIOSA 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 DIOSA トランザクション終了

DIOSA 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 API を実行します。なお、本 DIOSA トランザクション終了後に DIOSA の各種 API を使用する場合は、再度 DIOSA 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 DIOSA トランザクション終了を実行してください。

⑤ DIOSA スレッド終了

スレッドの最後に実行します。

⑥ DIOSA プロセス終了

プログラムの最後に実行します。なお、DIOSA 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 DIOSA 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 DIOSA 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、DIOSA 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

電文種別決定出口を使用するためには、環境定義 SYSMAP 節 PROTOCOL 項で出口関数名と出口関数を含むライブラリ名を定義し、DEFAULT 項・LOGSYSTEM 項・ACCESSPOINT 項のいずれかで PRTOCOL を指定します。

(2)　呼出契機

論理システム間通信用リスナが電文を受信した場合、もしくは都度接続送信デーモンの応答電文受信時に呼び出されます。

(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及び電文送信元の情報を与えます。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 電文種別決定出口名(void *RfuUca, t_diosa_getmsgtypeuca *GetMsgTypeUca)

void *RfuUca（入出力型）
将来拡張領域。現在 RfuUca には NULL が格納される。

t_diosa_getmsgtypeuca *GetMsgTypeUca（入出力型）
t_diosa_getmsgtypeuca 構造体により情報の授受を行う。
t_diosa_getmsgtypeuca 構造体は以下のメンバを含んでいる。
void *MsgPtr          電文格納領域へのポインタが格納される。(入力)
int  LdDtLen          電文格納領域の長さが格納される。(入力)
int  Complete         ヘッダ部のみを読み込んだ場合は DIOSA_NO(0)が、
                      電文全てを読み込んだ場合は DIOSA_YES(1)が格納される。(入力)
char LsName[15+1]     電文送信元の論理システム名(入力)
char AcsPoint[15+1]   電文送信元のアクセスポイント名(入力)
char Protocol[31+1]   アクセスポイントのプロトコル名(入力)
char Term[32]         電文を受信した端末名(入力)
char UserData[64]     TPBASE 端末定義ファイル中の USERDATA に設定された値(入力)
int  MsgLen           Complete が DIOSA_NO(0)の場合に、総電文長を返却する。(出力)
char TxId[7]          出口が実行されるノードで起動するトランザクション ID を
                      利用者が格納する。(出力)
```

(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 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もっていなかった場合、総電文長を MsgLen に指定して DIOSA_RETRY で電文種別決定出口終了し、電文種別決定出口を再実行します。DIOSA_RETRY は下位プロトコルが TCP/IP の場合にのみ有効です。
トランザクション ID を決定できるだけの長さをもっていた場合、③の処理に進みます。

③ 受信電文を解析し、起動するトランザクション ID(TxId)と総電文長(MsgLen)を指定して DIOSA_DONE で電文種別決定出口を終了します。

トランザクション ID および総電文長を決定できない場合は、電文を破棄して終了します。

(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バッファはバイト境界が正しいことを保証しません。
下位プロトコルが MQ の場合、本出口は呼び出されません。

3.3.2 電文種別決定初期化出口

電文種別決定出口で必要となるリソースの確保を行うための利用者出口を提供します。

(1) 使用方法

電文種別決定初期化出口を使用するためには、環境定義 SYSMAP 節の DEFAULT 項か LOGSYSTEM 項のいずれかで出口関数名と出口関数を含むライブラリ名を定義します。

(2) 呼出契機

論理システム間通信用リスナ、あるいは都度接続送信デーモンが起動するときに呼び出されます。

(3) プログラムインタフェース

リスナ初期化出口から呼び出されます。

```
int 電文種別決定初期化出口名(void *RfuUca)

void *RfuUca（入出力型）
将来拡張領域。現在 RfuUca には NULL が格納される。
```

(4) プログラム作成方法

電文種別決定出口で必要となるリソースの確保をします。

(例えば環境変数の取得やファイル読み込みなど)

3.3.3　電文種別決定終了出口

電文種別決定出口で必要となるリソースの解放を行うための利用者出口を提供します。

(1)　使用方法

電文種別決定終了出口を使用するためには、環境定義 SYSMAP 節の DEFAULT 項か LOGSYSTEM 項のいずれかで出口関数名と出口関数を含むライブラリ名を定義します。

(2)　呼出契機

論理システム間通信用リスナ、あるいは都度接続送信デーモンが終了するときに呼び出されます。

(3)　プログラムインタフェース

リスナ終了出口から呼び出されます。

```
void 電文種別決定終了出口名(void *RfuUca)

void *RfuUca（入出力型）
将来拡張領域。現在 RfuUca には NULL が格納される。
```

(4)　プログラム作成方法

電文種別決定初期化出口で確保したリソースの解放処理等を行います。

3.3.4　　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）
```

(4)　　注意事項

データベースへ接続する際、接続は成功するが、ステータス・コード(sqlcode)に警告が設定される場合があります。例えば「パスワード期限が近づいている」場合に警告が返却されます。

ステータス・コードによりデータベースへの接続が正常に行われたかを判断し、データベース接続出口の戻り値を決定してください。

3.4　アプリケーションの生成

3.4.1　CO プログラム

メイン CO および下位層の CO・BO、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 CO・BO 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

CO・BO、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
-pthread –fPIC	POSIX スレッドが使用可能となるように指定します。
-I /opt/diosa_xtp/include	DIOSA/XTP インクルードファイル格納位置を指定します。

以下にコンパイル実行例を記載します。

```
# cc -c -pthread -fPIC -I /opt/diosa_xtp/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 -shared -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
-pthread –fPIC	POSIX スレッドが使用可能となるように指定します。
-I /opt/diosa_xtp/include	DIOSA/XTP インクルードファイル格納位置を指定します。

以下にコンパイル実行例を記載します。

```
# cc -c -pthread -fPIC -I /opt/diosa_xtp/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 /opt/diosa_xtp /lib	DIOSA/XTP ライブラリファイル格納位置を指定します。
-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
-pthread –fPIC	POSIX スレッドが使用可能となるように指定します。
-I /opt/diosa_xtp/include	DIOSA/XTP インクルードファイル格納位置を指定します。

以下にコンパイル実行例を記載します。

```
# cc -c -pthread –fPIC -I /opt/diosa_xtp/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 /opt/diosa_xtp/lib	DIOSA/XTP ライブラリファイル格納位置を指定します。
-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
```

DIOSA/XTP V2.1
利用の手引

2017 年 3 月 3 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)