FAKULTA
ELEKTROTECHNICKÁ
ČVUT V PRAZE

Virtuální Instrumentace (B3M38VIN)

Implementace NTP protokolu do SW aplikace v C/C++

1. Synchronizace času pomocí NTP

NTP se používá pro synchronizaci času v sítích s proměnnou odezvou již od roku 1985 a to z něj dělá jeden z nejstarších internetových protokolů. Používá UDP protokol na 4. vrstvě OSI a port 123. Standardně dosahuje přesnosti 10 ms až 200 µs, v závislosti na kvalitě připojení.

NTP používá hierarchický systém tzv. „*stratum*“. Server typu *stratum* 0 získává ten nejpřesnější čas např. z cesiových hodin, ale není určený pro distribuci času do sítě. K tomu slouží server typu *stratum* 1, který přijímá čas ze *strata* 0. Dále pak existují servery *stratum* 2 až 15, které vždy získávají čas od nadřazeného serveru a jejich číslo v podstatě ukazuje vzdálenost od referenčních hodin.

Algoritmus NTP začíná vysláním definovaného paketu (RFC 5905), respektive datagramu, od klienta na server. Nejdůležitější informace předané tímto paketem jsou režim klienta (NTPv4), *stratum* lokálních hodin, přesnost lokálních hodin, a především čas **T1**, který značí čas lokálních hodin v době odchodu paketu do sítě. Po přijmutí paketu NTP serverem do něj server zapíše čas **T2**, který značí aktuální čas na hodinách serveru a těsně před odesláním čas **T3**, který značí čas odchodu paketu zpět do sítě. Po přijetí paketu klientem se konečně zapíše poslední čas **T4**, který značí příchod zpět ke klientovi. Pokud jsou tyto časy změřeny přesně, stačí díky vzorcům níže vypočítat dvě výsledné hodnoty. **Offset**, který symbolizuje posun hodin klienta od hodin na serveru a **Delay**, který představuje zpoždění průchodu paketu sítí, které může být díky přepínačům a síťovým technologiím značně variabilní. Součet těchto hodnot pak představuje finální posun lokálních hodin, který by měl být v ideálním případě roven nule.

$$Offset = \frac{(T2 - T1) + (T3 - T4)}{2}$$

$$Delay = (T4 - T1) + (T3 - T2)$$

$$Delta = Offset + Delay$$

2. NTP klient – knihovna (C++ DLL)

- **Popis**

Vyvinul jsem jednoduchou a jednoúčelovou knihovnu v jazyce C++ v prostředí Microsoft Visual Studio 2019. Vycházel jsem pouze z oficiální specifikace RFC 5905. Knihovna je aktuálně určena pro systém Windows NT, protože používá Win32 API pro čtení a zápis systémového času a *Winsock* pro UDP komunikaci. Nicméně v budoucnu ji není problém rozšířit pomocí direktiv `#ifdef` např. o POSIX *sockety*.

Jelikož knihovna obsahuje pouze jednu třídu **Client**, třídní diagram je zbytečný.

```
class Client : public IClient
```

Knihovna disponuje pouze dvěma veřejnýma metodama, **query** a **query_and_sync**.

```
virtual Status query(const char* hostname, ResultEx** result_out);
virtual Status query_and_sync(const char* hostname, ResultEx** result_out);
```

Query je jádrem celé knihovny. Na začátku této metody se nejprve vytvoří UDP paket, vyplní se aktuálními hodnotami, které jsem zmínil v první kapitole a odešle jej na NTP server. Po příchodu doplní čas T4 a provede výpočet, dle vzorce z první kapitoly. Časy jsou reprezentovány třídou `time_point` z knihovny `std::chrono` s rozlišením na nanosekundy (čas **t1**) a nebo třídou `high_resolution_clock` (čas **t1_b**).

```
typedef std::chrono::time_point<std::chrono::system_clock,
std::chrono::nanoseconds> time_point_t;

time_point_t t1 = std::chrono::time_point_cast<std::chrono::nanoseconds>
(std::chrono::system_clock::now());

auto t1_b = std::chrono::high_resolution_clock::now();
```

Tato kombinace je z toho důvodu, že časy v prvním vzorci (offset) musí být absolutní. Jedná se o časy **T2** a **T3**, které přišly ze serveru. nelze tedy použít `high_resolution_clock`, v druhém vzorci (delay) pak lze pro odečet **T1** od **T4** použít časy relativní, který lze získat pomocí `high_resolution_clock`. Následující vzorce ukazují výpočet pomocí tohoto přístupu, jednotky všech proměnných jsou nanosekundy.

```
offset = [(T2 - T1) + (T3 - T4)] / 2
delay = (T4b - T1b) - (T3 - T2)
```

Sečtením hodnot *offset* a *delay* získáme *delta*, tedy hodnotu, o kterou upravíme lokální systémové hodiny. To ovšem pouze v případě použití druhé veřejné metody **query_and_sync**, první zmíněná provede jen komunikaci se serverem a výpočet.

Výsledné vypočtené a získané hodnoty, včetně *jitteru* (ukazatele stability síťového připojení) se vrací uživateli buď ve struktuře `Result`, která slouží pro klasické C rozhraní, nebo ve třídě `ResultEx`, která na rozdíl od první obsahuje čas reprezentovaný třídou `time_point_t`, oproti času reprezentovaném klasickou strukturou `TimePt` s *integery*.

```
struct Result
{
    struct TimePt time;
    struct Metrics mtr;
};
```

```
class ResultEx
{
public:
    time_point_t time;
    Metrics mtr;
};
```

Tím je docílena kompatibilita mezi C a C++, která je u dynamické knihovny nutná. Pokud uživatel použije knihovnu přímo z C++, je výhodnější pracovat s časem reprezentovaným třídou `time_point_t`, v opačném případě nezbývá než použít strukturu.

```
struct TimePt
{
    int tm_nsec;
    int tm_usec;
    int tm_msec;
    int tm_sec;
    int tm_min;
    int tm_hour
    int tm_mday;
    int tm_mon;
    int tm_year;
};
```

```
struct Metrics
{
    double delay_ns;
    double offset_ns;
    double jitter_ns;
    double delta_ns;
};
```

Chybové stavy jsou vráceny jako *enumerátor* `Status`, kdy 0 znamená úspěch (podobně jako v POSIX) a cokoliv ostatního je chyba.

```
enum Status : int16_t
{
    OK = 0,
    UNKNOWN_ERR = 1,
    INIT_WINSOCK_ERR = 2,
    CREATE_SOCKET_ERR = 3,
    SEND_MSG_ERR = 4,
    RECEIVE_MSG_ERR = 5,
    RECEIVE_MSG_TIMEOUT = 6,
    SET_WIN_TIME_ERR = 7,
    ADMIN_RIGHTS_NEEDED = 8
};
```

Dále obsahuje knihovna několik statických bez-stavových metod pro ulehčení práce programátora, sloužící především k formátování výsledků a konverzi typů.

- **C++ rozhraní**

Standardní rozhraní knihovny pro použití s objektově orientovanými jazyky je ve formě *interface*, který vystavuje výše popsané dvě hlavní veřejné metody **query** a **query_and_sync**. `Interface` je pouze makro pro typ `struct`, samozřejmě by šlo použít proprietární MS `__interface`, ale většinou je se lepší držet osvědčených a kompatibilních věcí.

```
Interface IClient
{
    virtual Status query(const char* hostname, ResultEx** result_out) = 0;
    virtual Status query_and_sync(const char* hostname, ResultEx**result_out) = 0;
    virtual ~IClient() {};
};
```

- **C rozhraní**

Rozhraní použitelné pro volání DLL musí být kompatibilní s klasickým ANSI C, místo tříd je tak nutné použít klasický C OOP styl, a sice funkce, struktury a *opaque pointery*. Tyto funkce je pak třeba exportovat pomocí makra `EXPORT`, které je makro pro `__declspec(dllexport)`. Dále je nutné nastavit adekvátní volací konvenci, v našem případě se jedná o `__cdecl`, kdy ten kdo volá také uklízí zásobník.

Funkce **`Client__create`** vytvoří instanci knihovny, ta je reprezentována ukazatelem, respektive makrem, `HNTP`, kterému se v kontextu Windows říká *handle*.

```
typedef void* HNTP;
```

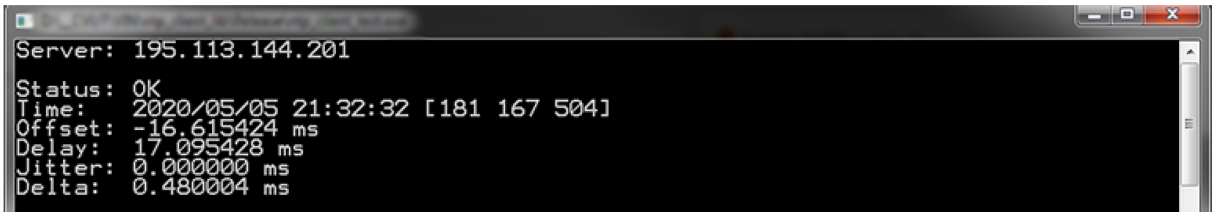
Ostatní funkce, například **`Client__query`**, nebo **`Client__query_and_sync`** berou tento ukazatel jako první argument. Zbytek už je velice podobný C++ rozhraní, nicméně jeden rozdíl to má. Místo `delete` se musí na konci zavolat **`Client__free_result`** a **`Client__close`**.

```
extern "C"
{
    /* object lifecycle */
    EXPORT HNTP __cdecl Client__create(void);
    EXPORT void __cdecl Client__close(HNTP self);
    /* main NTP server query functions */
    EXPORT enum Status __cdecl Client__query(HNTP self, const char* hostname,
    struct Result** result_out);
    EXPORT enum Status __cdecl Client__query_and_sync(HNTP self, const char*
    hostname, struct Result** result_out);
    /* helper functions */
    EXPORT void __cdecl Client__format_info_str(struct Result* result, char*
    str_out);
    EXPORT void __cdecl Client__get_status_str(enum Status status, char*
    str_out);
    EXPORT void __cdecl Client__free_result(struct Result* result);
}
```

- **Použití**

Pro spuštění je nutné mít nainstalovaný *runtime* **vc__redist** (2015-19). Kód je alespoň částečně okomentovaný a snad i přehledný. Snažil jsem se, aby bylo použití triviální. Vytvoří se instance klienta, zavolá se funkce *query*, ukončí se klient. Toto se může vykonávat v nekonečné smyčce s definovaným intervalem, aby se zajistila stála časová synchronizace. Následující řádky jsou vyňaty z konzolové aplikace, která slouží jako příklad použití.

```
enum Status s;
struct Result* result = nullptr;
HNTP client = Client__create()
s = Client__query_and_sync(client, "195.113.144.201", &result);
Client__free_result(result);
Client__close(client);
```



3. NTP klient – grafické rozhraní (CVI)

Dynamickou knihovnu jsem použil v prostředí LabWindows/CVI pro vytvoření grafického rozhraní NTP klienta, který se periodicky volá z vlastního vlákna. Na grafu pak vidíme zeleně hodnotu delta (aktuální rozdíl lokálních hodin od serveru), žlutě její průměr a červeně *jitter* síťové komunikace. Pro spuštění je nutný *CVI Runtime 2019*.