Creating a Timer, independent of System Time in Windows Phone

This article explains a number of approaches for creating timers that are independent of the Windows Phone system time.

Under Construction: This article is under construction and it may have outstanding issues. If you have any comments please use the comments tab.

Introduction

Sometimes an app needs to be able to detect whether a certain time has elapsed, independent of whether the app has been closed, the device been turned off, or even system time changes. One example of such a use case is an app which needs to block for a time after incorrect password entry - we don't want the user to be able to trick the system and gain too many opportunities to enter the password.

In this case we can't rely on app specific timers as they will be cleared when the app is closed or the device turned off, and we can't rely on comparing against the system time, because this can be easily altered by a user.

This article outlines a number of approaches that may be suitable for determining the elapsed time, depending on your use case.

Using network time

The code to lock a password screen for 30 minutes and penalise

```
IsolatedStorageSettings settings = IsolatedStorageSettings.ApplicationSettings;
DispatcherTimer timer;
DateTime currentServerTime ;
DateTime startSystemTime;

void LockUser(){
    settings["Locktime"] = currentServerTime = TimeFromServer().AddMinutes(30);
    startSystemTime = DateTime.Now;
    LockUI();
}

void LockUI(){
    // LocktheUI with a layer and show a timer.
    //...
    timer = new DispatcherTimer();
    timer.Interval = TimeSpan.FromSeconds(1);
    timer.Tick = new EventHandler(dispatcherTimer_Tick);
    dispatcherTimer.Start();
}

private void dispatcherTimer_Tick(object sender, EventArgs e)
{
    TimeSpan ElapsedTime = startSystemTime - DateTime.Now;
    if(ElapsedTime.Minutes<30)
    UITimerTextBlock.Text = (settings["Locktime"] as DateTime) -
currentTime.AddSeconds((double)ElapsedTime.Seconds); //  customise using a String Format
    else{
    timer.stop();
    UnlockUser();
    }
}
```


(C) Copyright Nokia 2014. All rights reserved.

```
void UnlockUser(){
    settings["Locktime"] = null;
    // UnLock the UI.
    //...
}

DateTime TimeFromServer(){
    //4 ways to do it

    //1 Get it from NIST
    GetNistTime();

    //2-getting time from google (html parsing) request on
https://www.google.com/search?q=utc+time&oq=utc+time and get the elements with the
classes "vk_bk" & "vk_ans"

    //3-getting time from an internet time services (list of time services
http://tf.nist.gov/tf-cgi/servers.cgi)
    // an example not tested
    /*
    var client = new TcpClient("64.90.182.55", 13);
    using (var streamReader = new StreamReader(client.GetStream()))
    {
        var response = streamReader.ReadToEnd();
        var utcDateTimeString = response.Substring(7, 17);
        return DateTime.ParseExact(utcDateTimeString, "yy-MM-dd hh:mm:ss",
CultureInfo.InvariantCulture, DateTimeStyles.AssumeUniversal);
    }
*/
//4-up your own webservice.

}

public void GetNistTime()
{
 HttpWebRequest request =
(HttpWebRequest)HttpWebRequest.Create("http://nist.time.gov/timezone.cgi?UTC/s/0");
 request.BeginGetResponse(GetRequestStreamCallback, request);
}

private void GetRequestStreamCallback(IAsyncResult result)
{
 HttpWebRequest request = result.AsyncState as HttpWebRequest;
 if (request != null)
 {
  try
  {
   HttpWebResponse response = (HttpWebResponse) request.EndGetResponse(result);
   DateTime dateTime;
   if (response.StatusCode == HttpStatusCode.OK)
   {
    StreamReader stream = new StreamReader(response.GetResponseStream());
    string html = stream.ReadToEnd().ToUpper();
    string time = Regex.Match(html, @">\d+:\d+:\d+<").Value; //HH:mm:ss format
    string date = Regex.Match(html, @">\w+,\s\w+\s\d+,\s\d+<").Value; //dddd, MMMM dd,
yyyy
```


(C) Copyright Nokia 2014. All rights reserved.

```
    dateTime = DateTime.Parse((date + " " + time).Replace(">", "").Replace("<", ""));
   }
  }
  catch (WebException e)
  {
  }
 }
}
```

Pro's of this approach

Network time is precisely correct and truly dependable.

Con's of this approach

Network is not always available. In that case, this approach fails.

Note: If your app is only usable with the network, you can go with this approach and wait until network is available in case if currently it's not.

Using TicksCount

According to MSDN documentation here, `TickCount` is a 32-bit signed integer containing the amount of time in milliseconds that has passed since the last time the computer was started. Value of `TickCount` property is derived from the system timer and is stored as a 32-bit signed integer. Consequently, if the system runs continuously, `TickCount` will increment from zero to `Int32.MaxValue` for approximately 24.9 days, then jump to `Int32.MinValue`, which is a negative number, then increment back to zero during the next 24.9 days.

```
int lockTickCount, curentTickCount, elapsedTimeInMiutes;

//while locking
Lock_User();
lockTickCount = Environment.TickCount & Int32.MaxValue;

//while checking for elapsed time
curentTickCount = Environment.TickCount & Int32.MaxValue;
elapsedTimeInMiutes = (lockTickCount - curentTickCount)/(1000*60);
if (elapsedTimeInMiutes >= 30)
{
  Unlock_User();
}
```

Pro's of using this approach

- `Environment.TickCount` is not effected by change in system time. It provides you amount of time in milliseconds that has passed since the last time the computer was started regardless of system time.

Con's of using this approach

- `TickCount` will reset at every system restart.
- Since `TickCount` is a signed integer, it can and will wrap roughly every 25 days. There is no way to prevent this from happening and if you try to you'll end up with bugs in your code.

Using periodic timers and notification services

```
protected override void OnInvoke(ScheduledTask task)
```


(C) Copyright Nokia 2014. All rights reserved.

```
{
  int elapsedTime, lockTime;

  //writes elapsed time to Isolated storage
  private void UpdateElapsedTime(int storageElapsedTime)
  {
    IsolatedStorageFile isoStore =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User | IsolatedStorageScope.Assembly,
null, null);
    using (IsolatedStorageFileStream isoStream = new
IsolatedStorageFileStream("TestStore.txt", FileMode.CreateNew, isoStore))
      {
        using (StreamWriter writer = new StreamWriter(isoStream))
        {
          writer.WriteLine(storageElapsedTime + "");
        }
      }
    }

  //reads and returns elapsedTime in minutes from isolated storage
  private int ReadElapsedTime()
  {
  string storageElapsedTime;
  IsolatedStorageFile isoStore = IsolatedStorageFile.GetStore(IsolatedStorageScope.User
| IsolatedStorageScope.Assembly, null, null);

  using (IsolatedStorageFileStream isoStream = new
IsolatedStorageFileStream("PeriodicAgentElapsedTime.txt", FileMode.Open, isoStore))
    {
      using (StreamReader reader = new StreamReader(isoStream))
      {
        storageElapsedTime = reader.ReadToEnd();
      }
    }
    return int.Parse(storageElapsedTime);
  }

  if (task is PeriodicTask)
  {
    elapsedTime = ReadElapsedTime() + 30;

    //if locktime is been served
    if (elapsedTime >= lockTime)
      unlock_user();
    else
      UpdateElapsedTime(elapsedTime + 30);
    }

    // Call NotifyComplete to let the system know the agent is done working.
    NotifyComplete();
}
```

Pro's of this approach

Although its not a very accurate approach for calculating elapsed time, it can be used for time estimation as `PeriodicTask` usually gets invoked close to every 30 minutes.

 (C) Copyright Nokia 2014. All rights reserved.

Con's of this approach

`PeriodicTask` doesn't run accurately after 30 minutes, it can run early or late based on system load or never run at all. So this approach is not reliable and can be used only for estimation.

Using Dispatcher Timer

```
int lockTime;

void OnTimerTick(Object sender, EventArgs args)
{
    unlock_user();
}

DispatcherTimer newTimer = new DispatcherTimer();

// timer interval specified as lockTime
newTimer.Interval = TimeSpan.FromMinutes(lockTime);

// Sub-routine OnTimerTick will be called at every 1 second
newTimer.Tick += OnTimerTick;

// starting the timer
newTimer.Start();
```

Pro's of this approach

`DispatcherTimer` is very accurate and has no dependency on current system time. So if Timer fires up, you can be assured of lock period being served and unlock the user.

Con's of this approach

For `DispatcherTimer` to run, the application process should be alive until timer fires. `DispatcherTimer` instance gets killed the moment user navigates away from the application which is a very normal case.

Mixed Approach

So to achieve this we will take advantage of few functions provided by the platform

1. Timer
2. Network
3. Periodic Task
4. Isolated Storage

```
//lock user if wrong password is entered 3 times
if (password wrong three times)
{
 lock_user();
 run x mins timer;
 run 30 mins periodic task;
}

//timer tick implies lock time is over
if (timer_ticks && user_is_locked())
{
 unlock_user();
```


(C) Copyright Nokia 2014. All rights reserved.

```
}

//if app closing while user is locked, write required data to isolated storage
if (app_closing)
{
 if (user_is_locked())
 {
  isolated_storage.write(elapsed_time_until_app_closed);
  isolated_storage.write(current_time);
 }
}

//every periodic task run implies 30 mins of elapsed time
if (periodic_task_ran && user_is_locked())
{
 isolated_storage.periodic_task_total_time = isolated_storage.periodic_task_total_time +
30;
 if ((isolated_storage.periodic_task_total_time +
isolated_storage.elapsed_time_until_app_closed) >= x)
  unlock_user();
}

//check if user is locked at app start
if (app_starting)
{
  if (user_is_locked())
  {
    // use network time if network is present else see if local time can be used
    if (network_present)
    {
      network_time = read_time_from_network;
      time_elapsed_after_app_closed = network_time - isolated_storage.current_time;
      total_elapsed_time = isolated_storage.elapsed_time_until_app_closed +
time_elapsed_after_app_closed;
      if(total_elapsed_time >= x && user_is_locked())
        {
        unlock_user();
        }
      else
        {
        start timer(x - total_elapsed_time);
        }
      }
      else
        {
        time_elapsed_after_app_closed = current_system_time -
isolated_storage.current_time;
        if (time_elapsed_after_app_closed > (
isolated_storage.elapsed_time_until_app_closed +
isolated_storage.periodic_task_total_time + 30))
          {
          //user changed time, punish him by increasing lock time
          }
        else
          {
          total_elapsed_time = isolated_storage.elapsed_time_until_app_closed +
time_elapsed_after_app_closed;
```


(C) Copyright Nokia 2014. All rights reserved.

```
          if(total_elapsed_time >= x && user_is_locked())
            {
            unlock_user();
            }
          else
            {
            start timer(x - total_elapsed_time);
            }
          }
        }
      }
}

lock_user()
{
  isolated_storage.write(user_locked);
}

user_is_locked()
{
  return isolated_storage.exists(user_locked);
}

unlock_user()
{
  isolated_storage.remove(periodic_task_total_time);
  isolated_storage.remove(elapsed_time_until_app_closed);
  isolated_storage.remove(current_time);
  isolated_storage.remove(user_locked);
}
```

Penalising user change time approach

This mechanism is described here.

The method stores the system time when the password is expected to expire and the last-run system time. If the system time is pushed forward past the password expiry time the user will be able to enter a new password. However at some point they will want to use the app with "correct" system time - the app detects that it was last run "in the future" and will block again, for an even longer period.

Effectively this approach relies on the fact that users will want to use the app without having to change time a lot.

Summary

None of the approaches described here are perfect - they all have advantages and limitations. Most ideas originated in this discussion.

(C) Copyright Nokia 2014. All rights reserved.

 (C) Copyright Nokia 2014. All rights reserved.