*Trabalhando com serviço e alarme no Android*

# Resumo

Aplicativos no android têm a necessidade, assim como em qualquer outro tipo de sistema, de executar tarefas em determinado momento, estando em execução ou não. Essas tarefas estão relacionadas, por exemplo, à função de backup da aplicação, de uma sincronização de dados ou mesmo a apresentação de uma notificação ao usuário. Esse documento tem como objetivo apresentar a estrutura do Service e do Alarm que fornecem as funções para execução dessas tarefas. A importância desse estudo teve origem em uma solicitação de transmissão de dados a qualquer momento, ou seja, com a aplicação em execução (background ou foreground) e após o usuário matá-la.

# Lista de ilustrações

# Sumário

# 1 Introdução

Grande parte das aplicações apresentam funcionalidades que demandam muito tempo de execução, um exemplo desse tipo de funcionalidade é a geração do backup de dados de um aplicativo. Para o usuário não é interessante perder tempo acompanhando no smartphone a geração do backup. Pensando nisso, há estruturas no android que possibilitam que a execução de uma tarefa seja feita sem que o usuário perceba que está sendo executada em background.

Dentro das estruturas existentes, encontramos aquelas que são executadas somente quando o aplicativo está em funcionamento e não abrange muitos cenários de utilização da aplicação. Na prática, quando uma dessas estruturas é utilizada mesmo que seja agendado um horário da de execução da tarefa, por exemplo backup, não será possível garantir a realização da tarefa.

Nesse contexto existe estruturas que possibilitam a execução de atividades referente a uma aplicação mesmo essa estando *dead*(*morta*). Essas estruturas têm o objetivo de executar funcionalidades quando o aplicativo não está (ou está) em execução para a manutenção do mesmo em certos horários pré estabelecidos.

O documento tem como objetivo explicar essas estruturas, uma vez que se viu necessário a execução da descarga um horário pré estabelecido. Essa necessidade deriva de cenários onde o vendedor não conseguiu efetuar descarga durante o seu horário de trabalho. Assim, o obketivo é efetuar a descarga automática quando a mesma não foi efetuada no dia.

# 2 Desenvolvimento

## 2.1 Service on Xamarin Android

Os aplicativos móveis não são como aplicativos da área de trabalho. As áreas de trabalho têm mais recursos como espaço real da tela, memória, espaço de armazenamento e uma fonte de alimentação conectada, os dispositivos móveis não. Essas restrições forçam os aplicativos móveis a se comportarem de forma diferente. Por exemplo, a tela pequena em um dispositivo móvel normalmente significa que apenas um aplicativo (ou seja, atividade) está visível por vez. Outras atividades são movidas para o plano de fundo e enviadas para um estado suspenso onde não podem executar nenhum trabalho. No entanto, só porque um aplicativo Android está em segundo plano não significa que é impossível para o aplicativo continuar funcionando.

Os aplicativos Android são compostos de pelo menos um dos quatro principais componentes a seguir: atividades, receptores de transmissão, provedores de conteúdo e serviços. As atividades são a base de muitos aplicativos do Android, pois fornecem a interface que permite interação do usuário com a aplicação. No entanto, quando se trata de executar trabalho simultâneo ou em segundo plano, as atividades nem sempre são a melhor opção (MICROSOFT, b).

O mecanismo principal de trabalho em segundo plano no Android é o serviço. Um serviço Android é um componente projetado para fazer algum trabalho sem a necessidade de um interface do usuário. Um serviço pode baixar um arquivo, reproduzir música ou aplicar um filtro a uma imagem. Os serviços também podem ser usados para IPC (comunicação entre processos) entre aplicativos Android. Por exemplo, um aplicativo Android pode usar o serviço de player de música que é de outro aplicativo ou um aplicativo pode apresentar dados (como as informações de contato de uma pessoa) para outros aplicativos por meio de um serviço.

Para resolver essa preocupação, um desenvolvedor pode usar threads em uma atividade para executar algum trabalho que bloqueie a interface do usuário. No entanto, isso pode causar problemas. É muito possível que o Android destrua e recrie as várias instâncias da atividade. No entanto, o Android não destruirá automaticamente os threads, o que pode resultar em vazamentos de memória. Um exemplo primo disso é quando o dispositivo é girado – o Android tentará destruir a instância da atividade e, em seguida, recriar uma nova.

Esse é um possível vazamento de memória – o thread criado pela primeira instância da atividade ainda estará em execução. Se o thread tiver uma referência à primeira instância da atividade, isso impedirá que o Android finalize o objeto de coleta de lixo. No entanto, a segunda instância da atividade ainda é criada (o que, por sua vez, pode criar um novo thread). A rotação do dispositivo várias vezes em uma rápida sucessão pode esgotar toda a RAM e forçar o Android a encerrar todo o aplicativo para recuperar memória (MICROSOFT, b).

Como regra geral, se o trabalho a ser executado deve sobreviver além uma atividade, um serviço deve ser criado para executar esse trabalho. No entanto, se o trabalho só for aplicável no contexto de uma atividade, a criação de um thread para executar o trabalho poderá ser mais apropriada. Por exemplo, criar uma miniatura para uma foto que acabou de ser adicionada a um aplicativo da galeria de fotos provavelmente ocorreria em um serviço. No entanto, um thread pode ser mais apropriado para reproduzir algumas músicas que só devem ser ouvidas quando uma atividade estiver em primeiro plano.

Os serviços e sua capacidade de executar trabalho em segundo plano são cruciais para fornecer uma interface de usuário suave e fluida. Todos os aplicativos Android têm um thread principal (também conhecido como um thread de interface do usuário) no qual as atividades são executadas. Para manter o dispositivo responsivo, o Android deve ser capaz de atualizar a interface do usuário na taxa de 60 quadros por segundo. Se um aplicativo Android executar muito trabalho no thread principal, o Android descartará os quadros, o que, por sua vez, faz com que a interface do usuário pareça Jerky (às vezes chamado de Janky). Isso significa que qualquer trabalho realizado no thread da interface do usuário deve ser concluído no período de tempo entre dois quadros, aproximadamente 16 milissegundos (1 segundo a cada 60 quadros) (MICROSOFT, b).

O trabalho em segundo plano pode ser dividido em duas classificações amplas:

1. *Tarefa de longa execução* – isso funciona em andamento até que seja interrompido explicitamente. Um exemplo de uma tarefa de execução demorada é um aplicativo que transmite música ou que deve monitorar dados coletados de um sensor. Essas tarefas devem ser executadas, mesmo que o aplicativo não tenha nenhuma interface do usuário visível.

2. *Tarefas periódicas* – (às vezes chamados de trabalho) uma tarefa periódica é aquela que tem duração relativamente curta (vários segundos) e é executada em uma agenda (ou seja, uma vez por dia por uma semana ou talvez apenas uma vez nos próximos 60 segundos). Um exemplo disso é baixar um arquivo da Internet ou gerar uma miniatura para uma imagem.

Há quatro tipos diferentes de serviços do Android:

1. *O serviço associado* – um serviço associado é um serviço que tem algum outro componente (normalmente uma atividade) associado a ele. Um serviço associado fornece uma interface que permite que o componente ligado e o serviço interajam entre si. Quando não houver mais clientes ligados ao serviço, o Android encerrará o serviço.

2. *IntentService* – um IntentService é uma subclasse especializada da classe *Service* que simplifica a criação e o uso de serviços. Um IntentService é destinado a tratar chamadas autônomas individuais. Ao contrário de um serviço, que pode lidar simultaneamente com

várias chamadas, um IntentService é mais parecido com um processador de fila de trabalho – o trabalho é colocado em fila e um IntentService processa cada trabalho de cada vez em um único thread de trabalho. Normalmente, um IntentService não está associado a uma atividade ou a um fragmento.

3. *O serviço iniciado* – um serviço iniciado é um serviço que foi iniciado por algum outro componente do Android (como uma atividade) e é executado continuamente em segundo plano até que algo indique explicitamente o serviço deve ser finalizado. Ao contrário de um serviço associado, um serviço iniciado não tem nenhum cliente associado diretamente a ele. Por esse motivo, é importante criar serviços iniciados para que eles possam ser reiniciados normalmente conforme necessário.

4. *O serviço híbrido* – um serviço híbrido é um serviço que tem as características de um serviço iniciado e um serviço associado. Um serviço híbrido pode ser iniciado pelo aplicativo quando um componente é associado a ele ou pode ser iniciado por algum evento. Um componente de cliente pode ou não estar associado ao serviço híbrido. Um serviço híbrido continuará em execução até que seja explicitamente solicitado a parar ou até que não haja mais clientes associados a ele.

O tipo de serviço a ser usado depende muito dos requisitos do aplicativo. Como regra prática, um IntentService ou um serviço associado é suficiente para a maioria das tarefas que um aplicativo Android deve executar, de modo que a preferência deve ser dada a um desses dois tipos de serviços. Uma IntentService é uma boa opção para tarefas "One-shot", como baixar um arquivo, enquanto um serviço associado seria adequado quando interações frequentes com uma atividade/fragmento forem necessárias.

Embora a maioria dos serviços seja executada em segundo plano, há uma subcategoria especial conhecida como serviço de primeiro plano. Esse é um serviço que recebe uma prioridade mais alta (em comparação com um serviço normal) para executar algum trabalho para o usuário (como tocar música).

Também é possível executar um serviço no próprio processo no mesmo dispositivo, às vezes, isso é chamado de serviço remoto ou como um serviço fora do processo. Isso exige mais esforço para criar, mas pode ser útil para quando um aplicativo precisa compartilhar funcionalidades com outros aplicativos e pode, em alguns casos, melhorar a experiência do usuário de um aplicativo.

As classes de serviço no Android precisam respeitar três princípios para o correto funcionamento (MICROSOFT, b):

1. Devem estender a classe *Android.App.Service* (MICROSOFT, c);

2. Devem ser decorados com o *Android.App.ServiceAttribute* (MICROSOFT, d);

3. Eles devem ser registrados também no *AndroidManifest.xml* e receber um nome exclusivo.

A figura 1 apresenta um exemplo de um código que contempla as três regras.

```
C#

[Service]
public class DemoService : Service
{
    // Magical code that makes the service do wonderful things.
}
XML

<service android:name="md5a0cbbf8da641ae5a4c781aaf35e00a86.DemoService" />
```

Figura 1 – Classe de criação do serviço e xml da aplicação declarando o serviço criado

No momento de execução da aplicação o Android registra o serviço através do elemento presente no AndroidManifest.xml.

É possível ainda que se crie um serviço para a execução através de outros aplicativos, ou seja, criar um serviço compartilhado. Esses serviços normalmente são utilizados para abrir certos tipos de arquivos. Um exemplo é um aplicativo para abrir arquivos pdf, o aplicativo provavelmente terá um serviço compartilhado que fornece estrutura para abrir um arquivo desse formato. Para a criação de um serviço compartilhado basta adicionar o trecho da figura 2, definindo a propriedade Exported no ServiceAttribute.

```
XML

<service android:exported="true" android:name="com.xamarin.example.DemoService" />
```

Figura 2 – Setando serviço como compartilhado

Os serviços têm seu próprio ciclo de vida com métodos de retorno de chamada que são invocados conforme o serviço é criado. Exatamente quais métodos são invocados dependem do tipo de serviço. Um serviço iniciado deve implementar diferentes métodos de ciclo de vida que um serviço associado, enquanto um serviço híbrido deve implementar os métodos de retorno de chamada para um serviço iniciado e um serviço associado. Esses métodos são todos membros da classe *Service*; a forma como o serviço é iniciado determinará quais métodos de ciclo de vida serão invocados. Esses métodos de ciclo de vida serão discutidos em mais detalhes posteriormente (MICROSOFT, b).

Por padrão, um serviço será iniciado no mesmo processo que um aplicativo Android. É possível iniciar um serviço em seu próprio processo definindo a propriedade *ServiceAttribute.IsolatedProcess* como true, como é apresentado na figura 3:

```
C#

[Service(IsolatedProcess=true)]
public class DemoService : Service
{
    // Magical code that makes the service do wonderful things, in it's own process!
}
```

Figura 3 – Setando serviço para executar em uma tarefa separada

### 2.1.1 Starting a Service

A maneira mais básica de iniciar um serviço no Android é distribuir um *Intent* que contém metadados para ajudar a identificar qual serviço deve ser iniciado. Há dois estilos diferentes de intenções que podem ser usados para iniciar um serviço:

1. A *intenção explícita* – uma intenção explícita identificará exatamente qual serviço deve ser usado para concluir uma determinada ação. Uma intenção explícita pode ser considerada como uma letra que tem um endereço específico. O Android roteará a intenção para o serviço identificado explicitamente. Este trecho de código é um exemplo de uso de uma intenção explícita para iniciar um serviço chamado DownloadService:

```
C#

// Example of creating an explicit Intent in an Android Activity
Intent downloadIntent = new Intent(this, typeof(DownloadService));
downloadIntent.data = Uri.Parse(fileToDownload);
```

Figura 4 – Executando serviço de forma explícita

2. A *intenção implícita* – esse tipo de tentativa identifica livremente a ação que o usuário deseja executar, mas o serviço exato para concluir essa ação é desconhecido. Uma intenção implícita pode ser considerada como uma letra que é endereçada "a quem ele pode se preocupar...". O Android examinará o conteúdo da intenção e determinará se há um serviço existente que corresponda à intenção.

   Um filtro de intenção é usado para ajudar a corresponder a intenção implícita com um serviço registrado. Um filtro de intenção é um elemento XML que é adicionado a *An-*

*droidManifest.xml* que contém os metadados necessários para ajudar a corresponder um serviço com uma intenção implícita.

```
C#

Intent sendIntent = new Intent("common.xamarin.DemoService");
sendIntent.Data = Uri.Parse(fileToDownload);
```

Figura 5 – Executando serviço de forma explícita

***Uma observação importante é que a partir do Android 5,0 (AP nível 21), uma intenção implícita não pode ser usada para iniciar um serviço.***

Sempre que possível, os aplicativos devem usar tentativas explícitas para iniciar um serviço. Uma intenção implícita não solicita que um serviço específico seja iniciado – é uma solicitação de algum serviço instalado no dispositivo para lidar com a solicitação. Essa solicitação ambígua pode fazer com que o serviço incorreto esteja manipulando a solicitação ou outro aplicativo a partir do início desnecessariamente (o que aumenta a pressão de recursos no dispositivo).

A forma como a intenção é distribuída depende do tipo de serviço e será discutida em mais detalhes posteriormente nos guias específicos de cada tipo de serviço.

No link é possível acessar um projeto de exemplo que utiliza serviço.

## 2.2 Android Job Scheduler

Uma das melhores maneiras de manter um aplicativo Android responsivo ao usuário é garantir que o trabalho complexo ou de longa execução seja executado em segundo plano. No entanto, é importante que o trabalho em segundo plano não afete negativamente a experiência do usuário com o dispositivo.

Por exemplo, um trabalho em segundo plano pode sondar um site a cada três ou quatro minutos para consultar alterações em um determinado conjunto de dados. Isso parece benigno, mas teria um impacto desastroso na vida útil da bateria. O aplicativo ativará repetidamente o dispositivo, elevará a CPU a um estado de energia mais alto, ligará as rádios, realizará solicitações de rede e, em seguida, processando os resultados. Isso é pior porque o dispositivo não será imediatamente desligado e retornará ao estado ocioso de baixa energia. Um trabalho em segundo plano agendado incorretamente pode manter o dispositivo inadvertidamente em um estado com requisitos de energia excessivas e desnecessários. Essa atividade aparentemente inocente (sondando um site) tornará o dispositivo inutilizável em um período de tempo relativamente curto (MICROSOFT, a).

O Android fornece as seguintes APIs para ajudar na execução do trabalho em segundo plano, mas por si só eles não são suficientes para o agendamento inteligente de trabalhos.

1. *Serviços de intenção* – serviço de intenção são ótimos para executar o trabalho, no entanto, eles não fornecem agendamento do trabalho.
2. O alarmmanager – essas APIs só permitem que o trabalho seja agendado, mas não fornecem como realmente realizar o trabalho. Além disso, o Alarmmanager permite apenas restrições baseadas em tempo, o que significa gerar um alarme em um determinado momento ou depois de um determinado período de tempo decorrido.
3. *Os receptores de difusão* – um aplicativo Android pode configurar receptores de difusão para executar o trabalho em resposta a eventos ou tentativas de todo o sistema. No entanto, os receptores de difusão não fornecem nenhum controle sobre quando o trabalho deve ser executado. Além disso, as alterações no sistema operacional Android serão restritas quando os receptores de transmissão funcionarem ou os tipos de trabalho aos quais eles podem responder.

Há dois recursos principais para executar com eficiência o trabalho em segundo plano (às vezes chamado de trabalho em segundo plano ou um trabalho) (MICROSOFT, a):

1. Agendando o trabalho de forma inteligente – é importante que, quando um aplicativo estiver trabalhando em segundo plano, ele faça isso como um bom cidadão. O ideal é que o aplicativo não exija que um trabalho seja executado. Em vez disso, o aplicativo deve especificar condições que devem ser atendidas para quando o trabalho puder ser executado e, em seguida, agendar esse trabalho com o sistema operacional que executará o trabalho quando as condições forem atendidas. Isso permite que o Android execute o trabalho para garantir a máxima eficiência no dispositivo. Por exemplo, as solicitações de rede podem ser executadas em lotes para executar tudo ao mesmo tempo para fazer o uso máximo da sobrecarga envolvida na rede.
2. Encapsular o trabalho – o código para executar o trabalho em segundo plano deve ser encapsulado em um componente discreto que pode ser executado independentemente da interface do usuário e será relativamente fácil de reagendar se o trabalho não for concluído por alguma falha.

### 2.2.1 AlarmManager Class

Esta classe fornece acesso aos serviços de alarme do sistema. Isso permite que você agende seu aplicativo para ser executado em algum momento no futuro. Quando um alarme dispara, o Intent que foi registrado para ele é transmitido pelo sistema, iniciando automaticamente

o aplicativo de destino se ele ainda não estiver em execução. Os alarmes registrados são retidos enquanto o dispositivo está no modo de suspensão (e, opcionalmente, podem ser ativados se o dispositivo disparar durante esse período), mas serão apagados se ele for desligado e reiniciado (Developers Google, ).

O Gerenciador de alarmes mantém um bloqueio de ativação da CPU enquanto o método *onReceive()* do receptor de alarme estiver em execução. Isso garante que o telefone não durma até que você termine de lidar com a transmissão. Quando *onReceive()* retorna, o Gerenciador de alarmes libera esse bloqueio de ativação. Isso significa que, em alguns casos, o telefone irá dormir assim que o método *onReceive()* for concluído. Se o seu receptor de alarme chamar *Context.startService()*, é possível que o telefone entre em suspensão antes do lançamento do serviço solicitado. Para evitar isso, seu *BroadcastReceiver* e *Service* precisará implementar uma política de bloqueio de ativação separada para garantir que o telefone continue funcionando até que o serviço fique disponível.

**Nota:** *O Gerenciador de alarmes destina-se aos casos em que você deseja que o código do seu aplicativo seja executado em um horário específico, mesmo se o aplicativo não estiver em execução no momento. Para operações normais de cronometragem (ticks, timeouts, etc), é mais fácil e muito mais eficiente usar o Handler.*

Além da classe *AlarmManager* associa-se o serviço de alarme com o componente *BroadCast Receiver* do android, assim o serviço irá invocar este receiver na hora agendada.

Um *broadcast receiver* ou *receptor de difusão* é um componente do Android que permite que um aplicativo responda a mensagens (uma Intenção do Android) que são transmitidas pelo sistema operacional Android ou por um aplicativo. As transmissões seguem um modelo de publicação-inscrição: um evento faz com que uma transmissão seja publicada e recebida pelos componentes que estão interessados no evento (MACORATTI, ).

Usa-se Intents que é um conceito abstrato para algum tipo de operação que deverá ser executada no sistema operacional Android. Intents ou Intenções no Android, são estruturas de dados que são objetos de mensagens. Intenções podem solicitar uma operação a ser realizada por algum outro componente no Android e são geralmente usadas para iniciar Atividades e Serviços.

Para criar um alarme é necessário a criação de uma classe que erda de *BroadCastReceiver* e implementa o método *OnReceive()*. A figura 6 apresenta um exemplo de criação da classe que, quando executado, apresenta uma mensagem a partir de Toast:

```
using Android.Content;
using Android.Widget;
namespace Droid_Alarme.BroadCast
{
    [BroadcastReceiver(Enabled = true)]
    public class AlarmeMensagemReceiver : BroadcastReceiver
    {
        public override void OnReceive(Context context, Intent intent)
        {
            Toast.MakeText(context, "Alarme do Macoratti", ToastLength.Long).Show();
        }
    }
}
```

Figura 6 – Criando classe de alarme para executar uma mensagem em forma de Toast

Para iniciar o alarme deve-se primeiramente instanciar uma classe de controle do *AlarmManager*. A partir dessa instância é que se programa a atividade a ser executada. Deve-se instanciar a classe criada (*AlarmMensagemReceiver*) e passar essa classe como parâmetro para a classe de controle do *AlarmManager*. Essa classe de controle possibilita dois tipos de programação de tarefa: uma recorrente e uma que executa apenas uma vez. A figura 7 apresenta essa parte de inicialização de um alarme de forma recorrente e não recorrente.

```
private void IniciarAlarme(bool isNotificacao, bool isRepetirAlarme)
{
    AlarmManager manager =
        (AlarmManager)GetSystemService(Android.Content.Context.AlarmService);

    Intent minhaIntent;
    PendingIntent pendingIntent;

    if(!isNotificacao)
    {
        minhaIntent = new Intent(this, typeof(AlarmeMensagemReceiver));
        pendingIntent = PendingIntent.GetBroadcast(this, 0, minhaIntent, 0);
    }
    else
    {
        minhaIntent = new Intent(this, typeof(AlarmeNotificacaoReceiver));
        pendingIntent = PendingIntent.GetBroadcast(this, 0, minhaIntent, 0);
    }

    if(!isRepetirAlarme)
    {
        manager.Set(AlarmType.RtcWakeup,
                    SystemClock.ElapsedRealtime() + 3000,
                    pendingIntent);
    }
    else
    {
        manager.SetRepeating(AlarmType.RtcWakeup,
                             SystemClock.ElapsedRealtime() + 3000, 60 * 1000,
                             pendingIntent);
    }
}
```

Figura 7 – Iniciando o alarme programado.

Dessa forma, quando o método *IniciarAlarme* é chamado ele programa o alarme para execução da tarefa colocada no método *AlarmeMensagemReceiver.OnReceive()*.

Afim de apresentar um teste de prova, o link apresenta o exemplo citado para criação do alarme.

No próximo capítulo será apresentado como foi utilizado essas tecnologias para criação do projeto de prova da funcionalidade de interesse.

# 3 Teste de Avaliação

Este capítulo tem como objetivo explicar a necessidade do cliente e como que as ferramentas já apresentadas contribuiram para alcançar o sucesso no projeto. Ele cita e apresenta como foi implementado cada seção para o funcionamento do código como um todo.

## 3.1 Contextualização do Problema

Foi listado que o cliente necessitava de uma funcionalidade que forçasse a descarga automática dado um horário sem que o usuário precisasse acessar essa opção. Essa descarga deveria ser feita independentemente do estado de como o aplicativo esteja, ou seja, se o aplicativo estiver em *backgound*, *foreground* ou *dead*.

Essa necessidade advém da dificuldade de controle da descarga de todos os vendedores no seu momento de finalização da rota diária, sendo essa definida por um horário e, se não finalizada, pode haver problemas dentro das políticas de legislação trabalhista. Basicamente o funcionário tem um horário:

1. Iniciar a jornada: Início do seu horário de trabalho e liberado para vender;
2. Horário de almoço: Horário em que o sistema não permite que o vendedor trabalhe, finalizado apenas quando o horário de almoço acaba;
3. Finalização da jornada: Horário em que o sistema trava e não possibilita que o vendedor execute nenhuma atividade no aplicativo;

Tendo esses horários definidos, se o vendedor não conseguir transmitir a descarga final para a empresa matriz os pedidos chegam apenas no outro dia. Com isso afeta todo o trabalho da empresa para concluir os pedidos, elaborar novas rotas, estabelecer seus KPIs e organizar rotas de entrega dos caminhões.

Afim de solucionar esse problema a tentativa é fazer com que o sistema atue no momento em que a jornada finaliza e, se não efetuada a transmissão pelo próprio vendedor, o sistema executa-la automaticamente, assim os pedidos chegarão na base do cliente.

## 3.2 Teste de Viabilidade

Para efetuar um teste de viabilidade da solução foi utilizado o sistema de forças de venda SFV. Ele comporta toda a parte de descarga das informações de venda do dia, tendo ainda opção

de limitar horários de funcionamento do sistema. Esse horário de funcionamento limita horários em que se pode descarregar.

Para o teste não foi utilizado a regra de intervalo de horário, pois a mesma é utilizada apenas na prática e não para os testes de viabilidade.

Foi necessário para aplicação do teste a alteração de uma das regras do sistema, a regra referente à inicialização automatica da aplicação. Essa regra faz com que, se o sistema estiver sido fechado pelo própio android, o aplicativo identifica em qual activity ele estava e reconstrói a mesma com os dados necessários, assim como efetua o login automático. Foi necessário alterá-la porque essa classe controla toda vez que a aplicação é aberta, e uma vez identificado que o próprio Android está abrindo-a (como o alarm e o serviço fazem) ele tenta inicializa-la automaticamente, abrindo as telas necessárias para isso, ao invés da tela de descarga, como é a necessidade do cliente.

Essa regra funciona a partir da classe *MDN_Application* que herda da classe *Application* e reescreve o método *OnCreate()* passando a chamar o método *CheckApplicationStartMode* que define o *login* automático e onde foi feita a alteração apresentada na figura 8.

```
else if ((!taskInfo.TopActivity.ClassName.EndsWith("ACN_Inicializacao")) &&
        (!taskInfo.TopActivity.ClassName.EndsWith("ACN_Manutencao")))
    {
        // Android restart mode
        Log.Info("LDX", "MDN_Application.CheckApplicationStartMode: Android restart mode: " +
                taskInfo.TopActivity.ClassName);
        AppSettings.IdfAutoLogin = false;
        AppSettings.IdfRecuperada = true;
        if (Locale.Default.Language.Contains("pt"))
            Toast.MakeText(Context, "Aguarde, reiniciando o sistema...", ToastLength.Long).Show();
        else if (Locale.Default.Language.Contains("es"))
            Toast.MakeText(Context, "Espere, reiniciando el sistema...", ToastLength.Long).Show();
        else
            Toast.MakeText(Context, "Please wait, restarting the system...", ToastLength.Long).Show();

        Intent launchIntent = Context.PackageManager.GetLaunchIntentForPackage(Context.PackageName);
        Intent mainIntent = Intent.MakeRestartActivityTask(launchIntent.Component);
        // StartActivity(mainIntent);    coment to work
        // Java.Lang.JavaSystem.Exit(0); coment to work
        return;
    }
```

Figura 8 – Alteração na classe *MDN_Application*

Para o funcionamento correto do sistema, foi alterado também as classes:

1. *ACN_Inicializacao*: Alterada para identificar quando o sistema está sendo chamado a partir de alguma aplicação externa (ou mesmo pelo própio Android). Alterada ainda para receber uma entrada do tipo *bool* que identifica se é para efetuar descarga automática. Devido à funcionalidade da aplicação, essa é a classe central do sistema onde processa o *login* e identifica se deve abrir ou não a tela *Home*. Foi alterado também para enviar para ambas as telas se o sistema precisa efetuar a descarga ou não. A última alteração foi feita para que, quando retornado da tela *Home*, o aplicativo valide se é retorno da descarga automática, se satisfeita a condição a aplicação se encerra automaticamente.

2. *ACN_Logon*: Alterado para receber uma entrada *bool* que identifica se a requisição é de uma descarga automática, se sim, é feito o login automático. Foi necessário alterar também um trecho do código utilizado pela tarefa de login automático, já citada anteriormente. Esse trecho estava preenchendo o campo de senha com os caracteres: "*******", assim o sistema identificava que a senha estava preenchida e não apresentava mensagem de erro. O problema é que a descarga utiliza, para validação com o LMMS, o mesmo usuário e senha que foram digitados no login. Então foi alterado para não preencher a senha automaticamente, mas sim validar que quando é descarga ou login automático o aplicativo não precisa validar se está preenchido o campo de senha. O aplicativo já estava preparado para armazenar a última senha e usuário logados, criptografados, então não existe o problema na identificação de qual usuário validar na descarga. Foi alterado também para não atualizar os dados de conexão quando for descarga automática.

3. *AC_Home*:Alterado para receber uma entrada *bool* que identifica se a requisição é de uma descarga automática, se sim abre a opção de descarga. Alterado ainda para que no retorno da descarga, se identificado que foi a partir de uma descarga automática, feche a *activity* enviando um sinal para a classe que a chamou (*ACN_Inicializacao*) que foi feita a partir de uma requisição de descarga automática. Essa tela foi alterada também para iniciar o serviço ou o alarme, dependendo do que foi utilizado. Essa classe é responsável por essa parte pois só pode agendar a atividade quando a aplicação já foi logada pelo usuário, ou seja, se o usuário não logar no dia, a descarga nunca será executada. Ademais foi alterado para tratar as mensagens de retorno da descarga, para que quando fosse identificado um retorno de descarga automática não seja apresentado nenhuma mensagem para o usuário.

4. *AC_Descarregar*: Alterado receber uma entrada *bool* que identifica se a requisição é de uma descarga automática, se sim efetua a descarga sem apresentar nenhuma mensagem. Alterado também para retornar para a classe que a instanciar uma identificação que o retorno é de uma descarga automática.

Com o conjunto de alterações feitas o sistema está preparado para efetuar descarga para quando instanciar a activity *ACN_Inicializacao* passando a chave *"has_service_been_started"* com o valor true. Essa chave pode ser qualquer nome escolhido, foi utilizado essa para o projeto por escolha aleatória apenas. Então para o funcionamento do sistema ocorrer da maneira desejável basta o serviço ou o alarme instânciar a classe passando esse parâmetro.

## 3.3 Serviço do Android abrindo a aplicação

Essa seção apresenta como fazer a descarga funcionar automaticamente utilizando serviço do Android. Como já foi explicado na seção *Starting a Service*, para a incialização do serviço deve-se primeiramente criá-lo. Para esse projeto foi criado então a classe *Timestamp-*

*Service* que extende e herda da classe *Service* do Android e implementa os métodos *OnCreate*, *OnStartCommand*, *OnStart* e *OnDestroy*. Assim como no exemplo dado na outra seção, quando essa classe é executada ela tenta executar alguma função específica. A diferença é que o serviço tenta abrir a aplicação e efetua a descarga.

A imagem 9 apresenta como a classe foi criada.

```
[Service(Enabled =true, Exported = false)]
2 references | 0 changes | 0 authors, 0 changes
public class TimestampService : Service
{
    Atributes
    1 reference | 0 changes | 0 authors, 0 changes
    public override void OnCreate()...

    0 references | 0 changes | 0 authors, 0 changes
    public override StartCommandResult OnStartCommand(Intent intent,
                                                       StartCommandFlags flags,
                                                       int startId)...

    0 references | 0 changes | 0 authors, 0 changes
    public override void OnStart(Intent intent, int startId)...

    0 references | 0 changes | 0 authors, 0 changes
    public override IBinder OnBind(Intent intent)...

    1 reference | 0 changes | 0 authors, 0 changes
    public override void OnDestroy()...

    Privates Methods

}
```

Figura 9 – Criação da classe de Serviço que executará as funções de chamada de descarga.

Para a chamada da aplicação foi criado o método *OpenAtivity()* que tem o objetivo de verificar o horário e, se chegada a hora de executar a descarga, chama o método *TrySend()* que abre a aplicação passando o parâmetro de descarga. A figura 10 apresenta os dois métodos. Vale ressaltar que o projeto foi apenas para provar a viabilidade de se utilizar o serviço para a tarefa, então a condição de início da tarefa foi aplicada fixa com um valor único e significativo para teste.

```
private void OpenAtivity()
{
    if (this.timestamper != null)
    {
        Log.Debug(TAG, this.timestamper.GetFormattedTimestamp());

        if (DateTime.Now.Minute == 5 && !this.opened)
        {
            this.TrySend();
            this.opened = true;
        }
        this.handler.PostDelayed(this.runnable, DELAY_BETWEEN_LOG_MESSAGES);
    }
}
private void TrySend()
{
    try
    {
        this.StartForeground(0, new Notification());

        Intent launchIntent = Application.Context.PackageManager.
            GetLaunchIntentForPackage(Application.Context.PackageName);
        Intent mainIntent =
            Intent.MakeRestartActivityTask(launchIntent.Component);

        mainIntent.SetPackage(Application.Context.PackageName);
        mainIntent.PutExtra(SERVICE_STARTED_KEY, true);
        this.StartActivity(mainIntent);
    }
    catch (Exception e)
    {
        MDN_GlobalVars.EscreveLogGeral("ErrorPi: " + e.Message, Ldx_Log.EN_NiveisLog.SEQUENCIA_METODOS);
    }
}
```

Figura 10 – Métodos de execução para abertura da descarga

A partir desses métodos é possível abrir a aplicação nos seguintes casos:

1. Aplicativo em foreground e telefone desbloqueado;
2. Aplicativo em foreground e telefone bloqueado;
3. Aplicativo em backgound e telefone desbloqueado;
4. Aplicativo em backgound e telefone bloqueado;

O serviço não funcionou quando a aplicação está *morta*. Foi estudado o porquê não foi possível abrir a aplicação pelo serviço e chegou a conclusão de que não tem a permissão para fazer isso. O objetivo de um serviço é executar tarefas em background, sem interação com o usuário, então não foi encontrado casos em que foi possível abrir a aplicação. Para efetuar o teste não foi possível fazer no modo debug, pois a aplicação precisava estar *morta*. Nesses testes não foi apresentado a tela do aplicativo quando o mesmo estava fechado, mesmo depois de ativado o serviço.

Para a execução do serviço foi implementado na classe *AC_Home* o trecho da figura 11 no método *OnCreate()*.

```
protected override void OnCreate(Bundle bundle)
{
    MDN_GlobalVars.EscreveLogGeral("AC_Home.OnCreate(Bundle bundle)", Ldx_Log.EN_NiveisLog.SEQUENCIA_METODOS);

    base.OnCreate(bundle);

    // Start services
    if (bundle != null)
    {
        this.isStarted = bundle.GetBoolean(SERVICE_STARTED_KEY, false);
    }

    this.serviceToStart = new Intent(this, typeof(Services.TimestampService));

    if (this.Intent.GetBooleanExtra(SERVICE_STARTED_KEY, false))
    {
        this.automatico = true;
        this.btn_descarregar_Click(null, null);
    }
    else
    {
        //this.StartAlarm();

        if (!this.isStarted)
        {
            this.StartService(this.serviceToStart);
        }

        this.IniciaActivity();
    }
}
```

Figura 11 – Trecho de instanciamento do serviço para executar a descarga do sistema.

Esse trecho de código tem como objetivo verificar se a chamada da tela é a partir de uma descarga automática, e se for utiliza a instância do serviço criado e coloca na pilha de execução do Android através do comando *StartService(Service)*.

## 3.4 Alarme do Android abrindo a aplicação

Como já foi explicado, o alarme do Android possibilita que uma tarefa seja agendada para executar em certo horário no dia. Para efetuar o teste essa funcionalidade não foi respeitada, uma vez que a criação do alarme funcionou de forma diferente.

Para criação da classe do alarme foi criada a classe *OpenApp*. Essa classe herda e implementa a *BroadcastReceiver* e tem a propriedade *Enabled* igual a *true*. A figura 12 apresenta a criação da classe. A figura apresenta também o método reescrito *OnReceive* que faz todo o papel da descarga. Vale ressaltar que o projeto foi apenas para provar a viabilidade de se utilizar o alarme para a tarefa, então a condição de início da tarefa foi aplicada fixa com um valor único e significativo para teste.

```
[BroadcastReceiver(Enabled = true)]
4 references | 0 changes | 0 authors, 0 changes
public class OpenApp : BroadcastReceiver
{

    Atributes

    Makers

    2 references | 0 changes | 0 authors, 0 changes
    public override void OnReceive(Context context, Intent intent)
    {
        if (this.executei)
            return;

        else if (DateTime.Now.Minute != 55)
            return;

        this.executei = true;
        Toast.MakeText(context, "Descarregando", ToastLength.Short).Show();
        Intent mainIntent = Intent.MakeRestartActivityTask(launchIntent.Component);

        mainIntent.PutExtra(SERVICE_STARTED_KEY, true);
        context.StartActivity(mainIntent);
    }

}
```

Figura 12 – Métodos de execução para abertura da descarga

O método tem um objetivo claro, se já conseguiu executar a descarga, ou se não for a hora de descarregar, não faz nada, e se for a hora e não executou, ele abre a aplicação passando a opção de descarga automática.

A forma de instanciar e chamar o alarm, o que foi diferente devido ao teste, está apresentado na figura 13.

```
private void StartAlarm()
{
    AlarmManager manager = (AlarmManager)GetSystemService(AlarmService);
    Intent minhaIntent;
    PendingIntent pendingIntent;
    minhaIntent = new Intent(this, new BroadCast.OpenApp(this).Class);
    pendingIntent = PendingIntent.GetBroadcast(this, 0, minhaIntent, 0);
    int hour = 12;
    int minute = 25;
    DateTime time = new DateTime(2020, 11, 19, hour, minute, 0, DateTimeKind.Utc);

    manager.SetRepeating(AlarmType.RtcWakeup,
                    (long)time.ToUniversalTime().Subtract(time).TotalMilliseconds,
                    60 * 1000,
                    pendingIntent);
}
```

Figura 13 – Instanciando um alarme para executar de forma recorrente até uma data fixa

O método tem como objetivo programar o objeto instanciado do alarm do android de forma que seja recorrente e a cada 60*1000 milissegundos execute a função declarada em *OpenApp.OnReceive()*. Ele ainda seta como data final de execução do alarme. Dessa forma o alarm fica executando a cada x milissegundos o método, que valida o horário e, se necessário, efetua a descarga. O resultado é listado a seguir:

1. Aplicativo em foreground e telefone desbloqueado;
2. Aplicativo em foreground e telefone bloqueado;
3. Aplicativo em backgound e telefone desbloqueado;
4. Aplicativo em backgound e telefone bloqueado;
5. Aplicativo inoperante e telefone desbloqueado;
6. Aplicativo inoperante e telefone bloqueado;

Assim foi possível executar todos os cenários de interesse em que se vê necessário a descarga do sistema. Vale ressaltar que a prova de teste apenas provou que existe maneiras de chamar a aplicação para a descarga automática e que a aplicação pode variar dependendo da necessidade, mas que abre porta para outras opções de tarefas automáticas a serem agendadas ou mesmo serem executadas em background, como a própria descarga e a carga das nossas aplicações.

# 4 Conclusão

O objetivo do trabalho era efetuar o teste de como executar tarefas em background de maneira automática no Android. Foi dado o cenário em que é necessário que a aplicação execute uma descarga em um tempo previamente programado, o que foi respeitado e simulado o mais próximo que se pode. Foi análisado duas formas para a execução da tarefa, utilizando o *Service* do Android e o *AlarmManager*, em resultados é possível ver quais foram as saídas dos dois e quais são as consequências de utilizar cada um.

Com a pesquisa conclui-se que pelo objetivo a ser alcançado o ideal é utilizar o *Alarm*, pois ele respeita todos os cenários previstos e possibilita o agendamento do horário como foi requerido. Através das imagens colocadas no documento é possível utilizar ambas as tecnologias e, inclusive, implementar o que foi implementado para efetuar a descarga.

# Referências

Developers Google. *AlarmManager | Android Developers*. Disponível em: <https://developer.android.com/reference/android/app/AlarmManager.html>. Citado na página 13.

MACORATTI, J. C. *Xamarin Android - Usando o serviço de Alarme*. Disponível em: <http://www.macoratti.net/17/04/xamand{_}alarm1.> Citado na página 13.

MICROSOFT, C. *Agendador de trabalhos do Android - Xamarin _ Microsoft Docs*. Disponível em: <https://docs.microsoft.com/pt-br/xamarin/android/platform/android-job-scheduler>. Citado 2 vezes nas páginas 11 e 12.

MICROSOFT, C. *Criando um serviço - Xamarin _ Microsoft Docs*. Disponível em: <https://docs.microsoft.com/pt-br/xamarin/android/app-fundamentals/services/creating-a-service/>. Citado 4 vezes nas páginas 6, 7, 8 e 9.

MICROSOFT, C. *Service Class Overview*. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/api/android.app.service?view=xamarin-android-sdk-9>. Citado na página 8.

MICROSOFT, C. *ServiceAttribute Class (Android*. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/api/android.app.serviceattribute?view=xamarin-android-sdk-9>. Citado na página 8.