SteelEye LifeKeeper for Linux
GenericARK 開発ガイド

rev1.2

サイオステクノロジー株式会社

更新履歴

日付	版数	更新内容
2009/09/04	1.0	初版
2009/09/09	1.1	細部の修正を行いました。
2013/03/07	1.2	細部の修正を行いました。

はじめに

LifeKeeper for Linux では、Application Recovery Kit(ARK)が対応していないアプリケーションを保護する仕組みとして、GenericARK を用意しています。GenericARK は、ユーザ自身の手で制御スクリプトを作成し、GenericARK リソースとして LifeKeeper に組み込み、LifeKeeper の保護対象とします。

本書では、実際のスクリプトの中で検討するべき事項や、LifeKeeper の動作上の特性との連携について、サンプルコードを交えてご案内いたします。

本文書の内容、また対象となる読者は以下の前提に従います。

- 本文書の内容は、LifeKeeper for Linux テクニカルトレーニングを受講したか、または同等程度の知識をお持ちであることを前提に書かれています。
- 本文書において、GeneriARK リソースとして保護する対象は、一般的なサーバアプリケーション(コマンドラインから起動/停止の制御が可能であり、クライアントに対して何らかのサービスを提供するもの)を想定しています。
- 記載内容は LifeKeeper for Linux v7 以降を対象とします。
- サンプルコードは bash によるシェル・スクリプトを使用します。

❑ 目次

1.表記上の約束について

本書では数種の表記上の約束を採用しています。

- *Italic* や**太字**は強調などに使用しています。

- typewriter はターミナルやコンソール上画面に表示されるシステムからのメッセージまたはプロンプトを表します。

```
Configuration successfully completed.
```

- ターミナルやコンソール上画面に使用されるプロンプトは以下の以下のプロンプトを表します。

  例:コマンドがスーパーユーザーで実行される場合

```
#
```

  例:コマンドが一般ユーザーで bash または ksh で実行される場合

```
$
```

- 文章中の【】に囲まれている単語は、ユーザが入力するキーおよびGUI上での操作を表します。

  例:【Return】キーを入力してください。

2.GenericARK の仕様

GenericARK のスクリプトに記述する内容について、LifeKeeper は特に制限を設けておりません。しかし、保護対象のアプリケーションをユーザの期待通りに制御するためには、GenericARK の仕組みや LifeKeeper の仕様についての理解が必要です。

ここでは、GenericARK リソースを作成する際に必要となる LifeKeeper や GenericARK の仕様についてご紹介します。

❒ フェイルオーバのトリガーと、切り替わりの動作

LifeKeeper において、フェイルオーバ(障害検知によるリソース切り替え)のトリガーとなる事象は、大きく分けて以下の 2 種類があります。

1. 各リソースの quickCheck により障害を検知し、待機ノードにフェイルオーバを行う。
2. ハートビートの切断により相手ノードの障害を検知し、待機ノードにフェイルオーバを行う。

項目 1 の「各リソースの quickCheck によるフェイルオーバの発生」では、以下のようにリソースの制御が行われます。

1. quickCheck スクリプトにより、障害を検知する。
2. recover スクリプトにより、障害を検知したノード上でリソースの再起動を行う。
3. recover でサービスが回復しなかった場合は、recover の失敗となり、待機ノードへのフェイルオーバの実行を決定する。
4. 障害を検知したノード上で、フェイルオーバの対象となる全リソースの remove(停止)が行われる。
5. 上記 4 の停止処理が完了した後、フェイルオーバ先のノード上で、必要なリソースの起動が行われる。

また、項目 2 の「ハートビート切断によるフェイルオーバ」では、以下のような制御となります。

1. 待機状態のノードからハートビートの切断を検知し、待機ノードへのフェイルオーバの実行を決定する。
2. フェイルオーバ先のノード上で、必要なリソースの起動を行う。

※ LifeKeeper for Linux では、常にどちらか片方のノードが共有ディスクにロックを掛けるため、ハートビートの切断が発生してもスプリットブレイン状態(複数ノードでサービスが有効となる状態)は発生いたしません。

このように、フェイルオーバ時の動作は、リソース障害に起因したものか、ハートビート障害に起因したものかによって、切り替わり時の動作が変わります。特に、**リソースの停止処理が含まれるかどうかは大きな違いです。**

❒ 各スクリプトのタイムアウト

GenericARK は登録されたスクリプトを自動的に実行しますが、quickCheck 以外のスクリプト(restore、remove、recover)に対しては、LifeKeeper はタイムアウトの仕組みを提供していません。

例えば、restore スクリプト内で実行した、アプリケーションの起動コマンドがハングアップ状態となった場合、restore スクリプトはこの起動コマンドの完了待ちとなります。restore には自動的なタイムアウトの仕組みがないため、restore スクリプトはいつまでも待ち状態を維持します。これは remove や recover でも同様です。

この問題を回避する手段としては、restore、remove、recover の各スクリプト内に、タイムアウトの仕組みを実装する方法が考えられます。以下はタイムアウトの構造のサンプルです。

サンプルコード 1:タイムアウトの実装

```
#!/bin/bash

. /etc/default/LifeKeeper

handlealarm()
{
       # INT を受け取った後の停止処理を記述します。この例では、自分自身に対して
       # TERM/KILL を送信しています。
       LANG=C; date; echo "Script was hung. Forcibly terminating."

       # ここにタイムアウトが発生した場合に、必要となる後処理を記述してください。

       if [ "$(ps $$ > /dev/null 2>&1; echo $?)" -ne 1 ]
       then
              kill -TERM $$
              sleep 5
              if [ "$(ps $$ > /dev/null 2>&1; echo $?)" -ne 1 ]
              then
                   LANG=C; date; echo "Script was still hung. sending KILL signal"
                   kill -KILL $$
              fi
       fi
       exit 1
}

# 自身で受け取った INT を trap し、handlealarm()を実行します。
trap "handlealarm" INT

# $RESTORETIMEOUT 秒後に、自身の PID に対して INT を発行します。
setalarm()
{
       (sleep $1; (ps $$ | grep $0) && kill -INT $$; (ps $$ | grep $0) && sleep
$1; (ps $$ | grep $0) && kill -KILL $$) &
}

#タイムアウト秒の指定です。これは/etc/default/LifeKeeper に指定しています。
setalarm $RESTORETIMEOUT

#以下に実際の処理内容を記述します。正常に完了した場合は、戻り値 0 を
#返します。

exit 0
```

上記の 11 行目に「# ここにタイムアウトが発生した場合に、必要となる後処理を記述してください。」という記載があります。ここには、タイムアウトとなった時点で実行されている起動コマンドや、チェックのコマンドの停止処理などを記述します。各スクリプトによって、どのような停止処理が必要となるかは異なりますが、タイムアウトによる強制停止の前処理として、必要な処理を書き加えてください。

quickCheck に関しては、スクリプトの開始から終了までのタイムアウトを LifeKeeper で管理することができます。この機能の詳細は、後述の**「5.監視(quickcheck)スクリプトの実装」**をご参照ください。

❒ タイムアウト発生時の戻り値の作成

上記のサンプルの通りにタイムアウトの仕組みを作成した場合は、スクリプトを実行しているプロセス自身が TERM、または KILL シグナルで停止することになります。これらのシグナルを受けてシェルが終了した場合は、0 以外の戻り値が返ります。

GenericARK スクリプトでは、終了時の戻り値を 0 か 1 に集約するようご案内しておりますが、このようなシグナルによる終了の場合は、0 か 1 以外の戻り値となる場合があります。LifeKeeper の仕様としては、戻り値 0 が正常終了、0 以外はエラー終了となりますので、TERM や KILL での終了は、エラー終了として扱われます。

restore、recover については、タイムアウトの発生をエラー扱いとすることは妥当と言えます。remove、quickCheck のタイムアウトをエラーとするかは、検討が必要です。

例えば、切り替え操作の最中に remove のタイムアウトが発生した場合、それをエラーと判断すると、そこで切り替え操作は終了します。また、quickCheck のタイムアウトは、LifeKeeper の他の ARK ではエラーとはしていません。

タイムアウトとなってもエラーとしない実装とする場合は、上記の「サンプルコード 1:タイムアウトの実装」の 11 行目「# ここにタイムアウトが発生した場合に、必要となる後処理を記述してください。」の部分に必要な後処理を記述した後、”exit 0”で正常終了としてください。

❒ **エラー時のリトライ**

GenericARK には、restore，remove，quickCheck，recover の各スクリプトの実行において、自動的にリトライを行うことはありません。スクリプトを実行結果に応じてリトライを行う必要がある場合は、スクリプトの内部にリトライの仕組みを作成する必要があります。

※サンプルコードが「3-3. スクリプトの戻り値の作成」にあります。参考としてください。

❒ **仮想 IP アドレスと仮想ホスト名**

通常、HA クラスタを構成した場合、サービスにアクセスするクライアントは、仮想 IP アドレスか、仮想 IP アドレスにバインドされた仮想ホスト名に対してアクセスを行います。

GenericARK で保護するアプリケーションにおいても、仮想 IP アドレスに紐づいた仮想ホスト名が必

要となるケースがあります。そのような設定が必要となるかどうかは、事前にアプリケーション側の要件を確認しておくことをお勧めいたします。

仮想 IP アドレスに紐付く仮想ホスト名が必要である場合は、/etc/hosts や DNS など、通常の名前解決のシステムで対応してください。

❒ デバッグ情報の収集

各スクリプトに、デバッグ情報を収集する機能を付加しておくことをお勧めいたします。本ガイドのように、シェルスクリプトで作成するのであれば、先頭付近(#!/bin/bash の直後)に”set -x”を入れておくことで、シェルの動作の詳細ログを収集することが出来ます。

但し、LifeKeeper のログの最大サイズは有限です。初期状態で 1Mbyte までの情報を格納でき、それを超えた分は古いものから上書きされます。restore や remove であれば、滅多に実行されることは無いと考えられますが、quickCheck は 2 分に一度(初期値)の頻度で実行されるため、1 回あたりのログ出力量次第では、すぐにログを消費してしまう可能性もあります。そのため、通常時はログに情報は記録せず、一時的なファイルとしてログをプールしておき、エラーが発生した場合のみログを出力し、それ以外は破棄するなど、ログが無限に増加しないための工夫が必要です。

3.起動(restore)スクリプトの実装

restore スクリプトにおいて、一般的なアプリケーションを保護する際の留意点についてご説明します。restore スクリプトを作成する場合、以下の点について注意が必要です。

- ❒ 起動対象のサービスが、既に起動状態である場合への配慮
- ❒ 前処理の実行
- ❒ 起動処理および起動後の状態確認
- ❒ 最終的な戻り値の作成
- ❒ リトライの制御
- ❒ restore のタイムアウトの作成

3-1. 起動対象のサービスが、既に起動状態である場合への配慮

LifeKeeper は起動の際に、最後に停止した時のステータスを再現するべく、必要に応じてリソースの起動を行います。OS の起動に伴う LifeKeeper の自動起動や、”lkstart”で LifeKeeper を単体で起動した場合でも同様の動作になります。

ここで注意しなければならないのは、”lkstop -f”コマンドで LifeKeeper を停止した場合です。”lkstop -f”コマンドは、稼動中のサービスを停止せずに、LifeKeeper のプロセスのみを停止します。このコマンドは、主に LifeKeeper のアップデートなどのメンテナンスを行う際に利用します。

“lkstop -f”コマンドで LifeKeeper を停止した後、”lkstart”コマンドで LifeKeeper の起動を行った場合は、LifeKeeper は各リソースの restore スクリプトを実行します。これは”lkstop -f”コマンドを実行する直前のステータスが「起動中(ISP)」であることが理由です。

製品として発売している各リカバリキットにおいては、”lkstop -f”後の”lkstart”で、保護対象アプリケーションの 2 重起動が発生しないように、配慮されています。多くのリカバリキットでは、既に起動中のアプリケーションの存在を確認した場合、特に何もせずに正常終了します。GenericARK リソースにおいても、”lkstop -f”の運用が必要であるならば、製品の ARK と同様に、2 重起動への配慮が必要です。

起動状態の確認は quickCheck で使用している監視コマンドと同じものを使用されることをお勧めします。(これは quickCheck そのものではなく、同スクリプトの中で、アプリケーションの監視を行っているコマンドという意味です)状況によって監視方式を変更すると、「restore には成功するのに、直後の quickCheck ではエラーとなる」というような問題の原因となり、切り分

けが困難になる場合があります。

3-2. 前処理の実行

実際のアプリケーションの起動を行う前に、必要な前処理を実行してください。どのような前処理が必要になるかは、アプリケーションによっても異なりますが、LifeKeeper の GenericARK としてスクリプトを作成するのであれば、以下のような点について注意する必要があります。

➔ 強制停止状態からのクリーンアップ処理

LifeKeeper は、共有ストレージを使用している環境において、全ハートビートの切断やストレージ障害への対応として、稼動中のノードの強制再起動を行うことがあります。この場合保護対象アプリケーションの停止処理は実行されません。アプリケーションによっては、次回のアプリケーションの起動前に、クリーンアップが必要となる可能性があります。

➔ remove のタイムアウトが発生した場合への対処

アプリケーションの挙動やスクリプトの実装にもよりますが、remove がタイムアウトとなった場合は、正しく停止処理が行われない可能性があります。そのような事態が想定される場合は、restore スクリプトのアプリケーション起動の前処理として、クリーンアップ処理が必要となる場合があります。

3-3. 起動処理および起動後の状態確認

保護対象の起動コマンドを実行し、アプリケーションの起動を行います。また起動後に、正しく起動出来たかどうかを restore スクリプトの中で確認し、起動の成功、失敗を明確にすることをお勧めします。

なお、起動コマンドの失敗や、起動後の確認において失敗が検知された場合は、不自然な状態でプロセスなどが残っている可能性も考えられます。後のスイッチバック時の問題とならないよう、停止処理を実行しておくなどの配慮が必要です。

3-4. スクリプトの戻り値の作成

起動コマンドの戻り値を、そのまま restore スクリプトの戻り値とするのではなく、起動コマンドの戻り値を確認した上で、改めてスクリプト自体の戻り値として、0(正常)/1(失敗)のいずれかの値を返すように作成してください。以下は戻り値作成のサンプルです。

サンプルコード 2:戻り値の作成

```
# パラメータ設定
APP="アプリケーション名"
APP_START="起動コマンド"
VHOSTNAME="仮想 IP アドレスに紐付いた論理ホスト名"
APP_CHECK1="監視コマンド"

# 関数の作成
APP_start()
{
echo "$(date +'[%Y/%m/%d %H:%M:%S]') : $APP starting on $HOSTNAME"
$APP_START $VHOSTNAME >> $lklog 2>&1
}

APP_check1()
{
$APP_CHECK1 $VHOSTNAME >> $lklog 2>&1
ret1=$?
}

# 起動コマンドの実行
APP_start

# ステータスが確定するまでリトライを繰り返します。
while true
do
        # APP_check1() でアプリケーションの状態を確認しています。
        APP_check1

        # 待ち時間
        sleep 5

        # APP1_check により、$ret1 にはアプリケーションの状態確認の戻り値が格納されて
        # います。このアプリケーションは、戻り値 0 が正常、12 で未確定 (要リトライ) 、その他の
        # 戻り値はエラー終了となります。

        case "$ret1" in
                0) echo "$(date +'[%Y/%m/%d %H:%M:%S]') "APP" : ¥
                        START UP COMPLETE on $HOSTNAME" >> $lklog
                   exit 0 ;;
                12) continue ;;
                *) echo "$(date +'[%Y/%m/%d %H:%M:%S]') "APP":ERROR: ¥
                        START UP FAILED ret="$ret1"" >> $lklog
                   exit 1 ;;
        esac
done
```

上記の例では、while で無限ループを作成し、戻り値の確認を行っています。このリトライ処理は上記の APP1_check()の戻り値が 0 となるか、またはタイムアウトによる停止 (7 ページを参照) によって終了します。

また、上記の例ではログが$lklog に出力されます。起動、停止、再起動であれば、実行される頻度は多くなく、この処理が原因でログが圧迫されることも無いと思いますが、監視において同様の処理を入れる場合は注意が必要です。監視は 2 分に 1 度の頻度で実行されま

すので、監視の都度、ログを記録しておりますと、早期にログがいっぱいになってしまう可能性があります。監視スクリプトでは、コマンド実行の出力を/dev/null に捨てるか、または一時的にログとは異なるファイルに出力しておき、エラーであった場合はファイルの内容にログに記録、正常であった場合は、次回のチェック時にファイルの内容を上書きするといった処置が有効です。

3-5. リトライの制御

既にご案内している通り、GenericARK ではスクリプトのリトライを行いません。リトライが必要である場合は、スクリプト内でリトライの仕組みを作成する必要があります。上記の「サンプルコード 2:戻り値の作成」では、アプリケーションのステータス確認コマンドの戻り値に応じて、成功、失敗、リトライに処理を分けています。

3-6. restore のタイムアウトの作成

restore スクリプトにはタイムアウトの定義がないため、起動処理中に実行したコマンドがハングアップ状態となった場合などは、いつまでもコマンドの実行を待ち続けることになります。

このような動作を防ぐには、GenericARK スクリプトの中でタイムアウトの仕組みを作成する方法が有効です。具体的な仕組みは、6 ページの「サンプルコード 1:タイムアウトの実装」を参考にしてください。

4.停止(remove)スクリプトの実装

一般的なアプリケーションを保護する際の、remove スクリプトの留意点についてご説明します。

- ❒ 停止対象のサービスが、既に停止状態である場合への配慮
- ❒ 停止処理および停止後の状態確認
- ❒ 最終的な戻り値の作成
- ❒ リトライの制御
- ❒ remove のタイムアウトの作成
- ❒ remove の失敗/タイムアウトを、エラー終了とするかどうかの検討

4-1. 停止対象のサービスが、既に停止状態である場合への配慮

アプリケーションが停止している状態で、重ねて停止コマンドを実行するとエラーを返すアプリケーションもあります。そのため、remove スクリプト内で、停止コマンドを実行する前に、アプリケーションの状態を確認することをお勧めします。**対象のアプリケーションが停止状態である場合は、停止の成功と同様の扱いとし、正常終了として扱うことができます。**

4-2. 停止処理および停止後の状態確認

停止コマンドを実行し、アプリケーションの停止を行います。ここでは、以下の点について検討する必要があります。

① 停止が正常に行われなかった場合、そのアプリケーションには強制停止の方法があるか
② 停止コマンドを実行したあと、正常に停止したかどうかの確認は必要か

①については、そのアプリケーションが強制停止の方法を提供しているかどうか(または KILL などによる停止をサポートしているか)に依存します。②についても、やはりアプリケーションの動作に依存します。停止コマンドの後の確認が必要ないのであれば、実質的に remove スクリプトの失敗(停止失敗)は発生しないため、常に正常終了とすることが出来ます。

4-3. 最終的な戻り値の作成

前述の**「3-3. スクリプトの戻り値の作成」**でもご案内していますが、停止処理についても、remove スクリプト自体の戻り値として 0 か 1 のいずれかを返すよう作りこむ必要があります。

4-4. リトライの制御

停止処理においては、停止のリトライよりも強制停止などの手法を採用するケースが多いため、リトライ制御が必要となるケースは少ないかも知れません。停止コマンドの戻り値において、リトライの余地がある状況が存在するのであれば、リトライの仕組みも有効と言えます。

4-5. remove のタイムアウトの作成

前述の「**3-5. restore のタイムアウトの作成**」でもご案内しましたが、remove にもタイムアウトの仕組みはありません。停止処理のハングアップに備えなければならない場合は、スクリプト内でタイムアウトを実装してください。

また、タイムアウトとなった後の制御についても、検討が必要な事項と言えます。この点については、次項の「**remove の失敗/タイムアウトを、エラー終了とするかどうかの検討**」を参照してください。

なお、LifeKeeper には、remove の実行そのものに対するタイムアウト機能はありませんが、**リソース障害に起因するフェイルオーバにおいて、障害を検知したノードへの停止処理要求のタイムアウト**があります。この機能の詳細と対処については、「**7-1. リモート要求のタイムアウトについて**」を参照してください。

4-6. remove の失敗/タイムアウトを、エラー終了とするかどうかの検討

remove は対象アプリケーションの停止処理であるため、停止の成功/失敗は、サービスの継続性に対しては大きな影響を持ちません。**ただし、リソース障害検知によるフェイルオーバの場合は、先に稼動系ノードでのリソース停止が行われます。そのため、remove がエラー終了となった場合は、停止処理の失敗となりフェイルオーバそのものが失敗します。**

LifeKeeper の動作シナリオとして、多くの ARK は remove の失敗をエラー終了として扱います。そのため、フェイルオーバの際に remove の失敗が生じると、フェイルオーバそのものが失敗となります。この動作は、手動での切り替え(スイッチオーバ)であっても同様です。

停止コマンドの失敗後や、タイムアウト発生時の remove スクリプトの戻り値をあえて 0(正常終了)にすることで、**停止の失敗が生じた場合であっても、フェイルオーバを強行するという動作を実現することが出来ます。**但し、このような動作を行う場合は、以下の点について十分な注意を払ってください。

➔ ユーザが作成する GenericARK において、停止処理の失敗を正常終了の範囲と見なした場合であっても、LifeKeeper が提供する他の ARK においては、停止処理の失敗はエラ

一終了となります。リソース階層を構成する全てのリソースに対して、一様に停止処理の失敗を正常終了として扱わせることは出来ません。

➔ 停止処理の失敗を無視して切り替えを強行した場合、保護対象が参照するデータやアプリケーションそのものに、悪影響を与えることが無いかを確認する必要があります。

例えば、以下のようなリソース階層があるとします。

```
[GenericARK]
  + [仮想 IP リソース]
      + [ファイルシステムリソース]
```

上記の[GenericARK]リソースは、停止の失敗を正常終了の範囲として扱うよう作成したものです。このリソース階層で[仮想 IP リソース]に障害が発生した場合は、上位の[GenericARK]リソースから、順番に停止を行うことになります。

このケースでは、[GenericARK]リソースの内部の停止コマンドが失敗しても remove は戻り値 0 を返すため、停止処理は先に進みます。続けて[仮想 IP リソース]を停止し、[ファイルシステムリソース]のアンマウントを行います。

[ファイルシステムリソース]のアンマウントを行う際に、[GenericARK]リソースのプロセスが、保護対象のファイルシステムにアクセスしていた場合は、LifeKeeper は”fuser -k”で当該プロセスの KILL を行った上で、アンマウントを行います。これは LifeKeeper Core のファイルシステムリカバリキットの機能です。

このような強制停止は、アプリケーションやデータに対して悪い影響を与える原因となることがあります。この点について十分な検討が必要です。

➔ 次回のリソース起動時に、正しく起動できるかの確認が必要です。停止処理が失敗するということは、pid ファイルやフラグファイルなどが残存している可能性があります。

停止コマンドの失敗やタイムアウトが生じた場合でも、remove スクリプト自体は正常終了とする仕様を採用するのであれば、上記のような点に十分注意しなければなりません。少なくとも、内部的に発生した問題を検知できるような仕組みを作成し、後から管理者がチェックを行うといった運用を検討されることをお勧めします。

5.監視(quickCheck)スクリプトの実装

quickCheck スクリプトにおける、一般的な構造と留意点についてご説明します。

- ❒ LifeKeeper 上のステータスの確認
- ❒ 監視コマンドの実行
- ❒ 最終的な戻り値の作成
- ❒ リトライの検討
- ❒ quickCheck のタイムアウト

5-1. LifeKeeper 上のステータスの確認

LifeKeeper では、リソース監視は常に稼動中のリソース(LifeKeeper 上のステータスが ISP であるリソース)に対して行われます。待機状態のリソースに対しては、監視を行いません。

quickCheck を実行する際に、始めに確認しなければならないのは、このリソースは監視する必要性があるのかという点です。これを確認するためには、LifeKeeper の”ins_list”コマンドを使用します。以下は記述例です。

サンプルコード 3:リソースのステータス確認

```
STAT=$(/opt/LifeKeeper/bin/ins_list -t APP1 | cut -f7 -d"^A")

if [ $STAT != ISP ]
then
        echo "resource "APP1" is not ISP status"
        exit 0
fi
```

APP1 リソースのステータスを確認しています。ins_list コマンドは、リソースの詳細情報を画面に出力します。コマンド実行時のリソースのステータスは、7 番目のフィールドに格納されていますので、cut コマンドでその部分を切り出してください。なお、ins_list コマンドは、各情報のデリミタとして[^A]のメタ・キャラクタを使用しています([ctrl] + [v] , [ctrl] + [a])。

※ LifeKeeper は、LCD(構成情報 DB)にリソースのステータスを保持していますので、上記の記述が無くとも、ISP ではないリソースに対してチェックを行うことはありません。これはより安全性を高めるための記述です。

5-2. 監視コマンドの実行

アプリケーションが監視用のコマンドを用意しているのであれば、そのコマンドを実行してアプリケーションの状態を確認してください。また、一般的に状態確認に使用することのあるコマンドについて、以下に例をご紹介します。

- プロセスの存在確認
  # ps auxwww | grep <プロセス名> | grep -v grep
- プロセスおよび待ち受けポートの確認
  # netstat -anp | grep <port 番号やプロセス名など>
- プロセスが必要なファイルを開いているかの確認
  # lsof | grep <プロセス名など>
- ポートの状態確認
  # nmap -p <ポート番号> <仮想 IP アドレス>
  # nc <仮想 IP アドレス> <ポート番号>

※ lsof は、open しているファイルが多い場合はシステムに負荷を与えることがあります。

5-3. 最終的な戻り値の作成

監視処理についても、スクリプト自体の戻り値として 0 か 1 のいずれかを返すように作りこんでください。

5-4. リトライの検討

監視コマンドの実行結果が正常以外であった場合は、そのステータスに応じてリトライを行うケースも考えられますが、リトライを行った分だけ障害を判定するまでの時間が余計にかかることにもなります。監視コマンドの実行結果にて、リトライによる確認が必要な応答があった場合のみリトライを行うなどの検討が必要です。

5-5. quickCheck のタイムアウト

LifeKeeper for Linux v7 からは、quichCheck のタイムアウト値を/etc/default/LifeKeeper ファイルに定義できるようになりました。以下、GenericARK リソースの監視のタイムアウト設定についてご説明します。

5-5-1. LifeKeeper による quickCheck のタイムアウト

/etc/default/LifeKeeper ファイルに以下のパラメータを追加することで、quickCheck のタイムアウトを設定することが出来ます。

$TAG_TIMEOUT=秒

スクリプト内で実行した確認コマンドの応答が返らないといった場合、quickCheck スクリプトは応答待ち状態となります。quickCheck の実行から、上記のパラメータで指定した秒数が経過した時点で、quickCheck スクリプトは LifeKeeper により kill されます。

このタイムアウトは、明確にエラーを検知した訳ではないため、正常終了として扱われます。そのため、フェイルオーバのトリガとはなりません。

なお、上記の設定を行わなかった場合は、GenericARK の quickCheck には 22 秒のタイムアウトが適用されます。

また、このタイムアウトの発生の記録は、LifeKeeper のログ中に出力されるのみであり、SNMP TRAP やメール通知の対象イベントにはなりません。

5-5-2. 独自の実装による quickCheck のタイムアウト

監視のタイムアウト発生をフェイルオーバのトリガとする場合、またはタイムアウト発生時に実行したい処理があるといった場合は、LifeKeeper が用意する監視の仕組みではなく、restore や remove の項でご案内してきたような、スクリプト内でのタイムアウトの実装が必要です。(具体的な実装は次項でご説明します)

独自にタイムアウトの仕組みを作成した場合は、「**5-5-1. LifeKeeper による quickCheck のタイムアウト**」でご紹介した、LifeKeeper が標準的に用意しているタイムアウト設定との共存が必要になります。具体的な方法としては、**/etc/default/LifeKeeper ファイルの”$TAG_TIMEOUT=秒“には、スクリプト内で定義したタイムアウトよりも、長い秒数を設定してください。**

また、”$TAG_TIMEOUT=秒”として設定するタイムアウトの秒数は、/etc/default/LifeKeeper の LKCHECKINTERVAL パラメータで定義された値よりも、小さい値をセットしてください。LKCHECKINTEVAL は、LifeKeeper に登録されたリソース全体の監視サイクルであり、初期値は 120 秒です。個々のリソースのタイムアウト値は、このパラメータが示す値よりも小さくなければなりません。

各タイムアウト設定の指針についてまとめます。これらのタイムアウト値の長さは、以下のルールに沿って定義されていなければなりません。

①LKCHECKINTERVAL(もっとも長い)

- 全リソースの監視サイクルのタイムアウト定義であり、もっとも長い値を設定する必要があります。
- /etc/default/LifeKeeper の”LKCHECKINTERVAL=120 “(初期値)で定義します。

②$TAG_TIMEOUT=秒(①よりも短く、③よりも長い)

- LifeKeeper が用意する GenericARK 向けの quickCheck スクリプトのタイムアウト定義であり、LKCHECKINTERVAL よりも短く、かつユーザ定義のタイムアウトよりも長い値を設定します。
- /etc/default/LifeKeeper の”$TAG_TIMEOUT=秒”で定義します。

③ユーザ作成のタイムアウト(②よりも短い)

- ユーザが quickCheck スクリプト内に作成する、タイムアウト定義です。
- ②の LifeKeeper が用意する quickCheck スクリプトのタイムアウトよりも短い値でなければなりません。
- 作成方法は前述のサンプルをご参照ください。

5-5-3. quickCheck のタイムアウト後の取り扱い

quickCheck のタイムアウトを独自に実装した場合は、タイムアウトが発生した後の動作についての検討も必要です。以下は検討事項の例となります。

➔ <u>タイムアウトが発生した場合に、フェイルオーバのトリガーとするには</u>

タイムアウトの発生をフェイルオーバのトリガとするのであれば、前述の通り、スクリプト内に独自のタイムアウトの仕組みを用意し、タイムアウト後の処理の中で、エラー終了とする方法が有効です。タイムアウト後に戻り値 1 で終了することで、フェイルオーバのトリガとすることが出来ます。

なお、フェイルオーバの発生は、LifeKeeper は SNMP TRAP やメール通知の対象としております。この場合、指定した宛て先にメッセージ送信を行います。

➔ <u>タイムアウトの発生をエラーとはしないが、タイムアウトが発生したという情報は必要</u>

タイムアウトが発生した場合に呼び出される処理において、メール送信や SNMPTRAP の実行を記述する方法が有効です。以下はタイムアウト時に管理者にメールを送信する場

合のサンプルです。

サンプルコード 4:タイムアウト発生時のメール送信

```
handlealarm()
{
       LANG=C; date; echo "Script was hung. Forcibly terminating."

       # INT を受け取った後、mail コマンドで管理者宛にタイムアウトが発生した旨のメールを
       # 送信します。$TO_MESSAGE には、管理者に送るメールの本文を格納してください。
        echo $TO_MESSAGE | mail -s"$(date -R) timeout occurred in ¥
                           quickCheck for $TAG" <メールアドレス>

       if [ "$(ps $$ > /dev/null 2>&1; echo $?)" -ne 1 ]
       then
              kill -TERM $$
              if [ "$(ps $$ > /dev/null 2>&1; echo $?)" -ne 1 ]
              then
                   LANG=C; date; echo "Script was still hung. sending ¥
                                       KILL signal"
                   kill -KILL $$
              fi
       fi
       exit 1
 }
```

なお、システム負荷などにより、恒常的にタイムアウトのメッセージが出力されるような場合は、システム負荷の軽減を行うか、タイムアウトとなるまでの条件をゆるくするなどの配慮が必要です。

6.再起動(recover)スクリプトの実装

recover スクリプトは、quickCheck スクリプトが戻り値 1 で終了した後、ローカルノードでサービスの再起動を行い、復旧を試みる仕組みです。このスクリプトは、GenericARK リソースの作成において必須のものではありません。

ここでは、recover を作成する場合の留意点についてご説明します。

- ❒ recover の有無についての検討
- ❒ 制御コマンドの実行
- ❒ 最終的な戻り値の作成
- ❒ リトライの検討
- ❒ recover のタイムアウト

6-1. recover の有無についての検討

recover の実行で回復が見込めるのであれば、quickCheck によるエラーの検知からサービスの回復までの時間は、一般的にはフェイルオーバを行った場合よりも短くなります。しかし、recover では回復しない場合は、他のノードへ切り替えてしまった方が早期に復旧するといったケースも考えられます。

サービスの復旧にはどちらがより早いのかなどを検討の上、recover を作成するかを決定してください。

6-2. 制御コマンドの実行

アプリケーション側に再起動用の制御コマンドが用意されているのであれば、それを実行するのが最も簡単な手法となります。そのようなコマンドが用意されていないアプリケーションに対しては、スクリプトの中で再起動の仕組みを作成しなければなりません。

LifeKeeper では、殆どの ARK が recover を用意しています。recover の内容は ARK によって様々ですが、いわゆるサーバアプリケーション(DB やメール、ファイル共有など)においては、以下のステップで再起動を試みます。

1. アプリケーションの状態確認(quickCheck と同等)
2. アプリケーションの停止(remove と同等)
3. アプリケーションの起動(restore と同等)
4. 再起動後のアプリケーションの状態確認(quickCheck と同等)

上記の 4 の状態確認において、復旧が確認できない場合は、待機ノードへのフェイルオーバを行います。

6-3. 最終的な戻り値の作成

これまでに説明して参りました、他のスクリプトと同様に、スクリプト自体の戻り値を用意する必要があります。また、停止や起動コマンドそのものの失敗などがあった場合、その時点でエラー終了として戻り値 1 を返すといった制御とすることで、recover の失敗を早期に確定し、迅速に待機ノードへのフェイルオーバを開始するといった制御も可能です。

6-4. リトライの検討

他のスクリプトの同じく、GenericARK には recover のリトライを行う仕組みはありません。リトライが必要である場合は、スクリプトの内部でリトライの仕組みを作成する必要があります。

6-5. recover のタイムアウト

restore、remove と同じく、recover の実行に対するタイムアウトの仕組みはありません。タイムアウトが必要である場合は、これまでの例と同様に、スクリプト内に作成してください。また、タイムアウト後はエラー扱いとするのが妥当と考えられます。

7.その他の留意事項

restore、remove、quickCheck、recover の各スクリプトに特有のご説明は以上です。7 章では、GenericARK に関するその他の留意事項についてご紹介します。

7-1. リモート要求のタイムアウトについて

GenericARK に限ったトピックではありませんが、GenericARK スクリプトの、特に remove スクリプトに深い関わりがあります。

LifeKeeper はフェイルオーバの際に、フェイルオーバ先のノードから、フェイルオーバ元(障害発生ノード)に対して、リソースの停止を要求します。この要求には、応答時間のタイムアウトが設けられています。このタイムアウトの仕組みについて以下のリソース階層を例としてご説明します。

リソース階層	Server1 priority	Server2 priority
Generic APP	1	10
Virtual IP (障害部位)	1	10
Filesystem	1	10
Device	1	10
Disk	1	10

① 上記の例では、Virtual IP にて障害を検知しています。この後、Virtual IP リソースはローカルリカバリに失敗し、Server1 から Server2 にフェイルオーバを行うという動作を想定します。

② 障害となった Virtual IP は、Generic APP リソースを頂点としたリソースツリーに含まれています。そのため、フェイルオーバの対象となるリソースは Generic APP リソースとなり、その下位にある各リソースは、Generic APP リソースの起動に必要であるという理由で、フェイルオーバの対象になります。

③ Virtual IP リソースが障害となった後、Server2 では、頂点にある Generic APP が起動できるかを確認します(まだ実際の resotre スクリプトは実行しません)。この時点で、Generic APP リソースの起動には、Virtual IP～Disk までの各リソースの起動が必要であることを認識します。

④ しかし、障害部位である Virtual IP 以下の Filesystem～Disk の各リソースは、この時点では Server1 側で起動状態(ISP ステータス)のままです。そのため、Server2 は、Server1 に対して、Disk リソースの停止を命じます。

⑤ Diskリソースを停止するためには、Server1はDiskリソースに依存する上位の全リソースを停止しなければなりません。全てのリソースの停止が完了すると、Server1は停止が完了したことをServer2に通知します。

⑥ 停止完了の連絡を受けたServer2は、各リソースを下位のものから順番に起動を行います。Generic APPの起動が完了すれば、フェイルオーバは成功です。

この一連の流れの中で、**項番④にあるServer2からServer1への停止要求の実行から、⑤のServer1からServer2への停止要求の完了通知まで**の動作に対して、リモートノードに対する要求のタイムアウトがセットされています。

これは/etc/default/LifeKeeperに、REMOTETIMEOUTパラメータを用いて定義されています。このパラメータの初期値は300秒にセットされており、④の停止要求の発行から300秒が経過した時点で、「リモートからの停止要求がタイムアウトした」と判定されます。リモートからの停止要求がタイムアウトした場合は、フェイルオーバの失敗と判定されます。

removeのタイムアウトをスクリプト内に作成する場合は、この REMOTETIMEOUT によるタイムアウト時間も考慮する必要があります。**停止に長い時間がかかるようなアプリケーションの場合は、remove内のタイムアウトとともに、REMOTETIMEOUTの調整も検討しなければなりません。**

SIOS

サイオステクノロジー株式会社