```
/*
 * include/parameter.h
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#pragma once

#include "utils.h"
#include "except.h"

namespace EasySip
{
    class Buffer
    {
    protected:

        char *data_;
        size_t len_;

    public:
        Buffer(size_t len) : len_(len)
        {
            data_ = new char[len_];
            memset(data_, 0, len_);
        }

        ~Buffer()
        {
            len_ = 0;
            if (data_) delete data_;
            data_ = 0;
        }

        char* data()
        {
            return data_;
        }

        size_t len()
        {
            return len_;
        }
    };
} // namespace EasiSip
```

```
/*
 * include/timer.h
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#pragma once

#include <sys/time.h>
#include <signal.h>
#include "thread.h"

namespace EasySip
{
    /*
     * void timeradd(struct timeval *a, struct timeval *b, struct timeval *res);
     * void timersub(struct timeval *a, struct timeval *b, struct timeval *res);
     * void timerclear(struct timeval *tvp);
     * int timerisset(struct timeval *tvp);
     * int timercmp(struct timeval *a, struct timeval *b, CMP);
     */
    extern bool operator== (struct itimerval &a, struct itimerval &b);
    extern bool operator!= (struct itimerval &a, struct itimerval &b);
    extern std::ostream& operator<< (std::ostream &o, struct timeval &a);
    extern std::ostream& operator<< (std::ostream &o, struct itimerval &a);
    extern std::ostream& operator<< (std::ostream &o, struct timespec &a);
    extern std::ostream& operator<< (std::ostream &o, struct itimerspec &a);

    class Time
    {
        time_t time_;
    public:
        static std::string now();
    };

//    void sigalrm_cb(int signo)
//    {
//        std::cout << "------------time's up-----------------\n";
//        std::cout << "signo: " << signo << "\n";//" settimer: " << setitimer(ITIMER_REAL, 0, &it_a) << '\n';
//        struct itimerval cur;
//
//        if (0 <= getitimer(ITIMER_REAL, &cur))
//                std::cout << cur << '\n';
//
////        timerclear(&cur.it_value);
////        timerclear(&cur.it_interval);
////        std::cout << cur << '\n';
//
//        std::cout << "++++++++++++time's up+++++++++++++++++\n";
//    }
//
//    void sigev_notify_cb(union sigval sigev_value)
//    {
//        std::cout << "------------time's up-----------------\n";
//        std::cout << "sigval.sival_int: [" << sigev_value.sival_int << "]\n";
////        std::cout << "timer id: [" << *(time_t*)data << "]\n";
//        std::cout << "++++++++++++time's up+++++++++++++++++\n";
//    }

    class Timer
    {
        unsigned long value_; // in  ms
        struct itimerval itv_;

    public:

        typedef Timer Base;

        Timer(unsigned long value)//unsigned long value /* ms */)
        :value_(value)
        {
//          signal(SIGALRM, sigalrm_cb);

            time_t sec = value_/1000;
            suseconds_t usec = (value_ % 1000) * 1000;

              itimev(sec, usec);
        }

        Timer(time_t sec, suseconds_t usec = 0)
        :value_(sec*1000 + usec/1000)
        {
//          signal(SIGALRM, sigalrm_cb);
//          signal(SIGVTALRM, sigalrm_cb);
//          signal(SIGPROF, sigalrm_cb);

              itimev(sec, usec);
        }

          Timer& itimev(time_t sec, suseconds_t usec = 0)
        {
            itv_.it_interval.tv_sec = sec;
            itv_.it_interval.tv_usec = usec;
            itv_.it_value.tv_sec = sec;
            itv_.it_value.tv_usec = usec;

            return *this;
        }

        Timer(std::string value)
        {
            value_ = time_string_to_ulong(value);
            time_t sec = value_/1000;
            suseconds_t usec = (value_ % 1000) * 1000;
              itimev(sec, usec);
        }

        ~Timer()
        {
        }

        void value(std::string value)
        {
            value_ = time_string_to_ulong(value);
        }

//        std::string value()
//        {
//            return timer_ulong_to_string(value_);
//        }

        unsigned long value()
        {
            return value_;
        }

        void value(unsigned long value)
        {
            value_ = value;
        }

//        ITIMER_REAL    decrements in real time, and delivers SIGALRM upon expiration.
//        ITIMER_VIRTUAL decrements only when the process is executing, and delivers SIGVTALRM upon expiration.
//        ITIMER_PROF    decrements both when the process executes and when the system is executing on behalf of
//                       the process. Coupled with ITIMER_VIRTUAL, this timer is usually used to profile the
//                       time spent by the application in user and kernel space.  SIGPROF is delivered upon expiration.
//
```

```
//      CLOCK_REALTIME
//             A settable system-wide real-time clock.
//      CLOCK_MONOTONIC
//             A nonsettable monotonically increasing clock that measures time from some unspecified point in the past
//             that does not change after system startup.
//      CLOCK_PROCESS_CPUTIME_ID (since Linux 2.6.12)
//             A clock that measures (user and system) CPU time consumed by (all of the threads in) the calling process.
//      CLOCK_THREAD_CPUTIME_ID (since Linux 2.6.12)
//             A clock that measures (user and system) CPU time consumed by the calling thread.

//      int timer_create(clockid_t clockid, struct sigevent *sevp, timer_t *timerid);
//      int timer_settime(timer_t timerid, int flags, const struct itimerspec *new_value, struct itimerspec * old_value);
//      int timer_gettime(timer_t timerid, struct itimerspec *curr_value);

        virtual void start()
        {
//-----------------------------------------------------------------------------------
            int t_id = ITIMER_REAL;//VIRTUAL;

            std::cout << "settimer: " << setitimer(t_id, &itv_, 0) << '\n';
            struct itimerval cur;

            getitimer(t_id, &cur);

            std::cout << itv_ << "|" << cur << '\n';
//-----------------------------------------------------------------------------------
//            int ret;
//            timer_t tm_id;
//
//            struct sigevent sevp;
//            sevp.sigev_notify = SIGEV_THREAD;
//            sevp.sigev_notify_function = sigev_notify_cb;
//            sevp.sigev_value.sival_ptr = &tm_id;
//
//            if (0 > (ret = timer_create(CLOCK_REALTIME, &sevp, &tm_id)))
//                std::cout << "timer_create: " << ret << ' ' << strerror(errno) << '\n';
//
//           std::cout << "tm_id: [" << tm_id << "]\n";
//
//           struct itimerspec itspec;
//
//           itspec.it_value.tv_sec = 3;
//           itspec.it_value.tv_nsec = 0;
//           itspec.it_interval.tv_sec = 3;
//           itspec.it_interval.tv_nsec = 0;
//
//           if (0 > (ret = timer_settime(tm_id, 0, &itspec, 0)))
//              std::cout << "timer_settime: " << ret << ' ' << strerror(errno) << '\n';
//
//           std::cout << "itspec: [" << itspec << "]\n";
//
//           struct itimerspec itscur;
//
//           if (0 > (ret = timer_gettime(tm_id, &itscur)))
//              std::cout << "timer_gettime: " << ret << ' ' << strerror(errno) << '\n';
//
//           std::cout << "itscur: [" << itscur << "]\n";
//-----------------------------------------------------------------------------------
        }

        static unsigned long time_string_to_ulong(std::string value)
        {
            // TODO: string value -> long value
            return 0;
        }

        static unsigned long time_ulong_to_string(unsigned long value)
        {
```

```
        // TODO: string value <- long value
        return 0;
    }

    unsigned long operator* (unsigned long val)
    {
        return (value_*val);
    }
};

// built-in timers
class T1 : public Timer
{
public:

    T1() : Timer("500")//ms")
    {
    }
};

class T2 : public Timer
{
public:

    T2() : Timer("4000")
    {
    }
};

class T4 : public Timer
{
public:

    T4() : Timer("5000")
    {
    }
};

// INVITE_RETRAN_INTERVAL
class TA : public Timer
{
public:

    TA() : Timer(T1().value())//ms") // TODO: T1 initial value
    {
    }
};

class TB : public Timer
{
public:

    TB() : Timer(T1()*64) // TODO: T1*64
    {
    }
};

class TC : public Timer // 4min
{
public:

    TC() : Timer((unsigned long)4*60*1000) // TODO: > 3min
    {
    }

    void value(std::string value)
    {
        // TODO: check >3min
        Base::value(value);
    }
};

class TD : public Timer
{
public:

    TD() : Timer("33000") // TODO: UDP: >32s, TCP/SCTP =0s
    {
    }
};

class TE : public Timer
{
public:

    TE() : Timer("500") // TODO: T1 initial value
    {
    }
};

class TF : public Timer
{
public:

    TF() : Timer(T1()*64) // TODO: T1*64
    {
    }
};

class TG : public Timer
{
public:

    TG() : Timer("500ms") // TODO: T1 initial value
    {
    }
};

class TH : public Timer
{
public:

    TH() : Timer(T1()*64) // TODO: T1*64
    {
    }
};

class TI : public Timer
{
public:

    TI() : Timer(T4().value()) // TODO: UDP: T4, TCP/SCTP =0s
    {
    }
};

class TJ : public Timer
{
public:

    TJ() : Timer(T1()*64) // TODO: UDP: 64*T1, TCP/SCTP =0s
    {
    }
};

class TK : public Timer
```

```
    {
    public:

        TK() : Timer(T4().value()) // TODO: UDP: 64*T1, TCP/SCTP =0s
        {
        }
    };
} // namespace EasySip
```

```
/*
 * include/header_field.h
 *
 * Author: Zex <top_zlynch@yahoo.com>
 *
 * References:
 *        Session Initiation Protocol (Sip) Parameters, IANA
 *        RFC-3261
 *        RFC-6665
 *       SIP, Understanding The Session Initiation Protocol, 2nd Ed, Artech House
 */
#pragma once

#include "uri.h"
#include "response_code.h"
#include "request_message.h"

namespace EasySip
{
    #define SIP_VERSION_1_0 "SIP/1.0"
    #define SIP_VERSION_2_0 "SIP/2.0"
    #define SIP_VERSION_2_0_UDP SIP_VERSION_2_0"/UDP"
    #define SIP_VERSION SIP_VERSION_2_0

    #define return_false_if_true(c) \
    {                              \
        if ((c)) return false;    \
    }

    #define ONE_HOUR 60*60 // in second

    enum
    {
        HF_CALLID = 1,
        HF_CSEQ,
        HF_FROM,
        HF_TO,
        HF_VIA,
        HF_ALERT_INFO,
        HF_ALLOW_EVENTS,
        HF_DATE,
        HF_CONTACT,
        HF_ORGANIZATION,
        HF_RECORD_ROUTE,
        HF_RETRY_AFTER,
        HF_SUBJECT,
        HF_SUPPORTED,
        HF_TIMESTAMP,
        HF_USER_AGENT,
        HF_ANSWER_MODE,
        HF_PRIV_ANSWER_MODE,
        HF_ACCEPT,
        HF_ACCEPT_CONTACT,
        HF_ACCEPT_ENCODING,
        HF_ACCEPT_LANGUAGE,
        HF_AUTHORIZATION,
        HF_CALL_INFO,
        HF_EVENT,
        HF_IN_REPLY_TO,
        HF_JOIN,
        HF_PRIORITY,
        HF_PRIVACY,
        HF_PROXY_AUTHORIZATION,
        HF_PROXY_REQUIRE,
        HF_P_OSP_AUTHTOKEN,
        HF_PASSERTED_IDENTITY,
        HF_PPREFERRED_IDENTITY,
        HF_MAX_FORWARDS,
        HF_REASON,
        HF_REFER_TO,
        HF_REFERRED_BY,
        HF_REPLY_TO,
        HF_REPLACES,
        HF_REJECT_CONTACT,
        HF_REQUEST_DISPOSITION,
        HF_REQUIRE,
        HF_ROUTE,
        HF_RACK,
        HF_SESSION_EXPIRES,
        HF_SUBSCRIPTION_STATE,
        HF_AUTHENTICATIONINFO,
        HF_ERROR_INFO,
        HF_MIN_EXPIRES,
        HF_MIN_SE,
        HF_PROXY_AUTHENTICATE,
        HF_SERVER,
        HF_UNSUPPORTED,
        HF_WARNING,
        HF_WWW_AUTHENTICATE,
        HF_RSEQ,
        HF_ALLOW,
        HF_CONTENT_ENCODING,
        HF_CONTENT_LENGTH,
        HF_CONTENT_DISPOSITION,
        HF_CONTENT_LANGUAGE,
        HF_CONTENT_TYPE,
        HF_EXPIRES,
        HF_MIME_VERSION,
    };

    struct HeaderField
    {
        std::string field_;
        std::string compact_form_;
        std::string values_;
        Parameters header_params_;
        bool is_hop_by_hop_;

        HeaderField(std::string f, bool is_hbh = false)
        : field_(f), is_hop_by_hop_(is_hbh)
        {
        }

        HeaderField(std::string f, std::string c, bool is_hbh = false)
        : field_(f), compact_form_(c), is_hop_by_hop_(is_hbh)
        {
        }

        HeaderField()
        {
        }

        ~HeaderField()
        {
        }

        std::string Compact()
        {
            return compact_form_;
        }

        std::string Field()
        {
            return field_;
        }
        bool is_value_valid()
        {
            return true;
        }

        virtual void generate_values() = 0;

        virtual int parse(std::string &msg, size_t &pos) = 0;

        std::string Values()
        {
            return values_;
        }

        HeaderField& HeaderParam(std::string n, std::string v)
        {
            header_params_.set_value_by_name(n, v);
            return *this;
        }

        friend std::ostream& operator<< (std::ostream& o, HeaderField& hf);

        std::string operator() ();

        void remove_tail_symbol(char sym)
        {
            if (values_.size() && values_.at(values_.size()-1) == sym)
                values_.erase(values_.size()-1);
        }
    };

    struct HFBase_1_ : public HeaderField
    {
        ContactList cons_;

        HFBase_1_(std::string f, bool is_hbh = false) : HeaderField(f, is_hbh)
        {
        }

        HFBase_1_(std::string f, std::string c, bool is_hbh = false) : HeaderField(f, c, is_hbh)
        {
        }

        virtual void generate_values();
        virtual int parse(std::string &msg, size_t &pos);

        virtual HFBase_1_& add_value(std::string)
        {
            return *this;
        }

        HFBase_1_& add_param(std::string key, std::string value = "")
        {
            if (!cons_.empty())
                cons_.last()->add_param(key, value);

            return *this;
        }

        HFBase_1_& add_uri(std::string uri)
        {
            if (cons_.empty() || cons_.last()->full() || !cons_.last()->uri().empty())
            {
                cons_.append_item();
            }

            if (cons_.last()->uri().empty())
            {
                cons_.last()->uri(uri);
            }

            return *this;
        }

        HFBase_1_& add_name(std::string name)
        {
            if (cons_.empty() || cons_.last()->full())
            {
                cons_.append_item();
            }

            if (cons_.last()->name().empty())
            {
                cons_.last()->name(name);
            }

            return *this;
        }
    };

    struct HFBase_2_ : public HeaderField
    {
        std::string digit_value_;

        HFBase_2_(std::string f, bool is_hbh = false) : HeaderField(f, is_hbh)
        {
        }

        HFBase_2_(std::string f, std::string c, bool is_hbh = false) : HeaderField(f, c, is_hbh)
        {
        }

        virtual void generate_values();
        virtual int parse(std::string &msg, size_t &pos);

        virtual HFBase_2_& add_value(std::string val)
        {
            digit_value_ = val;
            return *this;
        }
    };

    struct HFBase_3_ : public HeaderField
    {
        std::vector<std::string> opts_;
        char sym_;

        HFBase_3_(std::string f, bool is_hbh = false) : HeaderField(f, is_hbh)
        {
            sym_ = ',';
        }

        HFBase_3_(std::string f, std::string c, bool is_hbh = false) : HeaderField(f, c, is_hbh)
        {
            sym_ = ',';
        }

        virtual void generate_values();
        virtual int parse(std::string &msg, size_t &pos);

        virtual HFBase_3_& add_value(std::string val)
        {
            opts_.push_back(val);
            return *this;
        }
        virtual HFBase_3_& add_value(std::vector<std::string> &vals)
        {
            std::copy(opts_.begin(), vals.begin(), vals.end());
            return *this;
        }

    };

    struct HFBase_4_ : public HeaderField
    {
        PtsOf<ItemWithParams> its_;

        HFBase_4_(std::string f, bool is_hbh = false) : HeaderField(f, is_hbh)
        {
        }

        HFBase_4_(std::string f, std::string c, bool is_hbh = false) : HeaderField(f, c, is_hbh)
        {
        }

        virtual void generate_values();
        virtual int parse(std::string &msg, size_t &pos);

        HFBase_4_& add_value(std::string val)
        {
            ItemWithParams it(val);
            its_.append_item(it);
            return *this;
        }

        HFBase_4_& add_param(std::string key, std::string val = "")
        {
            if (!its_.empty())
                its_.last()->add_param(key, val);
            return *this;
        }
    };

    struct HFBase_5_ : public HeaderField
    {
        std::string challenge_;
        Parameters digest_cln_;

        HFBase_5_(std::string f, bool is_hbh = false) : HeaderField(f, is_hbh)
        {
            digest_cln_.Sym(",");
        }

        HFBase_5_(std::string f, std::string c, bool is_hbh = false) : HeaderField(f, c, is_hbh)
        {
            digest_cln_.Sym(",");
        }

        virtual void generate_values();
        virtual int parse(std::string &msg, size_t &pos);

        HFBase_5_& add_value(std::string val)
        {
            challenge_ = val;
            return *this;
        }

        HFBase_5_& add_param(std::string key, std::string val = "")
        {
            digest_cln_.append(key, val);
            return *this;
        }
    };
```

```
// ---------- Mandatory fields ---------------
/* Call-ID: 19283kjhj5h
 */
struct HFCallId : public HFBase_3_
{
    HFCallId() : HFBase_3_("Call-ID", "i")
    {
        sym_ = ' ';
    }

    bool operator== (HFCallId& val)
    {
        return (id() == val.id());
    }

    HFCallId& id(std::string val)
    {
        if (opts_.empty())
            HFBase_3_::add_value(val);
        else
            opts_.at(0) = val;

        return *this;
    }

    std::string id()
    {
        if (opts_.size())
            return opts_.at(0);
        return std::string();
    }
};

/* CSeq: 35246 INVITE
 */
struct HFCSeq : public HFBase_3_
{
    HFCSeq() : HFBase_3_("CSeq")
    {
        sym_ = ' ';
    }

    HFCSeq& cseq(std::string val)
    {
        if (opts_.empty())
            HFBase_3_::add_value(val);
        else
            opts_.at(0) = val;

        return *this;
    }

    HFCSeq& method(std::string val)
    {
        if (2 > opts_.size())
            HFBase_3_::add_value(val);
        else
            opts_.at(1) = val;

        return *this;
    }

    std::string cseq()
    {
        if (opts_.size())
            return opts_.at(0);
        return std::string();
    }

    std::string method()
    {
        if (1 < opts_.size())
            return opts_.at(1);
        return std::string();
    }

    void inc_seq()
    {
        unsigned int seq;

        std::istringstream i(cseq());
        i >> seq;
        seq++;

        std::ostringstream o;
        o << seq;
        cseq(o.str());
    }
};


/* From: Alice <sip:alice@atlanta.com>;tag=87263237
 */
struct HFFrom : public HFBase_1_
{
    HFFrom() : HFBase_1_("From", "f")
    {
    }

    std::string name()
    {
        if (cons_.empty())
            return std::string();
        return cons_.at(0)->name();
    }

    std::string uri()
    {
        if (cons_.empty())
            return std::string();
        return cons_.at(0)->uri();
    }

    std::string tag()
    {
        return header_params_.get_value_by_name("tag");
    }
};

/* To: Alice <sip:alice@atlanta.com>;tag=39u292sd7
 */
struct HFTo : public HFBase_1_
{
    HFTo() : HFBase_1_("To", "t")
    {
    }

    std::string name()
    {
        if (cons_.empty())
            return std::string();
        return cons_.at(0)->name();
    }

    std::string uri()
```

```
        {
            if (cons_.empty())
                return std::string();
            return cons_.at(0)->uri();
        }

        std::string tag()
        {
            return header_params_.get_value_by_name("tag");
        }
    };

    /* Via: SIP/2.0/UDP <aa.atlanta.com>;branch=38Z89sdhJ;received=192.168.0.50
     * Via: SIP/2.0/UDP <cc.atlanta.com>;branch=2998H933k;received=192.168.0.43
     * Via: SIP/2.0/UDP 135.180.130.133
     * ...
     */
    struct HFVia : public HeaderField
    {
        std::string sent_proto_;
        std::string sent_by_;

        HFVia();

        void generate_values();
        int parse(std::string &msg, size_t &pos);

        HFVia& add_proto(std::string proto)
        {
            sent_proto_ = proto;
            return *this;
        }

        HFVia& add_sentby(std::string by)
        {
            sent_by_ = by;
            return *this;
        }
    };

    // ------------------ Optional fields --------------------

    /* Alert-Info: <http://wwww.example.com/alice/photo.jpg> ;purpose=icon,
     *        <http://www.example.com/alice/> ;purpose=info
     */
    struct HFAlertInfo : public HeaderField
    {
        HFAlertInfo() : HeaderField("Alert-Info", true)
        {
//            header_params_.append("appearance");
//            header_params_.append("purpose");
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFAllowEvents : public HeaderField
    {
        HFAllowEvents() : HeaderField("Allow-Events", "u")
        {
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFDate : public HFBase_3_
    {
        HFDate() : HFBase_3_("Date", true)
        {
            sym_ = ' ';
        }
    };

    /* Contact: <sip:user@example.com?Route=%3Csip:sip.example.com%3E>
     */
    struct HFContact : public HFBase_1_
    {
        HFContact();

        ContactList& cons()
        {
            return cons_;
        }
    };

    struct HFOrganization : public HFBase_3_
    {
        HFOrganization() : HFBase_3_("Organization", true)
        {
            sym_ = ' ';
        }
    };

    /* Record-Route: <sip:+1-650-555-2222@iftgw.there.com;
     *         maddr=ss1.wcom.com>
     * Record-Route: <sip:139.23.1.44;lr>
     */
    struct HFRecordRoute : public HFBase_1_
    {
        HFRecordRoute() : HFBase_1_("Record-Route", true)
        {
        }
    };

    struct HFRetryAfter : public HeaderField
    {
        HFRetryAfter() : HeaderField("Retry-After")
        {
//            header_params_.append("duration");
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFSubject : public HFBase_3_
    {
        HFSubject() : HFBase_3_("Subject", "s")
        {
            sym_ = ' ';
        }
    };

    struct HFSupported : public HFBase_3_
    {
        HFSupported() : HFBase_3_("Supported", "k")
        {
            sym_ = ' ';
        }
    };

    struct HFTimestamp : public HeaderField
    {
        HFTimestamp() : HeaderField("Timestamp")
        {
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFUserAgent : public HeaderField
    {
        HFUserAgent() : HeaderField("User-Agent")
        {
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFAnswerMode : public HeaderField
    {
        HFAnswerMode() : HeaderField("Answer-Mode")
        {
//              header_params_.append("require");
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFPrivAnswerMode : public HeaderField
    {
        HFPrivAnswerMode() : HeaderField("Priv-Answer-Mode")
        {
//              header_params_.append("require");
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    // -------------------- Request header -----------------------------
    struct HFAccept : public HFBase_4_
    {
        HFAccept() : HFBase_4_("Accept") // type/sub-type
        {
//              header_params_.append("q");
        }

        HFAccept& add_value(std::string val)
        {
            if (val.find_first_of("/") == std::string::npos)
                return *this;

            HFBase_4_::add_value(val);

            return *this;
        }

        HFAccept& add_value(std::string ty, std::string subty)
        {
            ty += "/";
            ty += subty;

            HFBase_4_::add_value(ty);

            return *this;
        }
    };

    struct HFAcceptContact : public HeaderField
    {
        HFAcceptContact() : HeaderField("Accept-Contact", "a")
        {
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFAcceptEncoding : public HFBase_4_
    {
        HFAcceptEncoding() : HFBase_4_("Accept-Encoding")
        {
//              header_params_.append("q");
        }

    };

    struct HFAcceptLanguage : public HFBase_4_
    {
        HFAcceptLanguage() : HFBase_4_("Accept-Language")
        {
//              header_params_.append("q");
        }
    };

    struct HFAuthorization : public HFBase_5_
    {
        HFAuthorization();
    };

    struct HFCallInfo : public HFBase_1_
    {

        HFCallInfo();

        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFEvent : public HeaderField
    {
        HFEvent();

        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFInReplyTo : public HeaderField
    {
        HFInReplyTo() : HeaderField("In-Reply-To")
        {
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFJoin : public HeaderField
    {
        HFJoin() : HeaderField("Join")
        {
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    /*
     * Priority: non-urgent
     * Priority: normal
     * Priority: urgent
     * ...
     */
    struct HFPriority : public HFBase_3_
    {
        HFPriority() : HFBase_3_("Priority", true)
        {
            sym_ = ' ';
        }
    };

    struct HFPrivacy : public HeaderField
    {
        HFPrivacy() : HeaderField("Privacy", true)
        {
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFProxyAuthorization : public HFBase_5_
    {
        HFProxyAuthorization();
    };


    struct HFPOSPAuthToken : public HeaderField
    {
        HFPOSPAuthToken() : HeaderField("P-OSP-Auth-Token")
        {
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFPAssertedIdentity : public HeaderField
    {
        HFPAssertedIdentity() : HeaderField("P-Asserted-Identity")
        {
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFPPreferredIdentity : public HeaderField
    {
        HFPPreferredIdentity() : HeaderField("P-Preferred-Identity")
        {
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFMaxForwards : public HFBase_2_
    {
        HFMaxForwards() : HFBase_2_("Max-Forwards", true)
        {
        }

        bool is_zero_forward()
        {
            return digit_value_ == "0";
        }

        std::string max_forwards()
        {
            return digit_value_;
        }

        HFMaxForwards& max_forwards(std::string val)
        {
            digit_value_ = val;
            return *this;
        }
    };

    struct HFReason : public HeaderField
    {
        HFReason() : HeaderField("Reason", true)
        {
//          header_params_.append("cause");
//          header_params_.append("text");
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFReferTo : public HeaderField
    {
        HFReferTo() : HeaderField("Refer-To", "r")
        {
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    /* Referred-By: <sip:user@host.com>
     */
    struct HFReferredBy : public HeaderField
    {
        HFReferredBy() : HeaderField("Referred-By", "b")
        {
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFReplyTo : public HeaderField
    {
        HFReplyTo() : HeaderField("Replay-To")
        {
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFReplaces : public HeaderField
    {
        HFReplaces() : HeaderField("Replaces")
        {
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFRejectContact : public HeaderField
    {
        HFRejectContact() : HeaderField("Reject-Contact", "j")
        {
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct HFRequestDisposition : public HeaderField
    {
        HFRequestDisposition() : HeaderField("Request-Disposition")
        {
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };
```

```
/* Require: 100rel
 */
struct HFRequire : public HFBase_3_
{
    HFRequire() : HFBase_3_("Require", true)
    {
    }
};

struct HFProxyRequire : public HFBase_3_
{
    HFProxyRequire() : HFBase_3_("Proxy-Require", true)
    {
    }

    std::vector<std::string> misunderstand_tags()
    {
        // TODO: check for tags the element unable to understand
        return std::vector<std::string>();
    }

};

struct HFRoute : public HFBase_1_
{
    HFRoute() : HFBase_1_("Route", true)
    {
    }
};

struct HFRack : public HeaderField
{
    HFRack() : HeaderField("RACK")
    {
    }
    void generate_values();
    int parse(std::string &msg, size_t &pos);
};

struct HFSessionExpires : public HeaderField
{
    HFSessionExpires() : HeaderField("Session-Expires", "x")
    {
    }
    void generate_values();
    int parse(std::string &msg, size_t &pos);
};

struct HFSubscriptionState : public HeaderField
{
    HFSubscriptionState();

    void generate_values();
    int parse(std::string &msg, size_t &pos);
};

// -------------------- Response header -----------------------------
struct HFAuthenticationInfo : public HeaderField
{
    HFAuthenticationInfo();

    void generate_values();
    int parse(std::string &msg, size_t &pos);
};

struct HFErrorInfo : public HFBase_1_
{
    HFErrorInfo() : HFBase_1_("Error-Info", true)
    {
    }
};


struct HFMinSE : public HeaderField
{
    HFMinSE() : HeaderField("Min-SE")
    {
    }
    void generate_values();
    int parse(std::string &msg, size_t &pos);
};

struct HFProxyAuthenticate : public HFBase_4_
{
    HFProxyAuthenticate();
};

struct HFServer : public HFBase_3_
{
    HFServer() : HFBase_3_("Server")
    {
        sym_ = ' ';
    }
};

/* Unsupported: 100rel
 */
struct HFUnsupported : public HFBase_3_
{
    HFUnsupported() : HFBase_3_("Unsupported")
    {
        sym_ = ' ';
    }
};

struct HFWarning : public HeaderField
{
    struct WarningValue
    {
        std::string code_;
        std::string agent_;
        std::string text_;

        friend std::ostream& operator<< (std::ostream &o, WarningValue &w)
        {
            if (w.code_.size())
                o << w.code_ << ' ';

            if (w.agent_.size())
                o << w.agent_ << ' ';

            if (w.text_.size())
                o << " \"" << w.text_ << "";

            return o;
        }
    };

    std::vector<WarningValue> warn_vals_;

    HFWarning() : HeaderField("Warning")
    {
    }
    void generate_values();
    int parse(std::string &msg, size_t &pos);
```

```
};

struct HFWWWAuthenticate : public HFBase_5_
{
   HFWWWAuthenticate();
};

struct HFRSeq : public HeaderField
{
   HFRSeq() : HeaderField("RSeq")
   {
   }
   void generate_values();
   int parse(std::string &msg, size_t &pos);
};

struct HFAllow : public HFBase_3_
{
   HFAllow() : HFBase_3_("Allow")
   {
   }
};

struct HFContentEncoding : public HFBase_3_
{
   HFContentEncoding() : HFBase_3_("Content-Encoding", "e")
   {
   }
};

struct HFContentLength : public HFBase_2_
{
   HFContentLength() : HFBase_2_("Content-Length", "l", true)
   {
   }

   std::string length()
   {
      return digit_value_;
   }

   void length(std::string val)
   {
      digit_value_ = val;
   }

   void length(size_t val)
   {
      std::ostringstream o;
      o << val;
      digit_value_ = o.str();
   }
};

struct HFContentLanguage : public HeaderField
{
   HFContentLanguage() : HeaderField("Content-Language")
   {
   }
   void generate_values();
   int parse(std::string &msg, size_t &pos);
};

struct HFContentType : public HFBase_3_
{
   HFContentType() : HFBase_3_("Content-Type", "c")
   {
   }
```

```
    std::string type()
    {
        size_t ret;

        if (opts_.empty() || (ret = opts_.at(0).find_first_of("/") == std::string::npos))
            return std::string();

        return opts_.at(0).substr(0, ret);
    }

    std::string subtype()
    {
        size_t ret;

        if (opts_.empty()
        || (ret = opts_.at(0).find_first_of("/") == std::string::npos)
        || ret >= opts_.at(0).size())
            return std::string();

        return opts_.at(0).substr(ret+1);
    }

    HFContentType& type(std::string val)
    {
        opts_.push_back(val);
        return *this;
    }

    HFContentType& subtype(std::string val)
    {
        if (opts_.empty())
            return *this;

        opts_.at(0) += "/";
        opts_.at(0) += val;

        return *this;
    }
};

struct HFContentDisposition : public HFBase_3_
{
    HFContentDisposition() : HFBase_3_("Content-Disposition")
    {
    }
};

struct HFMinExpires : public HFBase_2_
{
    HFMinExpires() : HFBase_2_("Min-Expires")
    {
    }
};

struct HFExpires : public HFBase_2_
{
    HFExpires() : HFBase_2_("Expires")
    {
    }

    std::string expire()
    {
        return digit_value_;
    }

    void expire(std::string val)
    {
            digit_value_ = val;
        }
    };

    struct HFMIMEVersion : public HeaderField
    {
        std::string dotted_value_;

        HFMIMEVersion() : HeaderField("MIME-Version")
        {
        }
        void generate_values();
        int parse(std::string &msg, size_t &pos);
    };

    struct RequestLine
    {
        MethodMap method_;
        std::string request_uri_;
        std::string version_;

        RequestLine()
        {
        //    version_ = SIP_VERSION_2_0;
        }

        std::string operator() ()
        {
            std::ostringstream ret;

            ret << method_.name() << " " << request_uri_ << " " << version_ << "\n";
            return ret.str();
        }

        friend std::ostream& operator<< (std::ostream &o, RequestLine req)
        {
            return o << req.method_.name() << " " << req.request_uri_ << " " << req.version_ << "\r\n";
        }

        int parse(std::string &msg, size_t &pos);
    };

    struct ResponseStatus
    {
        std::string version_;
        RespCode resp_code_; // status_code_, reason_parase_

        ResponseStatus()
        {
//              version_ = SIP_VERSION_2_0;
//              resp_code_ = SIP_RESPONSE_SUCCESSFUL;
        }

        std::string operator() ()
        {
            std::ostringstream ret;

            ret << version_ << " " << resp_code_ << "\n";
            return ret.str();
        }

        friend std::ostream& operator<< (std::ostream &o, ResponseStatus res)
        {
            return o << res.version_ << " " << res.resp_code_ << "\r\n";
        }

        int parse(std::string &msg, size_t &pos);
    ResponseStatus& operator=(ResponseStatus resp)
    {
        version_ = resp.version_;
        resp_code_ = resp.resp_code_;
        return *this;
    }
};

#define out_if_not_null(o, it)     \
{                                  \
    if (it) o << *it;              \
}

#define out_if_not_empty(o, hf)    \
{                                  \
    for (auto &it : hf) o << *it;    \
}

typedef std::map<std::string, size_t> T_HF_MAP;

struct HeaderFields
{
    std::shared_ptr<RequestLine> req_line_;
    std::shared_ptr<ResponseStatus> resp_status_;
    // mandatory
    PtsOf<HFCallId> call_id_;
    PtsOf<HFCSeq> cseq_;
    PtsOf<HFFrom> from_;
    PtsOf<HFTo> to_;
    PtsOf<HFVia> via_;
    // Optional
    PtsOf<HFAlertInfo> alert_info_;
    PtsOf<HFAllowEvents> allow_events_;
    PtsOf<HFDate> date_;
    PtsOf<HFContact> contact_;
    PtsOf<HFOrganization> organization_;
    PtsOf<HFRecordRoute> record_route_;
    PtsOf<HFRetryAfter> retry_after_; // in second
    PtsOf<HFSubject> subject_;
    PtsOf<HFSupported> supported_;
    PtsOf<HFTimestamp> timestamp_;
    PtsOf<HFUserAgent> user_agent_;
    PtsOf<HFAnswerMode> answer_mode_;
    PtsOf<HFPrivAnswerMode> priv_answer_mode_;
    // request header fields
    PtsOf<HFAccept> accept_; // type/sub-type
    PtsOf<HFAcceptContact> accept_contact_;
    PtsOf<HFAcceptEncoding> accept_encoding_;
    PtsOf<HFAcceptLanguage> accept_language_;
    PtsOf<HFAuthorization> authorization_;
    PtsOf<HFCallInfo> call_info_;
    PtsOf<HFEvent> event_;
    PtsOf<HFInReplyTo> in_replay_to_;
    PtsOf<HFJoin> join_;
    PtsOf<HFPriority> priority_;
    PtsOf<HFPrivacy> privacy_;
    PtsOf<HFProxyAuthorization> proxy_authorization_;
    PtsOf<HFProxyRequire> proxy_require_;
    PtsOf<HFPOSPAuthToken> p_osp_auth_token_;
    PtsOf<HFPAssertedIdentity> p_asserted_identity_;
    PtsOf<HFPPreferredIdentity> p_preferred_identity_;
    PtsOf<HFMaxForwards> max_forwards_;
    PtsOf<HFReason> reason_;
    PtsOf<HFReferTo> refer_to_;
    PtsOf<HFReferredBy> referred_by_;
    PtsOf<HFReplyTo> reply_to_;
    PtsOf<HFReplaces> replaces_;
    PtsOf<HFRejectContact> reject_contact_;
```

```
        PtsOf<HFRequestDisposition> request_disposition_;
        PtsOf<HFRequire> require_;
        PtsOf<HFRoute> route_;
        PtsOf<HFRack> rack_;
        PtsOf<HFSessionExpires> session_expires_; // in second
        PtsOf<HFSubscriptionState> subscription_state_;
        // response header fields
        PtsOf<HFAuthenticationInfo> authentication_info_;
        PtsOf<HFErrorInfo> error_info_;
        PtsOf<HFMinExpires> min_expires_;
        PtsOf<HFMinSE> min_se_;
        PtsOf<HFProxyAuthenticate> proxy_authenticate_;
        PtsOf<HFServer> server_;
        PtsOf<HFUnsupported> unsupported_;
        PtsOf<HFWarning> warning_;
        PtsOf<HFWWWAuthenticate> www_authenticate_;
        PtsOf<HFRSeq> rseq_;
        // message header fields
        PtsOf<HFAllow> allow_;
        PtsOf<HFContentEncoding> content_encoding_;
        PtsOf<HFContentLength> content_length_;
        PtsOf<HFContentDisposition> content_disposition_;
        PtsOf<HFContentLanguage> content_language_;
        PtsOf<HFContentType> content_type_;
        PtsOf<HFExpires> expires_; // in second
        PtsOf<HFMIMEVersion> mime_version_;

        HeaderFields();

        ~HeaderFields();

    public:
        static T_HF_MAP allowed_fields_;
        static void init_allowed_fields();


    };

} // namespace EasySip
```

```
/*
 * include/parameter.h
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#pragma once

#include "utils.h"
#include "except.h"

namespace EasySip
{
 class Parameter : public std::pair<std::string, std::string>
 {
 public:
  Parameter()
  {
  }

  Parameter(std::string name, std::string value = "")
  {
   first = name;
   second = value;
  }

  ~Parameter()
  {
  }

  std::string name() const
  {
   return first;
  }

  std::string value() const
  {
   return second;
  }

  void name(const std::string n)
  {
   first = n;
  }

  void value(const std::string v)
  {
   second = v;
  }

  friend bool operator< (Parameter a, Parameter b)
  {
   return a.name() < b.name();
  }

  friend std::ostream& operator<< (std::ostream &o, Parameter p)
  {
   o << p.name();

   if (p.value().size())
   {
    o << "=" << p.value();
   }

   return o;
  }
 };

 class Parameters : public std::vector<Parameter>
```

```
{
 std::string sym_;

public:

 Parameters(std::string sym) : sym_(sym)
 {
 }

 Parameters() : sym_(";")
 {
 }

 ~Parameters()
 {
 }

 void Sym(std::string sym)
 {
  sym_ = sym;
 }

 std::string Sym() const
 {
  return sym_;
 }

 void append(std::string name, std::string value)
 {
  if (name.empty()) return;

  if (!has_name(name))
   push_back(Parameter(name, value));
 }

 void append(std::string name)
 {
  append(name, "");
 }

 bool has_name(std::string name)
 {
  for (auto &it : *this)
  {
   if (name == it.first)
    return true;
  }

  return false;
 }

 void set_value_by_name(std::string name, std::string value)
 {
  for (auto &it : *this)
  {
   if (name == it.first)
   {
    it.second = value;
    return;
   }
  }

  append(name, value);
 }

 std::string get_value_by_name(std::string name)
 {
  for (auto it : *this)
```

```
   {
    if (it.first == name)
     return it.second;
   }

   return std::string();
  }

  friend std::ostream& operator<< (std::ostream &o, Parameters &ps)
  {
   for (Parameters::iterator it = ps.begin(); it != ps.end(); it++)
   {
    if (std::distance(ps.begin(), it) > 0 && std::distance(ps.begin(), it) < (int)ps.size())
     o << ps.Sym();

    o << *it;
   }

   return o;
  }
 };

 struct ItemWithParams
 {
  std::vector<std::string> items_;
  Parameters params_;

  ItemWithParams()
  {
  }

  ItemWithParams(std::string item)
  {
   items_.push_back(item);
  }

  Parameters& params()
  {
   return params_;
  }

  friend std::ostream& operator<< (std::ostream &o, ItemWithParams &c)
  {
   for (auto &it : c.items_)
    o << it;

   if (c.params_.size())
    o << ";" << c.params_;

   return o;
  }

  void set_param(std::string name, std::string value)
  {
   params_.set_value_by_name(name, value);
  }

  void add_param(std::string name, std::string value = "")
  {
   params_.append(name, value);
  }

  bool has_param(std::string name)
  {
   return params_.has_name(name);
  }
 };
} // namespace EasiSip
```

```
/*
 * include/uri.h
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#pragma once

#include <memory>
#include <locale>
#include "parameter.h"

namespace EasySip
{
    class Contact : public ItemWithParams
    {
    public:
        Contact()
        {
            items_.resize(2);
        }

        Contact(std::string name, std::string uri)
        {
            items_.push_back(name);
            items_.push_back(uri);
        }

        std::string& name()
        {
            return items_.at(0);
        }

        std::string& uri()
        {
            return items_.at(1);
        }

        void name(std::string name)
        {
            items_.at(0) = name;
        }

        void uri(std::string uri)
        {
            items_.at(1) = uri;
        }

        friend std::ostream& operator<< (std::ostream &o, Contact &c)
        {
            if (c.name().size())// || c.params().size())
                o << c.name();// << " <";

            o << " <" << c.uri();

            if (c.params().size())
                o << ";" << c.params();

//          if (c.name().size() || c.params().size())
                o << ">";

            return o;
        }

        bool empty()
        {
            return (name().empty() && uri().empty());
        }
        bool full()
        {
            return (!uri().empty() && !name().empty());
        }
    };

    struct ContactList : public PtsOf<Contact>
    {
        void cleanup_empty_uri()
        {
            for (iterator it = begin(); it != end();)
            {
                if ((*it)->uri().empty())
                    erase(it);
                else
                    it++;
            }
        }

        void append(std::string uri, std::string name = "")
        {
            if (uri.empty()) return;

            append_item();
            last()->uri(uri);

            if (name.size())
                last()->name(name);
        }

        void append(ContactList& c)
        {
            insert(end(), c.begin(), c.end());
        }

        void append(ContactList::iterator from, ContactList::iterator to)
        {
            insert(end(), from, to);
        }

    };
//          uri_params_.append("aai");
//          uri_params_.append("bnc");
//          uri_params_.append("cause");
//          uri_params_.append("ccxml");
//          uri_params_.append("comp");
//          uri_params_.append("gr");
//          uri_params_.append("locale");
//          uri_params_.append("lr", false);
//          uri_params_.append("m");
//          uri_params_.append("maddr");
//          uri_params_.append("maxage");
//          uri_params_.append("maxstale");
//          uri_params_.append("method");
//          uri_params_.append("ob");
//          uri_params_.append("postbody");
//          uri_params_.append("repeat");
//          uri_params_.append("sg");
//          uri_params_.append("sigcomp-id");
//          uri_params_.append("target");
//          uri_params_.append("transport");
//          uri_params_.append("ttl");
//          uri_params_.append("user");
//          // RFC-4240
//          uri_params_.append("content-type");
//          uri_params_.append("delay");
//          uri_params_.append("duration");
//          uri_params_.append("extension");
```

```
//		uri_params_.append("param");
//		uri_params_.append("play");
//		uri_params_.append("voicexml");


} // namespace EasySip
```

```
/*
 * include/thread.h
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#pragma once

#include <pthread.h>
#include "except.h"

namespace EasySip
{
//#ifndef _GNU_SOURCE
//#define _GNU_SOURCE
//#endif

    class ThrAttr
    {
    public:
        struct Stack
        {
            void *stackaddr_;
            size_t stacksize_;
        };

        struct SchedParam
        {
            int policy_;
            struct sched_param param_;
        };

    private:
        pthread_attr_t attr_;
        cpu_set_t cpuset_;
        Stack stack_;
        SchedParam schedparam_;

    public:

        ThrAttr()
        {
            if (0 > pthread_attr_init(&attr_))
                std::cerr << "pthread_attr_init: " << strerror(errno) << '\n';
        }

        ~ThrAttr()
        {
            if (0 > pthread_attr_destroy(&attr_))
                std::cerr << "pthread_attr_destroy: " << strerror(errno) << '\n';
        }

        pthread_attr_t& Attr()
        {
            return attr_;
        }

        cpu_set_t& affinity_np()
        {
            if (0 > pthread_attr_getaffinity_np(&attr_, sizeof(cpu_set_t), &cpuset_))
                std::cerr << "pthread_attr_getaffinity_np: " << strerror(errno) << '\n';

            return cpuset_;
        }

        ThrAttr& affinity_np(cpu_set_t cpuset)
        {
            cpuset_ = cpuset;
            if (0 > pthread_attr_setaffinity_np(&attr_, sizeof(cpu_set_t), &cpuset_))
                std::cerr << "pthread_attr_setaffinity_np: " << strerror(errno) << '\n';

            return *this;
        }

        int detachstate()
        {
            int detachstate;

            if (0 > pthread_attr_getdetachstate(&attr_, &detachstate))
                std::cerr << "pthread_attr_getdetachstate: " << strerror(errno) << '\n';

            return detachstate;
        }

        ThrAttr& detachstate(int detachstate)
        {
            if (0 > pthread_attr_setdetachstate(&attr_, detachstate))
                std::cerr << "pthread_attr_setdetachstate: " << strerror(errno) << '\n';

            return *this;
        }

        size_t guardsize()
        {
            size_t guardsize;

            if (0 > pthread_attr_getguardsize(&attr_, &guardsize))
                std::cerr << "pthread_attr_getguardsize: " << strerror(errno) << '\n';

            return guardsize;
        }

        ThrAttr& guardsize(size_t guardsize)
        {
            if (0 > pthread_attr_setguardsize(&attr_, guardsize))
                std::cerr << "pthread_attr_setguardsize: " << strerror(errno) << '\n';

            return *this;
        }

        int inheritsched()
        {
            int inheritsched;

            if (0 > pthread_attr_getinheritsched(&attr_, &inheritsched))
                std::cerr << "pthread_attr_getinheritsched: " << strerror(errno) << '\n';

            return inheritsched;
        }

        ThrAttr& inheritsched(int inheritsched)
        {
            if (0 > pthread_attr_setinheritsched(&attr_, inheritsched))
                std::cerr << "pthread_attr_setinheritsched: " << strerror(errno) << '\n';

            return *this;
        }

        SchedParam& schedparam()
        {
            if (0 > pthread_attr_getschedparam(&attr_, &schedparam_.param_))
                std::cerr << "pthread_attr_getschedparam: " << strerror(errno) << '\n';

            return schedparam_;
        }
```

```
        ThrAttr& schedparam(int priority)
        {
//              schedparam_.policy_ = policy;
            schedparam_.param_.sched_priority = priority;

            if (0 > pthread_attr_setschedparam(&attr_, &schedparam_.param_))
                std::cerr << "pthread_attr_setschedparam: " << strerror(errno) << '\n';

            return *this;
        }

        int schedpolicy()
        {
            int schedpolicy;

            if (0 > pthread_attr_getschedpolicy(&attr_, &schedpolicy))
                std::cerr << "pthread_attr_getschedpolicy: " << strerror(errno) << '\n';

            return schedpolicy;
        }

        ThrAttr& schedpolicy(int schedpolicy)
        {
            if (0 > pthread_attr_setschedpolicy(&attr_, schedpolicy))
                std::cerr << "pthread_attr_setschedpolicy: " << strerror(errno) << '\n';

            return *this;
        }

        int scope()
        {
            int scope;

            if (0 > pthread_attr_getscope(&attr_, &scope))
                std::cerr << "pthread_attr_getscope: " << strerror(errno) << '\n';

            return scope;
        }

        ThrAttr& scope(int scope)
        {
            if (0 > pthread_attr_setscope(&attr_, scope))
                std::cerr << "pthread_attr_setscope: " << strerror(errno) << '\n';

            return *this;
        }

        Stack& stack()
        {
            if (0 > pthread_attr_getstack(&attr_, &stack_.stackaddr_, &stack_.stacksize_))
                std::cerr << "pthread_attr_getstack: " << strerror(errno) << '\n';

            return stack_;
        }

        ThrAttr& stack(void *stackaddr, size_t stacksize)
        {
            stack_.stackaddr_ = &stackaddr;
            stack_.stacksize_ = stacksize;

            if (0 > pthread_attr_setstack(&attr_, stack_.stackaddr_, stack_.stacksize_))
                std::cerr << "pthread_attr_setstack: " << strerror(errno) << '\n';

            return *this;
        }
// ------------------------------------deprecated !----------------------------------------
//          void* stackaddr()
//          {
```

```
//          if (0 > pthread_attr_getstackaddr(&attr_, &stack_.stackaddr_))
//              std::cerr << "pthread_attr_getstackaddr: " << strerror(errno) << '\n';
//
//          return stack_.stackaddr_;
//      }
//
//      ThrAttr& stackaddr(void *stackaddr)
//      {
//          stack_.stackaddr_ = &stackaddr;
//
//          if (0 > pthread_attr_setstackaddr(&attr_, stack_.stackaddr_))
//              std::cerr << "pthread_attr_setstackaddr: " << strerror(errno) << '\n';
//
//          return *this;
//      }
// ++++++++++++++++++++++++++++++++++++deprecated !++++++++++++++++++++++++++++++++++++++++

        size_t stacksize()
        {
            if (0 > pthread_attr_getstacksize(&attr_, &stack_.stacksize_))
                std::cerr << "pthread_attr_getstacksize: " << strerror(errno) << '\n';

            return stack_.stacksize_;
        }

        ThrAttr& stacksize(size_t stacksize)
        {
            stack_.stacksize_ = stacksize;

            if (0 > pthread_attr_setstacksize(&attr_, stack_.stacksize_))
                std::cerr << "pthread_attr_setstacksize: " << strerror(errno) << '\n';

            return *this;
        }
    };

// Xdestroy            Xsetdetachstate     pthread_cleanup_push_defer_np   pthread_self
// Xgetaffinity_np     Xsetguardsize       pthread_create                  pthread_setaffinity_np
// Xgetdetachstate     Xsetinheritsched    pthread_detach                  pthread_setcancelstate
// Xgetguardsize       Xsetschedparam      pthread_equal                   pthread_setcanceltype
// Xgetinheritsched    Xsetschedpolicy     pthread_exit                    pthread_setconcurrency
// Xgetschedparam      Xsetscope           pthread_getaffinity_np          pthread_setschedparam
// Xgetschedpolicy     Xsetstack           pthread_getattr_np              pthread_setschedprio
// Xgetscope           Xsetstackaddr       pthread_getconcurrency          pthread_sigmask
// Xgetstack           Xsetstacksize       pthread_getcpuclockid           pthread_sigqueue
// Xgetstackaddr       pthread_cancel                  pthread_getschedparam           pthread_testcancel
// Xgetstacksize       pthread_cleanup_pop             pthread_join                    pthread_timedjoin_np
// Xinit               pthread_cleanup_pop_restore_np  pthread_kill                    pthread_tryjoin_np
// Xsetaffinity_np     pthread_cleanup_push            pthread_kill_other_threads_np   pthread_yield

    class ThrCondAttr
    {
    protected:
        pthread_condattr_t cattr_;

    public:
        ThrCondAttr()
        {
            if (0 > pthread_condattr_init(&cattr_))
                std::cerr << "pthread_condattr_init: " << strerror(errno) << '\n';
        }

        ~ThrCondAttr()
        {
            if (0 > pthread_condattr_destroy(&cattr_))
                std::cerr << "pthread_condattr_destroy: " << strerror(errno) << '\n';
        }
        pthread_condattr_t& Attr()
        {
            return cattr_;
        }
    };

    class Mutex
    {
        pthread_mutex_t mutex_;
    };

    class ThrCond
    {
        ThrCondAttr attr_;
        pthread_cond_t cond_;
        Mutex mutex_;

        ThrCond()
        {

        }

        ~ThrCond()
        {
        }
    };


    class Thread
    {
    protected:

        pthread_t id_;
        ThrAttr attr_;
        void* (*loop_) (void*);
        void *arg_;

    public:

  Thread() : loop_(0), arg_(0)
  {
  }

        Thread(void* (*loop) (void*), void* arg = 0)
        : loop_(loop), arg_(arg)
        {
            if (0 > pthread_create(&id_, &attr_.Attr(), loop_, arg_))
              std::cerr << "pthread_create: " << strerror(errno) << '\n';
        }

        ~Thread()
        {
        }

        pthread_t id()
        {
            return id_;
        }

//        Thread& add_cleanup(void (*routine)(void *), void *arg, int n)
//        {
//            pthread_cleanup_push(routine, arg)
//            pthread_cleanup_pop(n)
//            return *this;
//        }

        Thread& schedprio(int prio)
        {
            pthread_setschedprio(id_, prio);
            return *this;
        }

        Thread& concurrency(int c)
        {
            if (0 > pthread_setconcurrency(c))
                std::cerr << "pthread_setconcurrency: " << strerror(errno) << '\n';
            return *this;
        }

        int concurrency()
        {
            return pthread_getconcurrency();
        }

        friend bool operator== (Thread &a, Thread &b)
        {
            return pthread_equal(a.id(), b.id());
        }

        Thread& cancel()
        {
            if (0 > pthread_cancel(id_))
                std::cerr << "pthread_cancel: " << strerror(errno) << '\n';
            return *this;
        }

        int join()
        {
            void *ret;

            if (0 > pthread_join(id_, &ret))
                std::cerr << "pthread_join: " << strerror(errno) << '\n';
            std::cout << (char*)ret << '\n';
            return 0;
        }
    };

    #define Thread(f, a) Thread(reinterpret_cast<void* (*) (void*)>(f), (void*)a)

} // namespace EasySip
```

```
/*
 * include/dialog.h
 */
#include "message.h"
#include <algorithm>

namespace EasySip
{
    class DialogId
    {
        HFCallId call_id_;
        std::string local_tag_;
        std::string remote_tag_;

    public:
        DialogId(HFCallId& id, std::string ltag, std::string rtag)
        : call_id_(id), local_tag_(ltag), remote_tag_(rtag)
        {
        }

        DialogId()
        {
        }

        HFCallId& call_id()
        {
            return call_id_;
        }

        std::string& local_tag()
        {
            return local_tag_;
        }

        std::string& remote_tag()
        {
            return remote_tag_;
        }

        void call_id(HFCallId val)
        {
            call_id_ = val;
        }

        void local_tag(std::string val)
        {
            local_tag_ = val;
        }

        void remote_tag(std::string val)
        {
            remote_tag_ = val;
        }

        friend bool operator== (DialogId a, DialogId b)
        {
            if (a.call_id() == b.call_id())
                if (a.local_tag() == b.local_tag())
                    if (a.remote_tag() == b.remote_tag())
                        return true;

            return false;
        }

        friend std::ostream& operator<< (std::ostream &o, DialogId id)
        {
            return o << id.call_id()
                << "local_tag: " << id.local_tag() << '\n'
                << "remote_tag: " << id.remote_tag() << '\n';
        }
    };

    class Dialog
    {
        DialogId id_;
        HFCSeq local_seq_;
        HFCSeq remote_seq_;
        std::string local_uri_;
        std::string remote_uri_;
        ContactList remote_target_;
        bool secure_flag_;
        PtsOf<HFRecordRoute> routes_;

        bool confirmed_;
        bool still_ringing_;

    public:

        Dialog()
        : secure_flag_(false), confirmed_(false),
         still_ringing_(false)
        {
        }

        Dialog(Dialog &dia);
        Dialog(ResponseMessage &in_msg);
        Dialog(RequestMessage &in_msg);

        DialogId& id()
        {
            return id_;
        }

        HFCSeq& local_seq()
        {
            return local_seq_;
        }

        HFCSeq& remote_seq()
        {
            return remote_seq_;
        }

        std::string& local_uri()
        {
            return local_uri_;
        }

        std::string& remote_uri()
        {
            return remote_uri_;
        }

        ContactList& remote_target()
        {
            return remote_target_;
        }

        bool& secure_flag()
        {
            return secure_flag_;
        }

        PtsOf<HFRecordRoute>& routes()
        {
            return routes_;
        }

        void id(DialogId val)
        {
            id_ = val;
        }

        void local_seq(HFCSeq val)
        {
            local_seq_ = val;
        }

        void remote_seq(HFCSeq val)
        {
            remote_seq_ = val;
        }

        void local_uri(std::string val)
        {
            local_uri_ = val;
        }

        void remote_uri(std::string val)
        {
            remote_uri_ = val;
        }

        void remote_target(ContactList val)
        {
            remote_target_ = val;
        }

        void secure_flag(bool val)
        {
            secure_flag_ = val;
        }

        void routes(PtsOf<HFRecordRoute> val)
        {
            routes_ = val;
        }

        void is_confirmed(bool c)
        {
            confirmed_ = c;
        }

        bool is_confirmed()
        {
            return confirmed_;
        }

        bool still_ringing()
        {
            return still_ringing_;
        }

        void still_ringing(bool ring)
        {
            still_ringing_ = ring;
        }

        friend std::ostream& operator<< (std::ostream &o, Dialog &dia);
    };

    class Dialogs : public PtsOf<Dialog>
    {
    public:
        Dialog* create_dialog();
        Dialog* create_dialog(Dialog &dialog);

        void cancel_dialog(DialogId val);

        Dialog* get_dialog_by_id(DialogId &val);

        Dialog* operator[] (DialogId val);
    };
} // namespace EasySip
```

```
/*
 * include/transaction.h
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#pragma once

#include "utils.h"
#include "except.h"
#include "timer.h"

namespace EasySip
{
    enum
    {
        T_FSM_CALLING,
        T_FSM_TRYING,
        T_FSM_PROCEEDING,
        T_FSM_COMPLETED,
        T_FSM_CONFIRMED,
        T_FSM_TERMINATED,
    };

    class Fsm// : public Thread
    {
    protected:

        int state_;
        bool run_;
  Thread t_;

    public:
        Fsm(int s, bool r = true) : state_(s), run_(r)
        {
            setup();
            t_ = Thread(&Fsm::main_loop, this);
        }

        ~Fsm()
        {
   t_.join();
        }

        void run(bool r)
        {
            run_ = r;
        }

        bool run()
        {
            return run_;
        }

        int state()
        {
            return state_;
        }

        void state(int s)
        {
            state_ = s;
        }

        Fsm& setup()
        {
            return *this;
        }
        int loop()
        {
            return (run_ = false);
        }

        int main_loop()
        {
            while (run_)
            {
                if (loop() == PROCEDURE_ERROR)
                    return PROCEDURE_ERROR;
            }

            return PROCEDURE_OK;
        }
    };

    class Transaction : public Fsm
    {
    public:

        Transaction() : Fsm(T_FSM_CALLING)
        {
        }

    protected:

        virtual int loop();
    };

}; // namespace EasySip
```

```
/*
 * include/socket.h
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#pragma once


#include <memory>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <ifaddrs.h>
#include <error.h>
#include <string.h>

#include "except.h"

namespace EasySip
{
    /*
     * domain:
     * AF_UNIX, AF_LOCAL    Local communication              unix(7)
     * AF_INET              IPv4 Internet protocols          ip(7)
     * AF_INET6             IPv6 Internet protocols          ipv6(7)
     * AF_IPX               IPX - Novell protocols
     * AF_NETLINK           Kernel user interface device     netlink(7)
     * AF_X25               ITU-T X.25 / ISO-8208 protocol   x25(7)
     * AF_AX25              Amateur radio AX.25 protocol
     * AF_ATMPVC            Access to raw ATM PVCs
     * AF_APPLETALK         Appletalk                        ddp(7)
     * AF_PACKET            Low level packet interface       packet(7)
     *
     * type:
     * SOCK_STREAM
     * SOCK_DGRAM
     * SOCK_SEQPACKET
     * SOCK_RAW
     * SOCK_RDM
     * SOCK_PACKET
     * SOCK_NONBLOCK
     * SOCK_CLOEXEC
    */
    class Socket
    {
    protected:

        int sk_;
        int domain_;
        int type_;
        int proto_;

    public:

        static std::string get_ip_addr();

        Socket()
        {
        }

        Socket(int domain, int type, int proto)
        : domain_(domain), type_(type), proto_(proto)
        {
            sk_ = socket(domain_, type_, proto_);

            // TODO: throw exception
            if (0 > sk_)
                std::cerr << "socket: " << strerror(errno) << '\n';
        }

        ~Socket()
        {
            if (0 < sk_)
                close(sk_);
        }

        int set_timeout(int sec);
    };

    class SocketIp4 : public Socket
    {
    protected:

        struct sockaddr_in sk_addr_;
        std::string addr_;

        struct sockaddr_in self_sk_addr_;
        std::string self_addr_;

        std::string msg_;
        int max_rx_;

    public:
        SocketIp4(int type, int proto = 0)
        : Socket(PF_INET, type, proto)
        {
            sk_addr_.sin_family = AF_INET;
            sk_addr_.sin_port = htons(0);
            sk_addr_.sin_addr.s_addr = htonl(INADDR_ANY);

            self_sk_addr_.sin_family = AF_INET;
            self_sk_addr_.sin_port = htons(0);
            self_sk_addr_.sin_addr.s_addr = htonl(INADDR_ANY);

            max_rx_ = 1024;

            set_timeout(3);
        }

        int Port()
        {
            return ntohs(sk_addr_.sin_port);
        }

        void Port(int port)
        {
            sk_addr_.sin_port = htons(port);
        }

        std::string& Addr()
        {
            return addr_;
        }

        void Addr(std::string addr)
        {
            addr_ = addr;
            inet_aton(addr_.c_str(), (in_addr*)&sk_addr_.sin_addr.s_addr);
        }

        int SelfPort()
        {
            return ntohs(self_sk_addr_.sin_port);
        }
        void SelfPort(int port)
        {
            self_sk_addr_.sin_port = htons(port);
        }

        std::string SelfAddr()
        {
            return self_addr_;
        }

        void SelfAddr(std::string addr)
        {
            self_addr_ = addr;
            inet_aton(self_addr_.c_str(), (in_addr*)&self_sk_addr_.sin_addr.s_addr);
        }

        std::string& Message()
        {
            return msg_;
        }

        void Message(std::string msg)
        {
            msg_ = msg;
        }

        void clear_msg()
        {
            msg_.clear();
        }

        int MaxRx()
        {
            return max_rx_;
        }

        void MaxRx(int max)
        {
            max_rx_ = max;
        }

        ~SocketIp4()
        {
        }

        friend std::ostream& operator<< (std::ostream &o, SocketIp4 sk)
        {
            return o << sk.SelfAddr() << ":" << sk.SelfPort() << '\n'
                << sk.Addr() << ":" << sk.Port();
        }
    };

    class SocketIp4UDP : public SocketIp4
    {
        bool binded_;
        bool need_bind_;
    public:
        SocketIp4UDP();
        SocketIp4UDP(std::string addr, int port);

        ~SocketIp4UDP();

        int setup_server();

        void send_buffer(const std::string msg);

        int recv_buffer(int selfloop = 1);
```

```
        void Bind(bool b)
        {
            binded_ = b;
        }

        void NeedBind(bool b)
        {
            need_bind_ = b;
        }

        bool Bind()
        {
            return binded_;
        }

        bool NeedBind()
        {
            return need_bind_;
        }
    };
} // namespace EasySip
```

```
/*
 * include/utils.h
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#pragma once

#include <sstream>
#include <set>
#include <map>
#include <vector>
#include <iostream>

namespace EasySip
{
    #define CASE_UPPER_ALPHA \
                case 'A': \
                case 'B': \
                case 'C': \
                case 'D': \
                case 'E': \
                case 'F': \
                case 'G': \
                case 'H': \
                case 'I': \
                case 'J': \
                case 'K': \
                case 'L': \
                case 'M': \
                case 'N': \
                case 'O': \
                case 'P': \
                case 'Q': \
                case 'R': \
                case 'S': \
                case 'T': \
                case 'U': \
                case 'V': \
                case 'W': \
                case 'X': \
                case 'Y': \
                case 'Z':
    #define CASE_LOWER_ALPHA \
                case 'a': \
                case 'b': \
                case 'c': \
                case 'd': \
                case 'e': \
                case 'f': \
                case 'g': \
                case 'h': \
                case 'i': \
                case 'j': \
                case 'k': \
                case 'l': \
                case 'm': \
                case 'n': \
                case 'o': \
                case 'p': \
                case 'q': \
                case 'r': \
                case 's': \
                case 't': \
                case 'u': \
                case 'v': \
                case 'w': \
                case 'x': \
                case 'y': \
                case 'z':

    #define CASE_ALPHA \
          CASE_UPPER_ALPHA \
          CASE_LOWER_ALPHA

    #define CASE_DIGIT \
                case '1': \
                case '2': \
                case '3': \
                case '4': \
                case '5': \
                case '6': \
                case '7': \
                case '8': \
                case '9': \
                case '0':

    #define CASE_ALPHA_NUM \
          CASE_ALPHA \
          CASE_DIGIT

    #define CASE_TOKEN \
                CASE_ALPHA_NUM \
                case '-': \
                case '.': \
                case '!': \
                case '%': \
                case '*': \
                case '_': \
                case '+': \
                case 39: \
                case '~':

    #define CASE_WORD \
                CASE_TOKEN \
                case '(': \
                case ')': \
                case '<': \
                case '>': \
                case ':': \
                case 92: \
                case 34: \
                case '/': \
                case '[': \
                case ']': \
                case '?': \
                case '{': \
                case '}':

    #define do_if_is_alpha(c, f)     \
    {                              \
       std::locale loc;            \
       if (std::isalpha(c, loc)) { f; }    \
    }

    enum
    {
       PROCEDURE_OK,        // everything's normal
       MESSAGE_PROCESSED,    // message issue, but handled
       PROCEDURE_ERROR,    // something wrong, unhandled
    };

    #define PROGRESS_WITH_FEEDBACK(opr, cond, p)\
    {                              \
       std::cout << opr;               \
       while (cond)                  \
       {                             \
        std::cout << " ..."; p;         \
    }                                   \
    std::cout << "\n";                  \
}

template<typename T>
T& RefOf(T& t) { return t; }

template<typename T>
class PtsOf : public std::vector<T*>
{
public:
    PtsOf()
    {
    }

    void append_item()
    {
        this->push_back(new T);
    }

    void append_item(T &it)
    {
        this->push_back(new T(it));
    }

    T* first()
    {
        return this->at(0);
    }

    T* last()
    {
        return this->at(this->size()-1);
    }

    friend std::ostream& operator<< (std::ostream &o, PtsOf<T> &pts)
    {
        for (auto &it : pts)
        {
            o << *it;
        }

        return o;
    }
};

class CodeMap : public std::pair<int, std::string>
{
public:

    void Code(int c)
    {
        first = c;
    }

    void name(std::string n)
    {
        second = n;
    }

public:

    int code() const
    {
        return first;
    }
```

```
        std::string name() const
        {
            return second;
        }

        CodeMap()
        {
        }

        CodeMap(int c)
        {
            first = c;
        }

        CodeMap(std::string n)
        {
            second = n;
        }

        CodeMap(int c, std::string n)
        {
            first = c;
            second = n;
        }

        std::string CodeStr() const
        {
            std::ostringstream o;
            o << first;
            return o.str();
        }

        bool operator< (CodeMap cm)
        {
            return (first < cm.first);
        }

        bool operator== (const CodeMap &cm)
        {
            return ((first == cm.first ) && (second == cm.second));
        }

        friend std::ostream& operator<< (std::ostream &o, CodeMap cm)
        {
            o << cm.first << " " << cm.second;
            return o;
        }

        std::string operator() ()
        {
            std::ostringstream o;
            o << first << " " << second << '\n';
            return o.str();
        }

        void operator() (CodeMap &cm)
        {
            *this = cm;
        }
    };

} // namespace EasiSip
```

```
/*
 * include/request_message.h
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#pragma once

#include "parameter.h"

namespace EasySip
{
    typedef CodeMap MethodMap;

    enum
    {
        METHOD_ID_INVITE,
        METHOD_ID_CANCEL,
        METHOD_ID_ACK,
        METHOD_ID_BYE,
        METHOD_ID_REGISTER,
        METHOD_ID_OPTIONS,
        METHOD_ID_SUBSCRIBE,
        METHOD_ID_NOTIFY,
        METHOD_ID_MESSAGE,
        METHOD_ID_INFO,
        METHOD_ID_UPDATE,
        METHOD_ID_REFER,
        METHOD_ID_PRACK,
    };

    // Requests since SIP 1.0
    // RFC-3261
    const MethodMap METHOD_INVITE(METHOD_ID_INVITE, "INVITE");
    const MethodMap METHOD_CANCEL(METHOD_ID_CANCEL, "CANCEL");
    const MethodMap METHOD_ACK(METHOD_ID_ACK, "ACK");
    const MethodMap METHOD_BYE(METHOD_ID_BYE, "BYE");
    const MethodMap METHOD_REGISTER(METHOD_ID_REGISTER, "REGISTER");
    const MethodMap METHOD_OPTIONS(METHOD_ID_OPTIONS, "OPTIONS");
    // Additional requests since SIP 2.0
    // RFC-6665
    const MethodMap METHOD_SUBSCRIBE(METHOD_ID_SUBSCRIBE, "SUBSCRIBE");
    const MethodMap METHOD_NOTIFY(METHOD_ID_NOTIFY, "NOTIFY");
    const MethodMap METHOD_MESSAGE(METHOD_ID_MESSAGE, "MESSAGE");
    // RFC-6086
    const MethodMap METHOD_INFO(METHOD_ID_INFO, "INFO");
    // RFC-3311
    const MethodMap METHOD_UPDATE(METHOD_ID_UPDATE, "UPDATE");
    // RFC-3515
    const MethodMap METHOD_REFER(METHOD_ID_REFER, "REFER");
    // RFC-3262
    const MethodMap METHOD_PRACK(METHOD_ID_PRACK, "PRACK");

    typedef std::set<MethodMap> MethodMapList;
} // namespace EasiSip
```

```
/*
 * include/except.h
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#pragma once

#include <string.h>
#include <stdexcept>
#include <iostream>

namespace EasySip
{
    class Except : std::exception
    {
    protected:

        std::string msg_;

    public:

        Except()
        {
        }

        Except(std::string msg)
        : msg_(msg)
        {
        }

        virtual const char* what();
    };
} // namespace EasySip
```

```
/*
 * include/Element/uaserver.h
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#pragma once

#include "Element/element.h"

namespace EasySip
{
 class UAServer : public Element
 {
 public:

  UAServer();

  ~UAServer()
  {
  }

 };

} // namespace EasySip
```

```
/*
 * include/Element/proxy.h
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#pragma once

#include "Element/element.h"

namespace EasySip
{
 class Proxy : public Element
 {
 public:

  Proxy();

  ~Proxy()
  {
  }

//  virtual int invite_request();
//  virtual int register_request();
//  virtual int bye_request();
//  virtual int cancel_request();
//  virtual int update_request();
//  virtual int info_request();
//  virtual int ack_request();
//  virtual int message_request();
//  virtual int subscribe_request();
//  virtual int notify_request();
//  virtual int refer_request();
//  virtual int options_request();
//  virtual int prack_request();
//
//  virtual int on_invite_request(RequestMessage &in_msg);
//  virtual int on_register_request(RequestMessage &in_msg);
//  virtual int on_bye_request(RequestMessage &in_msg);
//  virtual int on_ack_request(RequestMessage &in_msg);
//  virtual int on_cancel_request(RequestMessage &in_msg);
//  virtual int on_options_request(RequestMessage &in_msg);
//  virtual int on_refer_request(RequestMessage &in_msg);
//  virtual int on_subscribe_request(RequestMessage &in_msg);
//  virtual int on_notify_request(RequestMessage &in_msg);
//  virtual int on_message_request(RequestMessage &in_msg);
//  virtual int on_info_request(RequestMessage &in_msg);
//  virtual int on_prack_request(RequestMessage &in_msg);
//  virtual int on_update_request(RequestMessage &in_msg);
//  virtual int on_response(Message &in_msg);
 };

} // namespace EasySip
```

```
/*
 * include/Element/element.h
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#pragma once

#include "message.h"
#include "socket.h"
#include "dialog.h"
#include "transaction.h"
#include <queue>

namespace EasySip
{
    class Element
    {
    protected:

        MethodMapList allowed_methods_;
        RespCodeList allowed_responses_;
        SocketIp4UDP udp_;

        bool run_;
        Dialogs dialogs_;
        std::queue<std::string> msgq_;

        bool stateful_;

  Transaction ivt_;

    private:

        void init_allowed_methods();
        void init_allowed_responses();

    public:

        Element();

        ~Element();

        void run(bool r)
        {
            run_ = r;
        }

        bool run()
        {
            return run_;
        }

        bool stateful()
        {
            return stateful_;
        }

        void stateful(bool s)
        {
            stateful_ = s;
        }

        virtual int fetch_msg();
        virtual int start();
        virtual int on_receive_message(std::string &msg);
        virtual int on_receive_req(std::string &msg, const int code);
        virtual int on_receive_resp(std::string &msg, const int code);
        virtual int invite_request();
        virtual int register_request();
        virtual int bye_request();
        virtual int cancel_request();
        virtual int update_request();
        virtual int info_request();
        virtual int ack_request();
        virtual int message_request();
        virtual int subscribe_request();
        virtual int notify_request();
        virtual int refer_request();
        virtual int options_request();
        virtual int prack_request();

        virtual int on_invite_request(RequestMessage &in_msg);
        virtual int on_register_request(RequestMessage &in_msg);
        virtual int on_bye_request(RequestMessage &in_msg);
        virtual int on_ack_request(RequestMessage &in_msg);
        virtual int on_cancel_request(RequestMessage &in_msg);
        virtual int on_options_request(RequestMessage &in_msg);
        virtual int on_refer_request(RequestMessage &in_msg);
        virtual int on_subscribe_request(RequestMessage &in_msg);
        virtual int on_notify_request(RequestMessage &in_msg);
        virtual int on_message_request(RequestMessage &in_msg);
        virtual int on_info_request(RequestMessage &in_msg);
        virtual int on_prack_request(RequestMessage &in_msg);
        virtual int on_update_request(RequestMessage &in_msg);

        virtual void send_msg(RequestMessage &msg);
        virtual void send_msg(ResponseMessage &msg);

        virtual void echo(RequestMessage &in_msg);
        virtual void simple_response(const RespCode &rc, RequestMessage &in_msg);

        template<typename T>
        int dialog_preprocess(Dialog &dialog, T &in_msg)
        {
            if (!dialogs_[dialog.id()])
            {
                // TODO: configurable reject/accept
                if (true)
                {
                    simple_response(SIP_RESPONSE_CALL_OR_TRANSACTION_NOT_EXIST, in_msg);
                    return MESSAGE_PROCESSED;
                }
                else
                {
                    // TODO: restruct dialog
                }
            }

            return PROCEDURE_OK;
        }
    };

} // namespace EasySip
```

```
/*
 * include/Element/registar.h
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#pragma once

#include "Element/element.h"

namespace EasySip
{
 class Registar : public Element
 {
 protected:

  std::set<Contact> uri_binds_;

 public:

  Registar();

  ~Registar()
  {
  }

//  virtual int invite_request();
//  virtual int register_request();
//  virtual int bye_request();
//  virtual int cancel_request();
//  virtual int update_request();
//  virtual int info_request();
//  virtual int ack_request();
//  virtual int message_request();
//  virtual int subscribe_request();
//  virtual int notify_request();
//  virtual int refer_request();
//  virtual int options_request();
//  virtual int prack_request();
//
//  virtual int on_invite_request(RequestMessage &in_msg);
  virtual int on_register_request(RequestMessage &in_msg);
//  virtual int on_bye_request(RequestMessage &in_msg);
//  virtual int on_ack_request(RequestMessage &in_msg);
//  virtual int on_cancel_request(RequestMessage &in_msg);
//  virtual int on_options_request(RequestMessage &in_msg);
//  virtual int on_refer_request(RequestMessage &in_msg);
//  virtual int on_subscribe_request(RequestMessage &in_msg);
//  virtual int on_notify_request(RequestMessage &in_msg);
//  virtual int on_message_request(RequestMessage &in_msg);
//  virtual int on_info_request(RequestMessage &in_msg);
//  virtual int on_prack_request(RequestMessage &in_msg);
//  virtual int on_update_request(RequestMessage &in_msg);
//  virtual int on_response(Message &in_msg);
 };

} // namespace EasySip
```

```
/*
 * include/Element/uaclient.h
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#pragma once

#include "Element/element.h"

namespace EasySip
{
 class UAClient : public Element
 {
 public:
  UAClient();

  ~UAClient()
  {
  }

 };

} // namespace EasySip
```

```
/*
 * include/response_code.h
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#pragma once

#include "parameter.h"

namespace EasySip
{
    typedef CodeMap RespCode;
    typedef CodeMap WarnCode;
    typedef std::set<RespCode> RespCodeList;

    // RFC-3261
    // 1xx provisional
    const RespCode SIP_RESPONSE_TRYING(100, "Trying");
    const RespCode SIP_RESPONSE_RINGING(180, "Ringing");
    const RespCode SIP_RESPONSE_FORWARDING(181, "Call is Being Forwared");
    const RespCode SIP_RESPONSE_QUEUED(182, "Queued");
    const RespCode SIP_RESPONSE_SESSION_PROGRESS(183, "Session Progress");
    // 2xx successful
    const RespCode SIP_RESPONSE_SUCCESSFUL(200, "OK");
    const RespCode SIP_RESPONSE_ACCEPTED(202, "Accepted");
    // 3xx redirection
    const RespCode SIP_RESPONSE_MULTI_CHOICES(300, "Multiple Choices");
    const RespCode SIP_RESPONSE_MOVE_PERM(301, "Moved Permanently");
    const RespCode SIP_RESPONSE_MOVE_TEMP(302, "Moved Temporarily");
    const RespCode SIP_RESPONSE_USE_PROXY(305, "Use Proxy");
    const RespCode SIP_RESPONSE_ALTER_SERVICE(380, "Alternative Service");
    // 4xx request failure
    const RespCode SIP_RESPONSE_BAD_REQUEST(400, "Bad Request");
    const RespCode SIP_RESPONSE_UNAUTHORIZED(401, "Unauthorized");
    const RespCode SIP_RESPONSE_REQUIRE_PAYMENT(402, "Payment Required");
    const RespCode SIP_RESPONSE_FORBIDDEN(403, "Forbidden");
    const RespCode SIP_RESPONSE_NOT_FOUND(404, "Not Found");
    const RespCode SIP_RESPONSE_METHOD_NOT_ALLOWED(405, "Method Not Allowed");
    const RespCode SIP_RESPONSE_NOT_ACCEPTABLE(406, "Not Acceptable");
    const RespCode SIP_RESPONSE_REQUIRE_PROXY_AUTHENTICATION(407, "Proxy Authentication Required");
    const RespCode SIP_RESPONSE_REQUIRE_REQUEST_TIMEOUT(408, "Request Timeout");
    const RespCode SIP_RESPONSE_RESOURCE_NOT_AVAIL(410, "Gone");
    const RespCode SIP_RESPONSE_REQUEST_ENTITY_TOO_LARGE(413, "Request Entity Too Large");
    const RespCode SIP_RESPONSE_REQUEST_URI_TOO_LONG(414, "Request-URI Too Large");
    const RespCode SIP_RESPONSE_UNSUPPORTED_MEDIA_TYPE(415, "Unsupported Media Type");
    const RespCode SIP_RESPONSE_UNSUPPORTED_URI_SCHEME(416, "Unsupported URI Scheme");
    const RespCode SIP_RESPONSE_BAD_EXTENSION(420, "Bad Extension");
    const RespCode SIP_RESPONSE_REQUIRE_EXTENSION(421, "Extension Required");
    const RespCode SIP_RESPONSE_INTERVAL_TOO_BRIEF(423, "Interval Too Brief");
    const RespCode SIP_RESPONSE_UNAVAIL_TEMP(480, "Temporarily not available");
    const RespCode SIP_RESPONSE_CALL_OR_TRANSACTION_NOT_EXIST(481, "Call Leg/Transaction Does Not Exist");
    const RespCode SIP_RESPONSE_LOOP_DETECTED(482, "Loop Detected");
    const RespCode SIP_RESPONSE_TOO_MANY_HOPS(483, "Too Many Hops");
    const RespCode SIP_RESPONSE_ADDRESS_INCOMPLETE(484, "Address Incomplete");
    const RespCode SIP_RESPONSE_AMBIGUOUS_URI(485, "Ambiguous");
    const RespCode SIP_RESPONSE_BUSY(486, "Busy Here");
    const RespCode SIP_RESPONSE_REQUEST_TERMINATED(487, "Request Terminated");
    const RespCode SIP_RESPONSE_NOT_ACCEPTABLE_HERE(488, "Not Acceptable Here");
    const RespCode SIP_RESPONSE_REQUEST_PENDING(491, "Request Pending");
    const RespCode SIP_RESPONSE_UNDECIPHERABLE(493, "Undecipherable");
    // 5xx server failure
    const RespCode SIP_RESPONSE_SERVER_INTERNAL_ERROR(500, "Internal Server Error");
    const RespCode SIP_RESPONSE_FUNC_NOT_IMPLEMENTED(501, "Not Implemented");
    const RespCode SIP_RESPONSE_BAD_GATEWAY(502, "Bad Gateway");
    const RespCode SIP_RESPONSE_SERVICE_UNAVAIL(503, "Service Unavailable");
    const RespCode SIP_RESPONSE_SERVICE_TIMEOUT(504, "Service Time-out");
    const RespCode SIP_RESPONSE_UNSUPPORTED_VERSION(505, "SIP Version not supported");
    const RespCode SIP_RESPONSE_MESSAGE_TOO_LARGE(513, "Message Too Large");
    // 6xx global failures
    const RespCode SIP_RESPONSE_GLOBAL_BUSY(600, "Busy Everywhere");
    const RespCode SIP_RESPONSE_CALLEE_DECLINE(603, "Decline");
    const RespCode SIP_RESPONSE_GLOBAL_NOT_EXIST(604, "Does not exist anywhere");
    const RespCode SIP_RESPONSE_GLOBAL_NOT_ACCEPTABLE(606, "Not Acceptable");

    // Warning codes in response (Warning hearder field)
    // RFC-3261
    const WarnCode SIP_WARNING_300(300, "Incompatible network protocol");
    const WarnCode SIP_WARNING_301(301, "Incompatible network address formats");
    const WarnCode SIP_WARNING_302(302, "Incompatible transport protocol");
    const WarnCode SIP_WARNING_303(303, "Incompatible bandwidth units");
    const WarnCode SIP_WARNING_304(304, "Media type not available");
    const WarnCode SIP_WARNING_305(305, "Incompatible media format");
    const WarnCode SIP_WARNING_306(306, "Attribute not understood");
    const WarnCode SIP_WARNING_307(307, "Session description parmeter not understood");
    const WarnCode SIP_WARNING_330(330, "Multicast not available");
    const WarnCode SIP_WARNING_331(331, "Unicast not available");
    const WarnCode SIP_WARNING_370(370, "Insufficient bandwidth");
    // RFC-5630
    const WarnCode SIP_WARNING_380(380, "SIPS Not Allowed");
    const WarnCode SIP_WARNING_381(381, "SIPS Required");
    // RFC-3261
    const WarnCode SIP_WARNING_399(399, "Miscellaneous warning");

} // namespace EasySip
```

```
/*
 * include/message.h
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#pragma once

#include "header_field.h"

namespace EasySip
{
    /*
     * Mandatory fields:
     * Call-ID
     * CSeq
     * From
     * To
     * Via
     * Max-Forwards
    */
    class Message : public HeaderFields
    {
    protected:

        typedef Message Ancestor;

        std::string user_data_;
        std::string msg_; // message to send or received, which contains header fields and user data

    public:

        Message()
        {
        }

        Message(std::string msg)
        : msg_(msg)
        {
        }

        std::string Msg()
        {
            return msg_;
        }

        ~Message()
        {
        }

        MethodMap method()
        {
            return req_line_->method_;
        }

        std::string& append_userdata(std::string buf)
        {
            user_data_.append(buf);
            return user_data_;
        }

        virtual Message& create();
        virtual bool is_valid();

        virtual int parse(size_t &pos);

        #define parse_field(f, msg, pos) \
        { \
            f.last()->parse(msg, pos); \
        }

        virtual void parse_dispatch(std::string field, size_t &pos);

        static int get_method_from_buffer(
            MethodMapList &allowed_methods, std::string msg, std::string sym = " ");

        static int get_response_code_from_buffer(
            RespCodeList &allowed_responses, std::string msg, std::string sym = " ");

        static std::vector<std::string> split_by(std::string msg, std::string sym = " ");

        friend std::ostream& operator<< (std::ostream& o, Message& hf);


        // shotcut for each header field
         HFCallId* add_call_id();
         HFCSeq* add_cseq();
         HFFrom* add_from();
         HFTo* add_to();
         HFVia* add_via();
         HFAlertInfo* add_alert_info();
         HFAllowEvents* add_allow_events();
         HFDate* add_date();
         HFContact* add_contact();
         HFOrganization* add_organization();
         HFRecordRoute* add_record_route();
         HFRetryAfter* add_retry_after();
         HFSubject* add_subject();
         HFSupported* add_supported();
         HFTimestamp* add_timestamp();
         HFUserAgent* add_user_agent();
         HFAnswerMode* add_answer_mode();
         HFPrivAnswerMode* add_priv_answer_mode();
         HFAccept* add_accept();
         HFAcceptContact* add_accept_contact();
         HFAcceptEncoding* add_accept_encoding();
         HFAcceptLanguage* add_accept_language();
         HFAuthorization* add_authorization();
         HFCallInfo* add_call_info();
         HFEvent* add_event();
         HFInReplyTo* add_in_replay_to();
         HFJoin* add_join();
         HFPriority* add_priority();
         HFPrivacy* add_privacy();
         HFProxyAuthorization* add_proxy_authorization();
         HFProxyRequire* add_proxy_require();
         HFPOSPAuthToken* add_p_osp_auth_token();
         HFPAssertedIdentity* add_p_asserted_identity();
         HFPPreferredIdentity* add_p_preferred_identity();
         HFMaxForwards* add_max_forwards();
         HFReason* add_reason();
         HFReferTo* add_refer_to();
         HFReferredBy* add_referred_by();
         HFReplyTo* add_reply_to();
         HFReplaces* add_replaces();
         HFRejectContact* add_reject_contact();
         HFRequestDisposition* add_request_disposition();
         HFRequire* add_require();
         HFRoute* add_route();
         HFRack* add_rack();
         HFSessionExpires* add_session_expires();
         HFSubscriptionState* add_subscription_state();
         HFAuthenticationInfo* add_authentication_info();
         HFErrorInfo* add_error_info();
         HFMinExpires* add_min_expires();
         HFMinSE* add_min_se();
         HFProxyAuthenticate* add_proxy_authenticate();
```

```
     HFServer* add_server();
     HFUnsupported* add_unsupported();
     HFWarning* add_warning();
     HFWWWAuthenticate* add_www_authenticate();
     HFRSeq* add_rseq();
     HFAllow* add_allow();
     HFContentEncoding* add_content_encoding();
     HFContentLength* add_content_length();
    HFContentDisposition* add_content_disposition();
     HFContentLanguage* add_content_language();
     HFContentType* add_content_type();
     HFExpires* add_expires();
     HFMIMEVersion* add_mime_version();
};

// ---------------- Request messages --------------------------
class ResponseMessage;

class RequestMessage : public Message
{
public:
    RequestMessage()
    {
        req_line_ = std::make_shared<RequestLine>();
    }

    RequestMessage(std::string &in_msg)
    {
        req_line_ = std::make_shared<RequestLine>();
        msg_ = in_msg;
    }

    RequestMessage(RequestMessage &in_msg)
    {
        req_line_ = std::make_shared<RequestLine>();
        *this = in_msg;
    }

    RequestMessage(ResponseMessage &in_msg);

    RequestMessage& create();

    virtual int parse(size_t &pos);

    virtual int parse()
    {
        size_t pos = 0;
        return parse(pos);
    }

    void SipVersion(std::string ver)
    {
        req_line_->version_ = ver;
    }

    std::string SipVersion()
    {
        return req_line_->version_;
    }

    void RequestURI(std::string ver)
    {
        req_line_->request_uri_ = ver;
    }

    std::string RequestURI()
    {
        return req_line_->request_uri_;
    }

    std::string Method()
    {
        return req_line_->method_.name();
    }
};

class InviteMessage : public RequestMessage
{
public:

    InviteMessage()
    : RequestMessage()
    {
        req_line_->method_ = METHOD_INVITE;
    }

    InviteMessage(std::string &in_msg)
    : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_INVITE;
    }

    InviteMessage(RequestMessage &in_msg)
    : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_INVITE;
    }

    ~InviteMessage()
    {
    }

    bool is_valid();
};

class RegisterMessage : public RequestMessage
{
public:
    RegisterMessage()
    : RequestMessage()
    {
        req_line_->method_ = METHOD_REGISTER;
    }

    RegisterMessage(std::string &in_msg)
    : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_REGISTER;
        msg_ = in_msg;
    }

    RegisterMessage(RequestMessage &in_msg)
    : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_REGISTER;
    }

    ~RegisterMessage()
    {
    }

    bool is_valid();
};

class ByeMessage : public RequestMessage
{
```

```
public:
    ByeMessage() : RequestMessage()
    {
        req_line_->method_ = METHOD_BYE;
    }

    ByeMessage(std::string &in_msg)
    : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_BYE;
    }

    ByeMessage(RequestMessage &in_msg)
    : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_BYE;
    }

    ~ByeMessage()
    {
    }

    bool is_valid();
};

class AckMessage : public RequestMessage
{
public:
    AckMessage() : RequestMessage()
    {
        req_line_->method_ = METHOD_ACK;
    }

    AckMessage(std::string &in_msg)
    : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_ACK;
    }

    AckMessage(RequestMessage &in_msg)
    : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_ACK;
    }

    AckMessage(ResponseMessage &in_msg);

    ~AckMessage()
    {
    }

    bool is_valid();
};

class CancelMessage : public RequestMessage
{
public:
    CancelMessage() : RequestMessage()
    {
        req_line_->method_ = METHOD_CANCEL;
    }

    CancelMessage(std::string &in_msg)
    : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_CANCEL;
    }
    CancelMessage(RequestMessage &in_msg)
    : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_CANCEL;
    }

    ~CancelMessage()
    {
    }

    bool is_valid();
};

class OptionsMessage : public RequestMessage
{
public:
    OptionsMessage() : RequestMessage()
    {
        req_line_->method_ = METHOD_OPTIONS;
    }

    OptionsMessage(std::string &in_msg)
    : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_OPTIONS;
    }

    OptionsMessage(RequestMessage &in_msg)
    : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_OPTIONS;
    }

    ~OptionsMessage()
    {
    }

    bool is_valid();
};

class ReferMessage : public RequestMessage
{
public:
    ReferMessage() : RequestMessage()
    {
        req_line_->method_ = METHOD_REFER;
    }

    ReferMessage(std::string &in_msg) : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_REFER;
    }

    ReferMessage(RequestMessage &in_msg) : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_REFER;
    }

    ~ReferMessage()
    {
    }

    bool is_valid();
};

class SubscribeMessage : public RequestMessage
{
public:
    SubscribeMessage() : RequestMessage()
    {
        req_line_->method_ = METHOD_SUBSCRIBE;
    }

    SubscribeMessage(std::string &in_msg) : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_SUBSCRIBE;
    }

    SubscribeMessage(RequestMessage &in_msg) : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_SUBSCRIBE;
    }

    ~SubscribeMessage()
    {
    }

    bool is_valid();
};

class NotifyMessage : public RequestMessage
{
public:
    NotifyMessage() : RequestMessage()
    {
        req_line_->method_ = METHOD_NOTIFY;
    }

    NotifyMessage(std::string &in_msg) : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_NOTIFY;
    }

    NotifyMessage(RequestMessage &in_msg) : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_NOTIFY;
    }

    ~NotifyMessage()
    {
    }

    bool is_valid();
};

class MessageMessage : public RequestMessage
{
public:
    MessageMessage() : RequestMessage()
    {
        req_line_->method_ = METHOD_MESSAGE;
    }

    MessageMessage(std::string &in_msg) : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_MESSAGE;
    }

    MessageMessage(RequestMessage &in_msg) : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_MESSAGE;
    }

    ~MessageMessage()
    {
    }
    bool is_valid();
};

class InfoMessage : public RequestMessage
{
public:
    InfoMessage() : RequestMessage()
    {
        req_line_->method_ = METHOD_INFO;
    }

    InfoMessage(std::string &in_msg) : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_INFO;
    }

    InfoMessage(RequestMessage &in_msg) : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_INFO;
    }

    ~InfoMessage()
    {
    }

    bool is_valid();
};

class PrackMessage : public RequestMessage
{
public:
    PrackMessage() : RequestMessage()
    {
        req_line_->method_ = METHOD_PRACK;
    }

    PrackMessage(std::string &in_msg) : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_PRACK;
    }

    PrackMessage(RequestMessage &in_msg) : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_PRACK;
    }

    ~PrackMessage()
    {
    }

    bool is_valid();
};

class UpdateMessage : public RequestMessage
{
public:
    UpdateMessage() : RequestMessage()
    {
        req_line_->method_ = METHOD_UPDATE;
    }

    UpdateMessage(std::string &in_msg) : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_UPDATE;
    }

    UpdateMessage(RequestMessage &in_msg) : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_UPDATE;
    }

    ~UpdateMessage()
    {
    }

    bool is_valid();
};

// ---------------- Response messages --------------------------
class ResponseMessage : public Message
{

public:
    ResponseMessage()
    {
        resp_status_ = std::make_shared<ResponseStatus>();
    }

    ResponseMessage(std::string &msg)
    {
        resp_status_ = std::make_shared<ResponseStatus>();
        msg_ = msg;
    }

    ResponseMessage(const RespCode &resp)
    {
        resp_status_ = std::make_shared<ResponseStatus>();
        resp_status_->resp_code_ = resp;
    }

    ResponseMessage(RequestMessage &in_msg);

    ~ResponseMessage()
    {
    }

    ResponseStatus& RespStatus()
    {
        return *resp_status_;
    }

    void ResponseCode(const RespCode& resp)
    {
        resp_status_->resp_code_ = resp;
    }

    RespCode& ResponseCode()
    {
        return resp_status_->resp_code_;
    }

    void SipVersion(std::string ver)
    {
        resp_status_->version_ = ver;
    }

    std::string& SipVersion()
    {
        return resp_status_->version_;
    }

    bool is_1xx_resp()
    {
        return (99 < resp_status_->resp_code_.code() && 200 > resp_status_->resp_code_.code());
    }
```

```
        bool is_2xx_resp()
        {
            return (199 < resp_status_->resp_code_.code() && 300 > resp_status_->resp_code_.code());
        }

        bool is_3xx_resp()
        {
            return (299 < resp_status_->resp_code_.code() && 400 > resp_status_->resp_code_.code());
        }

        bool is_4xx_resp()
        {
            return (399 < resp_status_->resp_code_.code() && 500 > resp_status_->resp_code_.code());
        }

        bool is_5xx_resp()
        {
            return (499 < resp_status_->resp_code_.code() && 600 > resp_status_->resp_code_.code());
        }

        bool is_6xx_resp()
        {
            return (599 < resp_status_->resp_code_.code() && 700 > resp_status_->resp_code_.code());
        }

        bool is_resp2invite()
        {
            return (METHOD_INVITE.name() == cseq_.first()->method());
        }

        bool is_resp2register()
        {
            return (METHOD_REGISTER.name() == cseq_.first()->method());
        }


        virtual ResponseMessage& create();

        virtual int parse(size_t &pos);

        virtual int parse()
        {
            size_t pos = 0;
            return parse(pos);
        }

    };

//    class MessageQueue : public std::queue<Message>
//    {
//    public:
//        void append(Message &msg)
//        {
//        }
//
//        void append(RequestMessage &msg)
//        {
//        }
//
//        void append(ResponseMessage &msg)
//        {
//        }
//    };
} // namespace EasySip
```

```
/*
 * src/transaction.cpp
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#include "transaction.h"

namespace EasySip
{
    int Transaction::loop()
    {
        switch (state_)
        {
            case T_FSM_CALLING:
            {
                T1 t1;
                //element_.invite_request();
                break;
            }
            case T_FSM_TRYING:
            {
                break;
            }
            case T_FSM_PROCEEDING:
            {
                break;
            }
            case T_FSM_COMPLETED:
            {
                break;
            }
            case T_FSM_CONFIRMED:
            {
                break;
            }
            case T_FSM_TERMINATED:
            {
                break;
            }
            default:
            {
            //    return PROCEDURE_ERROR;
            }
        }

        return PROCEDURE_OK;
    }
}; // namespace EasySip
```

```
/*
 * src/except.cpp
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#include "except.h"

namespace EasySip
{
    const char* Except::what()
    {
        return msg_.c_str();
    }
} // namespace EasySip
```

```
/*
 * src/message.cpp
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#include "message.h"

namespace EasySip
{
    HFCallId* Message::add_call_id()
    {
        call_id_.append_item();
        return call_id_.last();
    }

    HFCSeq* Message::add_cseq()
    {
        cseq_.append_item();
        return cseq_.last();
    }

    HFFrom* Message::add_from()
    {
        from_.append_item();
        return from_.last();
    }

    HFTo* Message::add_to()
    {
        to_.append_item();
        return to_.last();
    }

    HFVia* Message::add_via()
    {
        via_.append_item();
        return via_.last();
    }

    HFAlertInfo* Message::add_alert_info()
    {
        alert_info_.append_item();
        return alert_info_.last();
    }

    HFAllowEvents* Message::add_allow_events()
    {
        allow_events_.append_item();
        return allow_events_.last();
    }

    HFDate* Message::add_date()
    {
        date_.append_item();
        return date_.last();
    }

    HFContact* Message::add_contact()
    {
        contact_.append_item();
        return contact_.last();
    }

    HFOrganization* Message::add_organization()
    {
        organization_.append_item();
        return organization_.last();
    }
```

```
HFRecordRoute* Message::add_record_route()
{
   record_route_.append_item();
   return record_route_.last();
}

HFRetryAfter* Message::add_retry_after()
{
   retry_after_.append_item();
   return retry_after_.last();
}

HFSubject* Message::add_subject()
{
   subject_.append_item();
   return subject_.last();
}

HFSupported* Message::add_supported()
{
   supported_.append_item();
   return supported_.last();
}

HFTimestamp* Message::add_timestamp()
{
   timestamp_.append_item();
   return timestamp_.last();
}

HFUserAgent* Message::add_user_agent()
{
   user_agent_.append_item();
   return user_agent_.last();
}

HFAnswerMode* Message::add_answer_mode()
{
   answer_mode_.append_item();
   return answer_mode_.last();
}

HFPrivAnswerMode* Message::add_priv_answer_mode()
{
   priv_answer_mode_.append_item();
   return priv_answer_mode_.last();
}

HFAccept* Message::add_accept()
{
   accept_.append_item();
   return accept_.last();
}

HFAcceptContact* Message::add_accept_contact()
{
   accept_contact_.append_item();
   return accept_contact_.last();
}

HFAcceptEncoding* Message::add_accept_encoding()
{
   accept_encoding_.append_item();
   return accept_encoding_.last();
}

HFAcceptLanguage* Message::add_accept_language()
```

```
{
   accept_language_.append_item();
   return accept_language_.last();
}

HFAuthorization* Message::add_authorization()
{
   authorization_.append_item();
   return authorization_.last();
}

HFCallInfo* Message::add_call_info()
{
   call_info_.append_item();
   return call_info_.last();
}

HFEvent* Message::add_event()
{
   event_.append_item();
   return event_.last();
}

HFInReplyTo* Message::add_in_replay_to()
{
   in_replay_to_.append_item();
   return in_replay_to_.last();
}

HFJoin* Message::add_join()
{
   join_.append_item();
   return join_.last();
}

HFPriority* Message::add_priority()
{
   priority_.append_item();
   return priority_.last();
}

HFPrivacy* Message::add_privacy()
{
   privacy_.append_item();
   return privacy_.last();
}

HFProxyAuthorization* Message::add_proxy_authorization()
{
   proxy_authorization_.append_item();
   return proxy_authorization_.last();
}

HFProxyRequire* Message::add_proxy_require()
{
   proxy_require_.append_item();
   return proxy_require_.last();
}

HFPOSPAuthToken* Message::add_p_osp_auth_token()
{
   p_osp_auth_token_.append_item();
   return p_osp_auth_token_.last();
}

HFPAssertedIdentity* Message::add_p_asserted_identity()
{
   p_asserted_identity_.append_item();
```

```
    return p_asserted_identity_.last();
}

HFPPreferredIdentity* Message::add_p_preferred_identity()
{
    p_preferred_identity_.append_item();
    return p_preferred_identity_.last();
}

HFMaxForwards* Message::add_max_forwards()
{
    max_forwards_.append_item();
    return max_forwards_.last();
}

HFReason* Message::add_reason()
{
    reason_.append_item();
    return reason_.last();
}

HFReferTo* Message::add_refer_to()
{
    refer_to_.append_item();
    return refer_to_.last();
}

HFReferredBy* Message::add_referred_by()
{
    referred_by_.append_item();
    return referred_by_.last();
}

HFReplyTo* Message::add_reply_to()
{
    reply_to_.append_item();
    return reply_to_.last();
}

HFReplaces* Message::add_replaces()
{
    replaces_.append_item();
    return replaces_.last();
}

HFRejectContact* Message::add_reject_contact()
{
    reject_contact_.append_item();
    return reject_contact_.last();
}

HFRequestDisposition* Message::add_request_disposition()
{
    request_disposition_.append_item();
    return request_disposition_.last();
}

HFRequire* Message::add_require()
{
    require_.append_item();
    return require_.last();
}

HFRoute* Message::add_route()
{
    route_.append_item();
    return route_.last();
}
```

```
HFRack* Message::add_rack()
{
   rack_.append_item();
   return rack_.last();
}

HFSessionExpires* Message::add_session_expires()
{
   session_expires_.append_item();
   return session_expires_.last();
}

HFSubscriptionState* Message::add_subscription_state()
{
   subscription_state_.append_item();
   return subscription_state_.last();
}

HFAuthenticationInfo* Message::add_authentication_info()
{
   authentication_info_.append_item();
   return authentication_info_.last();
}

HFErrorInfo* Message::add_error_info()
{
   error_info_.append_item();
   return error_info_.last();
}

HFMinExpires* Message::add_min_expires()
{
   min_expires_.append_item();
   return min_expires_.last();
}

HFMinSE* Message::add_min_se()
{
   min_se_.append_item();
   return min_se_.last();
}

HFProxyAuthenticate* Message::add_proxy_authenticate()
{
   proxy_authenticate_.append_item();
   return proxy_authenticate_.last();
}

HFServer* Message::add_server()
{
   server_.append_item();
   return server_.last();
}

HFUnsupported* Message::add_unsupported()
{
   unsupported_.append_item();
   return unsupported_.last();
}

HFWarning* Message::add_warning()
{
   warning_.append_item();
   return warning_.last();
}

HFWWWAuthenticate* Message::add_www_authenticate()
```

```
{
   www_authenticate_.append_item();
   return www_authenticate_.last();
}

HFRSeq* Message::add_rseq()
{
   rseq_.append_item();
   return rseq_.last();
}

HFAllow* Message::add_allow()
{
   allow_.append_item();
   return allow_.last();
}

HFContentEncoding* Message::add_content_encoding()
{
   content_encoding_.append_item();
   return content_encoding_.last();
}

HFContentLength* Message::add_content_length()
{
   content_length_.append_item();
   content_length_.last()->length(user_data_.size());
   return content_length_.last();
}

HFContentDisposition* Message::add_content_disposition()
{
   content_disposition_.append_item();
   return content_disposition_.last();
}

HFContentLanguage* Message::add_content_language()
{
   content_language_.append_item();
   return content_language_.last();
}

HFContentType* Message::add_content_type()
{
   content_type_.append_item();
   return content_type_.last();
}

HFExpires* Message::add_expires()
{
   expires_.append_item();
   return expires_.last();
}

HFMIMEVersion* Message::add_mime_version()
{
   mime_version_.append_item();
   return mime_version_.last();
}

std::ostream& operator<< (std::ostream& o, Message& msg)
{
   out_if_not_null(o, msg.req_line_);
   out_if_not_null(o, msg.resp_status_);
   out_if_not_empty(o, msg.call_id_);
   out_if_not_empty(o, msg.cseq_);
   out_if_not_empty(o, msg.from_);
   out_if_not_empty(o, msg.to_);
```

```
        out_if_not_empty(o, msg.via_);
        out_if_not_empty(o, msg.alert_info_);
        out_if_not_empty(o, msg.allow_events_);
        out_if_not_empty(o, msg.date_);
        out_if_not_empty(o, msg.contact_);
        out_if_not_empty(o, msg.organization_);
        out_if_not_empty(o, msg.record_route_);
        out_if_not_empty(o, msg.retry_after_);
        out_if_not_empty(o, msg.subject_);
        out_if_not_empty(o, msg.supported_);
        out_if_not_empty(o, msg.timestamp_);
        out_if_not_empty(o, msg.user_agent_);
        out_if_not_empty(o, msg.answer_mode_);
        out_if_not_empty(o, msg.priv_answer_mode_);
        out_if_not_empty(o, msg.accept_);
        out_if_not_empty(o, msg.accept_contact_);
        out_if_not_empty(o, msg.accept_encoding_);
        out_if_not_empty(o, msg.accept_language_);
        out_if_not_empty(o, msg.authorization_);
        out_if_not_empty(o, msg.call_info_);
        out_if_not_empty(o, msg.event_);
        out_if_not_empty(o, msg.in_replay_to_);
        out_if_not_empty(o, msg.join_);
        out_if_not_empty(o, msg.priority_);
        out_if_not_empty(o, msg.privacy_);
        out_if_not_empty(o, msg.proxy_authorization_);
        out_if_not_empty(o, msg.proxy_require_);
        out_if_not_empty(o, msg.p_osp_auth_token_);
        out_if_not_empty(o, msg.p_asserted_identity_);
        out_if_not_empty(o, msg.p_preferred_identity_);
        out_if_not_empty(o, msg.max_forwards_);
        out_if_not_empty(o, msg.reason_);
        out_if_not_empty(o, msg.refer_to_);
        out_if_not_empty(o, msg.referred_by_);
        out_if_not_empty(o, msg.reply_to_);
        out_if_not_empty(o, msg.replaces_);
        out_if_not_empty(o, msg.reject_contact_);
        out_if_not_empty(o, msg.request_disposition_);
        out_if_not_empty(o, msg.require_);
        out_if_not_empty(o, msg.route_);
        out_if_not_empty(o, msg.rack_);
        out_if_not_empty(o, msg.session_expires_);
        out_if_not_empty(o, msg.subscription_state_);
        out_if_not_empty(o, msg.authentication_info_);
        out_if_not_empty(o, msg.error_info_);
        out_if_not_empty(o, msg.min_expires_);
        out_if_not_empty(o, msg.min_se_);
        out_if_not_empty(o, msg.proxy_authenticate_);
        out_if_not_empty(o, msg.server_);
        out_if_not_empty(o, msg.unsupported_);
        out_if_not_empty(o, msg.warning_);
        out_if_not_empty(o, msg.www_authenticate_);
        out_if_not_empty(o, msg.rseq_);
        out_if_not_empty(o, msg.allow_);
        out_if_not_empty(o, msg.content_encoding_);
//      out_if_not_empty(o, msg.content_length_);
        out_if_not_empty(o, msg.content_disposition_);
        out_if_not_empty(o, msg.content_language_);
        out_if_not_empty(o, msg.content_type_);
        out_if_not_empty(o, msg.expires_);
        out_if_not_empty(o, msg.mime_version_);
        out_if_not_empty(o, msg.content_length_);

        o << msg.user_data_ << "\r\n";

        return o;
    }
    Message& Message::create()
    {
        if (!is_valid())
        {
            // TODO: thown exception and log this
            std::cerr << __PRETTY_FUNCTION__ << ": message invalid!\n";
            return *this;
        }

        return *this;
    }

    bool Message::is_valid()
    {
        return_false_if_true(call_id_.empty())
        return_false_if_true(cseq_.empty())
        return_false_if_true(from_.empty())
        return_false_if_true(to_.empty())
        return_false_if_true(via_.empty())
//      return_false_if_true(max_forwards_.empty() && resp_status_.empty())

        return true;
    }

    int Message::get_method_from_buffer(
        MethodMapList &allowed_methods, std::string msg, std::string sym)
    {
        std::string ret = msg.substr(0, msg.find_first_of(sym));

        for (auto &it : allowed_methods)
        {
            if (ret == it.second)
                return it.code();
        }

        return -1;
    }

    int Message::get_response_code_from_buffer(
        RespCodeList &allowed_responses, std::string msg, std::string sym)
    {
        RespCodeList::iterator it;
        size_t pos = msg.find_first_of(sym);

        if (pos == std::string::npos)
            return -1;

        pos++;
        int next = msg.find_first_of(sym, pos);
        std::string ret = msg.substr(pos, next-pos);

        for (it = allowed_responses.begin(); it != allowed_responses.end(); it++)
        {
            if (ret == it->CodeStr())
                return it->code();
        }

        return -1;
    }

    std::vector<std::string> Message::split_by(std::string msg, std::string sym)
    {
        size_t pos, next;
        std::vector<std::string> ret;

        for (next = pos = 0; ; pos = next+1)
        {
            next = msg.find_first_of(sym, pos);
            if (next == std::string::npos)
                break;

            ret.push_back(msg.substr(pos, next-pos));
        }

        return ret;
    }

    void Message::parse_dispatch(std::string field, size_t &pos)
    {
        switch(allowed_fields_[field])
        {
            case HF_CALLID:            add_call_id()->parse(msg_, pos); break;
            case HF_CSEQ:              add_cseq()->parse(msg_, pos); break;
            case HF_FROM:              add_from()->parse(msg_, pos); break;
            case HF_TO:                add_to()->parse(msg_, pos); break;
            case HF_VIA:               add_via()->parse(msg_, pos); break;
            case HF_ALERT_INFO:        add_alert_info()->parse(msg_, pos); break;
            case HF_ALLOW_EVENTS:      add_allow_events()->parse(msg_, pos); break;
            case HF_DATE:              add_date()->parse(msg_, pos); break;
            case HF_CONTACT:           add_contact()->parse(msg_, pos); break;
            case HF_ORGANIZATION:      add_organization()->parse(msg_, pos); break;
            case HF_RECORD_ROUTE:      add_record_route()->parse(msg_, pos); break;
            case HF_RETRY_AFTER:       add_retry_after()->parse(msg_, pos); break;
            case HF_SUBJECT:           add_subject()->parse(msg_, pos); break;
            case HF_SUPPORTED:         add_supported()->parse(msg_, pos); break;
            case HF_TIMESTAMP:         add_timestamp()->parse(msg_, pos); break;
            case HF_USER_AGENT:        add_user_agent()->parse(msg_, pos); break;
            case HF_ANSWER_MODE:       add_answer_mode()->parse(msg_, pos); break;
            case HF_PRIV_ANSWER_MODE:  add_priv_answer_mode()->parse(msg_, pos); break;
            case HF_ACCEPT:            add_accept()->parse(msg_, pos); break;
            case HF_ACCEPT_CONTACT:    add_accept_contact()->parse(msg_, pos); break;
            case HF_ACCEPT_ENCODING:   add_accept_encoding()->parse(msg_, pos); break;
            case HF_ACCEPT_LANGUAGE:   add_accept_language()->parse(msg_, pos); break;
            case HF_AUTHORIZATION:     add_authorization()->parse(msg_, pos); break;
            case HF_CALL_INFO:         add_call_info()->parse(msg_, pos); break;
            case HF_EVENT:             add_event()->parse(msg_, pos); break;
            case HF_IN_REPLY_TO:       add_in_replay_to()->parse(msg_, pos); break;
            case HF_JOIN:              add_join()->parse(msg_, pos); break;
            case HF_PRIORITY:          add_priority()->parse(msg_, pos); break;
            case HF_PRIVACY:           add_privacy()->parse(msg_, pos); break;
            case HF_PROXY_AUTHORIZATION:add_proxy_authorization()->parse(msg_, pos); break;
            case HF_PROXY_REQUIRE:     add_proxy_require()->parse(msg_, pos); break;
            case HF_P_OSP_AUTHTOKEN:   add_p_osp_auth_token()->parse(msg_, pos); break;
            case HF_PASSERTED_IDENTITY: add_p_asserted_identity()->parse(msg_, pos); break;
            case HF_PPREFERRED_IDENTITY:add_p_preferred_identity()->parse(msg_, pos); break;
            case HF_MAX_FORWARDS:      add_max_forwards()->parse(msg_, pos); break;
            case HF_REASON:            add_reason()->parse(msg_, pos); break;
            case HF_REFER_TO:          add_refer_to()->parse(msg_, pos); break;
            case HF_REFERRED_BY:       add_referred_by()->parse(msg_, pos); break;
            case HF_REPLY_TO:          add_reply_to()->parse(msg_, pos); break;
            case HF_REPLACES:          add_replaces()->parse(msg_, pos); break;
            case HF_REJECT_CONTACT:    add_reject_contact()->parse(msg_, pos); break;
            case HF_REQUEST_DISPOSITION:add_request_disposition()->parse(msg_, pos); break;
            case HF_REQUIRE:           add_require()->parse(msg_, pos); break;
            case HF_ROUTE:             add_route()->parse(msg_, pos); break;
            case HF_RACK:              add_rack()->parse(msg_, pos); break;
            case HF_SESSION_EXPIRES:   add_session_expires()->parse(msg_, pos); break;
            case HF_SUBSCRIPTION_STATE: add_subscription_state()->parse(msg_, pos); break;
            case HF_AUTHENTICATIONINFO: add_authentication_info()->parse(msg_, pos); break;
            case HF_ERROR_INFO:        add_error_info()->parse(msg_, pos); break;
            case HF_MIN_EXPIRES:       add_min_expires()->parse(msg_, pos); break;
            case HF_MIN_SE:            add_min_se()->parse(msg_, pos); break;
            case HF_PROXY_AUTHENTICATE: add_proxy_authenticate()->parse(msg_, pos); break;
            case HF_SERVER:            add_server()->parse(msg_, pos); break;
            case HF_UNSUPPORTED:       add_unsupported()->parse(msg_, pos); break;
            case HF_WARNING:           add_warning()->parse(msg_, pos); break;
            case HF_WWW_AUTHENTICATE:   add_www_authenticate()->parse(msg_, pos); break;
            case HF_RSEQ:               add_rseq()->parse(msg_, pos); break;
            case HF_ALLOW:              add_allow()->parse(msg_, pos); break;
            case HF_CONTENT_ENCODING:   add_content_encoding()->parse(msg_, pos); break;
            case HF_CONTENT_LENGTH:     add_content_length()->parse(msg_, pos); break;
            case HF_CONTENT_DISPOSITION: add_content_disposition()->parse(msg_, pos); break;
            case HF_CONTENT_LANGUAGE:   add_content_language()->parse(msg_, pos); break;
            case HF_CONTENT_TYPE:       add_content_type()->parse(msg_, pos); break;
            case HF_EXPIRES:            add_expires()->parse(msg_, pos); break;
            case HF_MIME_VERSION:       add_mime_version()->parse(msg_, pos); break;
            default:
            {
                std::cerr << "Unexpected header: " << field << '\n';
            }
        }
    }

    /* parse buffered header into formated header fields
     */
    int Message::parse(size_t &pos)
    {
        if (msg_.empty()) return PROCEDURE_ERROR;

        bool run = true;
        std::string buffer;

        while (run)
        {
            if (pos+1 >= msg_.size()) break;

            switch(msg_.at(pos))
            {
                CASE_ALPHA
                case '-':
                {
                    buffer += msg_.at(pos++);
                    break;
                }
                case '\r':
                {
                    pos++;
                    break;
                }
                case '\n':
                {
                    pos++;
                    buffer.clear();
                    break;
                }
                case ':':
                {
                    pos++;

                    parse_dispatch(buffer, pos);
                    buffer.clear();

                    break;
                }
                default:
                {
                    if (content_length_.size())
                    {
                        run = false;
                    }
                    else
                    {
                        std::cerr << __PRETTY_FUNCTION__ << " Unexpected '" << msg_.at(pos) << '(' << (int)msg_.at(pos)<< ")': " << buf
                        pos++;
                        buffer.clear();
                    }
                }
            }
        }

        if (content_length_.size())
        {
            size_t i = 1, len = 0;
            std::istringstream in(content_length_.first()->length());
            in >> len;

            while (pos < msg_.size() && i < len)
            {
                buffer += msg_.at(pos++);
            }

            user_data_ = buffer;
        }

        return PROCEDURE_OK;
    }

    RequestMessage& RequestMessage::create()
    {
        Ancestor::create();
        std::ostringstream o;

    //      std::ostringstream len;
    //      len << user_data_.size();
    //      add_content_length().length(len.str());

        o << *this;
        msg_ = o.str();

        return *this;
    }

    int RequestMessage::parse(size_t &pos)
    {
        if (msg_.empty()) return PROCEDURE_ERROR;

        int ret;

        if (PROCEDURE_OK != (ret = req_line_->parse(msg_, pos)))
            return ret;
        Ancestor::parse(pos);

        if (!is_valid())
        {
            std::cerr << __PRETTY_FUNCTION__ << ": message invalid!\n";
        }

        std::cout << "-request------------------------\n";
        std::cout << *this;
        std::cout << "--------------------------------\n";

        return PROCEDURE_OK;
    }

    RequestMessage::RequestMessage(ResponseMessage &in_msg)
    {
        req_line_ = std::make_shared<RequestLine>();

        add_call_id()
        ->id(in_msg.call_id_.first()->id());

        add_from()
        ->add_name(in_msg.from_.first()->name())
        .add_uri(in_msg.from_.first()->uri());

        for (auto &it : in_msg.from_.first()->header_params_)
        {
            from_.first()->HeaderParam(it.name(), it.value());
        }

        add_to()
        ->add_name(in_msg.to_.first()->name())
        .add_uri(in_msg.to_.first()->uri());

        for (auto &it : in_msg.to_.first()->header_params_)
        {
            to_.first()->HeaderParam(it.name(), it.value());
        }

        add_via()
        ->add_proto(SIP_VERSION_2_0_UDP)
        .add_sentby(in_msg.via_.first()->sent_by_);

        for (auto &it : in_msg.via_.first()->header_params_)
        {
            via_.first()->HeaderParam(it.name(), it.value());
        }
    }

    bool InviteMessage::is_valid()
    {
        return_false_if_true(!Ancestor::is_valid())
        return_false_if_true(contact_.empty())

        return true;
    }
    // RegisterMessage
    bool RegisterMessage::is_valid()
    {
        return_false_if_true(!Ancestor::is_valid())

        if (record_route_.size()) record_route_.clear();

        return true;
    }
    // AckMessage
    bool AckMessage::is_valid()
    {
        return_false_if_true(!Ancestor::is_valid())
        return true;
    }

    AckMessage::AckMessage(ResponseMessage &in_msg)
    : RequestMessage(in_msg)
    {
        req_line_->method_ = METHOD_ACK;

        add_cseq()
        ->cseq("1")
        .method(METHOD_ACK.name());
    }
    // ByeMessage
    bool ByeMessage::is_valid()
    {
        return_false_if_true(!Ancestor::is_valid())
        return true;
    }
    // CancelMessage
    bool CancelMessage::is_valid()
    {
        return_false_if_true(!Ancestor::is_valid())
        return true;
    }
    // OptionsMessage
    bool OptionsMessage::is_valid()
    {
        return_false_if_true(!Ancestor::is_valid())
        return true;
    }

    // ReferMessage
    bool ReferMessage::is_valid()
    {
        return_false_if_true(!Ancestor::is_valid())
        return_false_if_true(contact_.empty())
        return_false_if_true(refer_to_.empty())

        return true;
    }
    // SubscribeMessage
    bool SubscribeMessage::is_valid()
    {
        return_false_if_true(!Ancestor::is_valid())
        return_false_if_true(contact_.empty())
        return_false_if_true(event_.empty())
        return_false_if_true(allow_events_.empty())

        return true;
    }
    // NotifyMessage
    bool NotifyMessage::is_valid()
    {
        return_false_if_true(!Ancestor::is_valid())
        return_false_if_true(contact_.empty())
        return_false_if_true(event_.empty())
        return_false_if_true(allow_events_.empty())
        return_false_if_true(subscription_state_.empty())

        return true;
    }
    // MessageMessage
    bool MessageMessage::is_valid()
    {
        return_false_if_true(!Ancestor::is_valid())
        return true;
    }
    // InfoMessage
    bool InfoMessage::is_valid()
    {
        return_false_if_true(!Ancestor::is_valid())
        return true;
    }
    // PrackMessage
    bool PrackMessage::is_valid()
    {
        return_false_if_true(!Ancestor::is_valid())
        return true;
    }
    // UpdateMessage
    bool UpdateMessage::is_valid()
    {
        return_false_if_true(!Ancestor::is_valid())
        return_false_if_true(contact_.empty())

        return true;
    }

    ResponseMessage& ResponseMessage::create()
```

```
    {
        Ancestor::create();
        std::ostringstream o;

//      std::ostringstream len;
//      len << user_data_.size();
//      add_content_length().length(len.str());

        o << *this;
        msg_ = o.str();

        return *this;
    }

    int ResponseMessage::parse(size_t &pos)
    {
        if (msg_.empty()) return PROCEDURE_ERROR;

        int ret;

        if (PROCEDURE_OK != (ret = resp_status_->parse(msg_, pos)))
            return ret;

        Ancestor::parse(pos);

        if (!is_valid())
        {
            std::cerr << __PRETTY_FUNCTION__ << ": message invalid!\n";
        }

        std::cout << "-reponse------------------------\n";
        std::cout << *this;
        std::cout << "--------------------------------\n";

        return PROCEDURE_OK;
    }

    ResponseMessage::ResponseMessage(RequestMessage &in_msg)
    {
        resp_status_ = std::make_shared<ResponseStatus>();

        add_call_id()
        ->id(in_msg.call_id_.last()->id());

        add_from()
        ->add_name(in_msg.from_.last()->name())
        .add_uri(in_msg.from_.last()->uri());

        for (auto &it : in_msg.from_.last()->header_params_)
        {
            from_.last()->HeaderParam(it.name(), it.value());
        }

        add_to()
        ->add_name(in_msg.to_.last()->name())
        .add_uri(in_msg.to_.last()->uri());

        for (auto &it : in_msg.to_.last()->header_params_)
        {
            to_.last()->HeaderParam(it.name(), it.value());
        }

        add_cseq()
        ->cseq(in_msg.cseq_.last()->cseq())
        .method(in_msg.cseq_.last()->method())
        .inc_seq();

        add_via()
        ->add_proto(SIP_VERSION_2_0_UDP)
        .add_sentby(in_msg.via_.last()->sent_by_);

        for (auto &it : in_msg.via_.last()->header_params_)
        {
            via_.last()->HeaderParam(it.name(), it.value());
        }
    }
} // namespace EasySip
```

```
/*
 * src/thread.cpp
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#include "thread.h"
#include <iostream>

namespace EasySip
{
//    Thread::Thread()
//    : id_(0), routine_(0), arg_(0)
//    {
//        pthread_attr_init(&attr_);
//        pthread_create(&id_, &attr_, routine_, arg_);
//    }
//
//    Thread::~Thread()
//    {
//        pthread_attr_destroy(&attr_);
//
//        void *err = 0;
//
//        if (0 != pthread_join(id_, &err))
//        {
//            //TODO log and throw error
//            std::cerr << "pthread_join: " << reinterpret_cast<char*>(err) << '\n';
//        }
//
//        free(err);
//    }

} // namespace EasySip
```

```
/*
 * src/header_field.cpp
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#include "header_field.h"

namespace EasySip
{
    T_HF_MAP HeaderFields::allowed_fields_;

    int RequestLine::parse(std::string &msg, size_t &pos)
    {
        size_t next = 0;
        // read method
        if ((next = msg.find_first_of(" ", pos)) != std::string::npos)
        {
            method_.name(msg.substr(pos, next-pos));
            pos = next + 1;
        }

        // read request-uri
        if ((next = msg.find_first_of(" ", pos)) != std::string::npos)
        {
            request_uri_ = msg.substr(pos, next-pos);
            pos = next + 1;
        }
        // TODO: check uri scheme return code 416 on error

        // read version
        if ((next = msg.find_first_of("\n", pos)) != std::string::npos)
        {
            version_ = msg.substr(pos, next-pos);
            pos = next + 1;
        }

        return PROCEDURE_OK;
    }

    int ResponseStatus::parse(std::string &msg, size_t &pos)
    {
        size_t next = 0;
        // read version
        if ((next = msg.find_first_of(" ", pos)) != std::string::npos)
        {
            version_ = msg.substr(pos, next-pos);
            pos = next + 1;
        }
        // read code
        if ((next = msg.find_first_of(" ", pos)) != std::string::npos)
        {
            int code;
            std::istringstream in(msg.substr(pos, next-pos));
            in >> code;
            resp_code_.Code(code);
            pos = next + 1;
        }
        // read reason
        if ((next = msg.find_first_of("\n", pos)) != std::string::npos)
        {
            resp_code_.name(msg.substr(pos, next-pos));
            pos = next + 1;
        }

        return PROCEDURE_OK;
    }

    std::ostream& operator<< (std::ostream& o, HeaderField &hf)
    {
        o << hf.field_ << ": ";
        hf.generate_values();

        o << hf.Values();
        o << "\r\n";
//      o << hf.header_params_ << "\n";

        return o;
    }

    std::string HeaderField::operator() ()
    {
        std::ostringstream o;
        o << *this;

        return o.str();
    }

    int HFBase_1_::parse(std::string &msg, size_t &pos)
    {
        bool read_head_param = false, run = true, in_aquote = false, in_dquote = false;
        std::string buffer, key;

        while (msg.at(pos) == ' ' || msg.at(pos) == '\t') pos++;

        while (run)
        {
            switch (msg.at(pos))
            {
                case '"':
                {
                    in_dquote = !in_dquote;

                    buffer += msg.at(pos++);

                    if (!in_dquote)
                    {
                        add_name(buffer);
                        buffer.clear();
                    }
                    break;
                }
                CASE_TOKEN
                case '/':
                case '?':
                case ':':
                case '@':
                {
                    buffer += msg.at(pos++);
                    break;
                }
                case '<':
                {
                    in_aquote = true;
                    pos++;
                    buffer.clear();
                    break;
                }
                case '>':
                {
                    in_aquote = false;

                    if (key.size())
                    {
                        add_param(key, buffer);
                        key.clear();
                    }
                    else if (buffer.size())
                    {
                        add_uri(buffer);
                    }

                    pos++;
                    buffer.clear();
                    break;
                }
                case ',':
                {
                    if (in_dquote)
                    {
                        buffer += msg.at(pos++);
                        break;
                    }

                    if (key.size())
                    {
                        add_param(key, buffer);
                        key.clear();
                    }
                    else if (buffer.size())
                    {
                        add_uri(buffer);
                    }

                    pos++;
                    buffer.clear();
                    break;
                }
                case ';':
                {
                    if (in_aquote)
                    {
                        if (key.size())
                        {
                            add_param(key, buffer);
                            key.clear();
                        }
                        else if (buffer.size())
                        {
                            add_uri(buffer);
                        }
                    }
                    else
                    {
                        if (read_head_param)
                        {
                            header_params_.append(key, buffer);
                            key.clear();
                        }
                        else if (buffer.size())
                        {
                            add_uri(buffer);
                        }

                        if (!read_head_param)
                            read_head_param = true;
                    }

                    pos++;
                    buffer.clear();
                    break;
                }
                case '=':
                {
                    key = buffer;
```

```
                buffer.clear();
                pos++;
                break;
            }
            case ' ':
            {
                if (in_dquote)
                {
                    buffer += msg.at(pos++);
                    break;
                }

                if (in_aquote)
                {
                    pos++;
                    break;
                }

                if (buffer.size())
                {
                    add_name(buffer);
                }
                buffer.clear();
            }
            case '\t':
            case '\r':
            {
                pos++;
                break;
            }
            case '\n':
            {
                if (read_head_param)
                {
                    header_params_.append(key, buffer);
                    key.clear();
                    read_head_param = false;
                }

                else if (buffer.size())
                {
                    add_uri(buffer);
                }

                if (pos+1 >= msg.size()) { run = false; break; }
                do_if_is_alpha(msg.at(pos+1), run = false)

                pos++;
                buffer.clear();
                break;
            }
            default:
            {
                std::cerr << __PRETTY_FUNCTION__ << " Unexpected '" << msg.at(pos) << '(' << (int)msg.at(pos) << ')' << "': " << bu
                pos++;
                buffer.clear();
            }
        }
    }

    return PROCEDURE_OK;
}

void HFBase_2_::generate_values()
{
    values_ = digit_value_;
```

```
        std::stringstream p;
        p << header_params_;

        values_ += p.str();
    }

    int HFBase_2_::parse(std::string &msg, size_t &pos)
    {
        bool run = true;
        std::string buffer;

        while (msg.at(pos) == ' ' || msg.at(pos) == '\t') pos++;

        while (run)
        {
            switch (msg.at(pos))
            {
                CASE_DIGIT
                {
                    buffer += msg.at(pos++);
                    break;
                }
                case '\r':
                {
                    pos++;
                    break;
                }
                case '\n':
                {
//                  if (digit_value_.empty())
                        digit_value_ = buffer;

                    run = false;

                    pos++;
                    buffer.clear();
                    break;
                }
                default:
                {
                    std::cerr << __PRETTY_FUNCTION__ << " Unexpected '" << msg.at(pos) << '(' << (int)msg.at(pos) << ')' << "': " << bu
                    pos++;
                    buffer.clear();
                }
            }
        }

        return PROCEDURE_OK;
    }

    void HFBase_3_::generate_values()
    {
        values_.clear();

        for (auto &it : opts_)
            values_ +=  it + sym_;

        remove_tail_symbol(sym_);

        if (header_params_.size())
        {
            std::ostringstream p;
            p << ";" << header_params_;
            values_ += p.str();
        }
    }

    int HFBase_3_::parse(std::string &msg, size_t &pos)
```

```
{
    bool run = true, read_head_param = false;
    std::string buffer, key;

    while (msg.at(pos) == ' ' || msg.at(pos) == '\t') pos++;

    while (run)
    {
        switch (msg.at(pos))
        {
            case ':':
            case '/':
            case '@':
            CASE_TOKEN
            case '\t':
            case ' ':
            case ',':
            {
                if (sym_ != msg.at(pos))
                {
                    buffer += msg.at(pos++);
                    break;
                }

                if (buffer.size())
                {
                    add_value(buffer);
                }

                pos++;
                buffer.clear();
                break;
            }
            case '=':
            {
                key = buffer;

                pos++;
                buffer.clear();
                break;
            }
            case ';':
            {
                if (read_head_param)
                {
                    header_params_.append(key, buffer);
                    key.clear();
                }
                else
                {
                    if (buffer.size())
                        add_value(buffer);
                    read_head_param = true;
                }

                pos++;
                buffer.clear();
                break;
            }
            case '\r':
            {
                pos++;
                break;
            }
            case '\n':
            {
                if (read_head_param)
                {
```

```
                header_params_.append(key, buffer);
                key.clear();
            }
            else if (buffer.size())
            {
                add_value(buffer);
            }

            if (pos+1 >= msg.size()) { run = false; break; }
            do_if_is_alpha(msg.at(pos+1), run = false)

            pos++;
            buffer.clear();
            break;
        }
        default:
        {
            std::cerr << __PRETTY_FUNCTION__ << " Unexpected '" << msg.at(pos) << '(' << (int)msg.at(pos) << ')' << "': " << bu
            pos++;
            buffer.clear();
        }
      }
    }

    return PROCEDURE_OK;
}

void HFBase_4_::generate_values()
{
    char sym = ',';
    std::ostringstream o;

    for (auto &it : its_)
    {
        o << *it << sym;
    }

    values_ = o.str();
    remove_tail_symbol(sym);

    if (header_params_.size())
    {
        std::ostringstream p;
        p << ";" << header_params_;
        values_ += p.str();
    }
}

int HFBase_4_::parse(std::string &msg, size_t &pos)
{
    bool run = true, in_dquote = false;
    std::string buffer, key;

    while (msg.at(pos) == ' ' || msg.at(pos) == '\t') pos++;

    while (run)
    {
        switch (msg.at(pos))
        {
        case '"':
        {
            in_dquote = !in_dquote;
        }
        case '/':
        CASE_TOKEN
        {
            buffer += msg.at(pos++);
            break;
```

```
}
case ',':
{
   if (in_dquote)
   {
      buffer += msg.at(pos++);
      break;
   }

   if (key.size())
   {
      add_param(key, buffer);
      key.clear();
   }
   else if (buffer.size())
   {
      add_value(buffer);
   }

   pos++;
   buffer.clear();
   break;
}
case ';':
{
   if (key.size())
   {
      header_params_.append(key, buffer);
      key.clear();
   }
   else if (buffer.size())
   {
      add_value(buffer);
   }

   pos++;
   buffer.clear();
   break;
}
case '=':
{
   key = buffer;

   buffer.clear();
   pos++;
   break;
}
case ' ':
case '\t':
case '\r':
{
   pos++;
   break;
}
case '\n':
{
   if (key.size())
   {
      header_params_.append(key, buffer);
      key.clear();
   }
   else if (buffer.size())
   {
      add_value(buffer);
   }

   if (pos+1 >= msg.size()) { run = false; break; }
   do_if_is_alpha(msg.at(pos+1), run = false)
```

```
                pos++;
                buffer.clear();
                break;
            }
            default:
            {
                std::cerr << __PRETTY_FUNCTION__ << " Unexpected '" << msg.at(pos) << '(' << (int)msg.at(pos) << ')' << "': " << bu
                pos++;
                buffer.clear();
            }
        }
    }

    return PROCEDURE_OK;
}

void HFBase_5_::generate_values()
{
    char sym = ' ';
    values_ = challenge_;

    if (digest_cln_.empty())
        return;

    values_ += sym;

    std::ostringstream o;
    o << digest_cln_;

    values_ += o.str();

    remove_tail_symbol(sym);

    std::ostringstream p;
    p << header_params_;
    values_ += p.str();
}

int HFBase_5_::parse(std::string &msg, size_t &pos)
{
    bool run = true, in_dquote = false;
    std::string buffer, key;

    while (msg.at(pos) == ' ' || msg.at(pos) == '\t') pos++;

    while (run)
    {
        switch (msg.at(pos))
        {
            case '"':
            {
                in_dquote = !in_dquote;
            }
            CASE_TOKEN
            case ':':
            {
                buffer += msg.at(pos++);
                break;
            }
            case '=':
            {
                key = buffer;

                pos++;
                buffer.clear();
                break;
            }
```

```
case '\t':
case ' ':
{
    if (challenge_.size() || in_dquote)
    {
        buffer += msg.at(pos++);
        break;
    }

    if (buffer.size())
        challenge_ = buffer;

    pos++;
    buffer.clear();
    break;
}
case ',':
{
    if (in_dquote)
    {
        buffer += msg.at(pos++);
        break;
    }
    if (key.size())
    {
        digest_cln_.append(key, buffer);
        key.clear();
    }
    else
    {
        digest_cln_.append(buffer, "");
    }

    pos++;
    buffer.clear();
    break;
}
case '\r':
{
    pos++;
    break;
}
case '\n':
{
    if (challenge_.empty())
    {
        challenge_ = buffer;
    }
    else if (key.size())
    {
        digest_cln_.append(key, buffer);
        key.clear();
    }
    else
    {
        digest_cln_.append(buffer, "");
    }

    if (pos+1 >= msg.size()) { run = false; break; }
    do_if_is_alpha(msg.at(pos+1), run = false)

    pos++;
    buffer.clear();
    break;
}
default:
{
    std::cerr << __PRETTY_FUNCTION__ << " Unexpected '" << msg.at(pos) << '(' << (int)msg.at(pos) << ')' << "': " << bu
```

```
                pos++;
                buffer.clear();
            }
        }
    }
    return PROCEDURE_OK;
}

HFVia::HFVia() : HeaderField("Via", "v", true)
{
//      header_params_.append("alias");
//      header_params_.append("branch");
//      header_params_.append("comp");
//      header_params_.append("keep");
//      header_params_.append("maddr");
//      header_params_.append("oc");
//      header_params_.append("oc-algo");
//      header_params_.append("oc-seq");
//      header_params_.append("oc-validity");
//      header_params_.append("received");
//      header_params_.append("rport");
//      header_params_.append("sigcomp-id");
//      header_params_.append("ttl");
}

void HFVia::generate_values()
{
    std::ostringstream o;

    o << sent_proto_ << ' ' << sent_by_;

    if (header_params_.size())
        o << ";" << header_params_;

    values_ = o.str();
}

int HFVia::parse(std::string &msg, size_t &pos)
{
    bool read_head_param = false, run = true;
    std::string buffer, key;

    while (msg.at(pos) == ' ' || msg.at(pos) == '\t') pos++;

    while (run)
    {
        switch (msg.at(pos))
        {
            CASE_TOKEN
            case ':':
            case '/':
            {
                buffer += msg.at(pos++);
                break;
            }
            case ' ':
            {
                if (sent_proto_.empty())
                {
                    sent_proto_ = buffer;
                }
                else if (sent_by_.empty())
                {
                    sent_by_ = buffer;
                }

                buffer.clear();
            }
            case '\t':
            case '\r':
            {
                pos++;
                break;
            }
            case ';':
            {
                if (read_head_param)
                {
                    header_params_.append(key, buffer);
                    key.clear();
                }
                else
                {
                    if (sent_by_.empty())
                    {
                        sent_by_ = buffer;
                    }

                    read_head_param = true;
                }

                pos++;
                buffer.clear();
                break;
            }
            case '=':
            {
                key = buffer;

                pos++;
                buffer.clear();
                break;
            }
            case '\n':
            {
                if (read_head_param)
                {
                    header_params_.append(key, buffer);
                    key.clear();
                }
                else
                {
                    if (sent_by_.empty())
                    {
                        sent_by_ = buffer;
                    }
                }

                if (pos+1 >= msg.size()) { run = false; break; }
                do_if_is_alpha(msg.at(pos+1), run = false)

                pos++;
                buffer.clear();
                break;
            }
            default:
            {
                std::cerr << __PRETTY_FUNCTION__ << " Unexpected '" << msg.at(pos) << '(' << (int)msg.at(pos) << ')' << "' :" << bu
                pos++;
                buffer.clear();
            }
        }
    }
    return PROCEDURE_OK;
}
```

```
    HFContact::HFContact() : HFBase_1_("Contact", "m")
    {
//          header_params_.append("expires");
//          header_params_.append("mp");
//          header_params_.append("np");
//          header_params_.append("pub-gruu");
//          header_params_.append("q");
//          header_params_.append("rc");
//          header_params_.append("reg-id");
//          header_params_.append("temp-gruu");
//          header_params_.append("temp-gruu-cookie");
    }

    void HFRetryAfter::generate_values()
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
    }

    int HFRetryAfter::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }


    void HFAlertInfo::generate_values()
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
    }

    int HFAlertInfo::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }

    void HFAllowEvents::generate_values()
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
    }

    int HFAllowEvents::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }

    void HFBase_1_::generate_values()
    {
        char sym = ',';
        std::ostringstream o;

        cons_.cleanup_empty_uri();

        for (auto &it : cons_)
        {
            o << *it << sym;
        }

        values_ = o.str();
        remove_tail_symbol(sym);

        if (header_params_.size())
        {
            std::ostringstream p;
            p << ";" << header_params_;
            values_ += p.str();
        }
    }


    void HFTimestamp::generate_values()
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
    }

    int HFTimestamp::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }

    void HFUserAgent::generate_values()
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
    }

    int HFUserAgent::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }

    void HFAnswerMode::generate_values()
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
    }

    int HFAnswerMode::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }

    void HFPrivAnswerMode::generate_values()
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
    }

    int HFPrivAnswerMode::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }

    void HFAcceptContact::generate_values()
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
    }

    int HFAcceptContact::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }

    HFAuthorization::HFAuthorization() : HFBase_5_("Authorization")
    {
//        header_params_.append("algorithm");
//        header_params_.append("auts");
//        header_params_.append("cnonce");
//        header_params_.append("nc");
//        header_params_.append("nonce");
//        header_params_.append("opaque");
//        header_params_.append("qop");
//        header_params_.append("realm");
//      header_params_.append("response");
//      header_params_.append("uri");
//      header_params_.append("username");
    }

    HFCallInfo::HFCallInfo() : HFBase_1_("Call-Info", true)
    {
//      header_params_.append("m");
//      header_params_.append("purpose");
    }

    void HFCallInfo::generate_values()
    {
        char sym = ',';
        std::ostringstream o;

        cons_.cleanup_empty_uri();

        for (auto &it : cons_)
        {
            o << '<' << it->uri() << '>' << it->params() << sym;
        }

        values_ = o.str();

        remove_tail_symbol(sym);
    }

    int HFCallInfo::parse(std::string &msg, size_t &pos)
    {
        bool run = true, in_aquote = false, in_dquote = false;
        std::string buffer, key;

        while (msg.at(pos) == ' ' || msg.at(pos) == '\t') pos++;

        while (run)
        {
            switch (msg.at(pos))
            {
                case '"':
                {
                    in_dquote = !in_dquote;
                    buffer += msg.at(pos++);

                    if (!in_dquote)
                    {
                        add_name(buffer);
                        buffer.clear();
                    }

                    break;
                }
                CASE_TOKEN
                case '/':
                case '?':
                case ':':
                case '@':
                {
                    buffer += msg.at(pos++);
                    break;
                }
                case '<':
                {
                    in_aquote = true;
                    pos++;
                    buffer.clear();
                    break;
                }
```

```
case '>':
{
    in_aquote = false;

    if (buffer.size())
    {
        if (key.size())
        {
            add_param(key, buffer);
            key.clear();
        }
        else
        {
            add_uri(buffer);
        }
    }

    pos++;
    buffer.clear();
    break;
}
case ',':
{
    if (in_dquote)
    {
        buffer += msg.at(pos++);
        break;
    }

    if (buffer.size())
    {
        if (key.size())
        {
            add_param(key, buffer);
            key.clear();
        }
        else
        {
            add_uri(buffer);
        }
    }

    pos++;
    buffer.clear();
    break;
}
case ';':
{
    if (key.size())
    {
        add_param(key, buffer);
        key.clear();
    }
    else
    {
        if (buffer.size())
            add_uri(buffer);
    }

    pos++;
    buffer.clear();
    break;
}
case '=':
{
    key = buffer;

    buffer.clear();
```

```
                pos++;
                break;
            }
            case ' ':
            {
                if (in_dquote)
                {
                    buffer += msg.at(pos++);
                    break;
                }

                if (in_aquote || key.size())
                {
                    pos++;
                    break;
                }

                buffer.clear();
            }
            case '\t':
            case '\r':
            {
                pos++;
                break;
            }
            case '\n':
            {
                if (key.size())
                {
                    add_param(key, buffer);
                    key.clear();
                }
                else
                {
                    add_uri(buffer);
                }

                if (pos+1 >= msg.size()) { run = false; break; }
                do_if_is_alpha(msg.at(pos+1), run = false)

                pos++;
                buffer.clear();
                break;
            }
            default:
            {
                std::cerr << __PRETTY_FUNCTION__ << " Unexpected '" << msg.at(pos) << '(' << (int)msg.at(pos) << ')' << "': " << bu
                pos++;
                buffer.clear();
            }
        }
    }
    return PROCEDURE_OK;
}

HFEvent::HFEvent() : HeaderField("Event", "o")
{
//      header_params_.append("adaptive-min-rate");
//      header_params_.append("body");
//      header_params_.append("call-id");
//      header_params_.append("effective-by");
//      header_params_.append("from-tag");
//      header_params_.append("id");
//      header_params_.append("include-session-description");
//      header_params_.append("max-rate");
//      header_params_.append("min-rate");
//      header_params_.append("model");
//      header_params_.append("profile-type");
//      header_params_.append("shared");
//      header_params_.append("to-tag");
//      header_params_.append("vendor");
//      header_params_.append("version");
}

void HFEvent::generate_values()
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
}

int HFEvent::parse(std::string &msg, size_t &pos)
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
    return PROCEDURE_OK;
}

void HFInReplyTo::generate_values()
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
}

int HFInReplyTo::parse(std::string &msg, size_t &pos)
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
    return PROCEDURE_OK;
}

void HFJoin::generate_values()
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
}

int HFJoin::parse(std::string &msg, size_t &pos)
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
    return PROCEDURE_OK;
}

void HFPrivacy::generate_values()
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
}

int HFPrivacy::parse(std::string &msg, size_t &pos)
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
    return PROCEDURE_OK;
}

HFProxyAuthorization::HFProxyAuthorization() : HFBase_5_("Proxy-Authorization", true)
{
//      header_params_.append("algorithm");
//      header_params_.append("auts");
//      header_params_.append("cnonce");
//      header_params_.append("nc");
//      header_params_.append("nonce");
//      header_params_.append("opaque");
//      header_params_.append("qop");
//      header_params_.append("realm");
//      header_params_.append("response");
//      header_params_.append("uri");
//      header_params_.append("username");
}

void HFPOSPAuthToken::generate_values()
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
```

```
}

int HFPOSPAuthToken::parse(std::string &msg, size_t &pos)
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
    return PROCEDURE_OK;
}

void HFPAssertedIdentity::generate_values()
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
}

int HFPAssertedIdentity::parse(std::string &msg, size_t &pos)
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
    return PROCEDURE_OK;
}

void HFPPreferredIdentity::generate_values()
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
}

int HFPPreferredIdentity::parse(std::string &msg, size_t &pos)
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
    return PROCEDURE_OK;
}

void HFReason::generate_values()
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
}

int HFReason::parse(std::string &msg, size_t &pos)
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
    return PROCEDURE_OK;
}

void HFReferTo::generate_values()
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
}

int HFReferTo::parse(std::string &msg, size_t &pos)
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
    return PROCEDURE_OK;
}

void HFReferredBy::generate_values()
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
}

int HFReferredBy::parse(std::string &msg, size_t &pos)
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
    return PROCEDURE_OK;
}

void HFReplyTo::generate_values()
{
    std::cout << __PRETTY_FUNCTION__ << '\n';
}
```

```
    int HFReplyTo::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }

    void HFReplaces::generate_values()
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
    }

    int HFReplaces::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }

    void HFRejectContact::generate_values()
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
    }

    int HFRejectContact::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }

    void HFRequestDisposition::generate_values()
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
    }

    int HFRequestDisposition::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }

    void HFRack::generate_values()
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
    }

    int HFRack::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }

    void HFSessionExpires::generate_values()
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
    }

    int HFSessionExpires::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }

    HFSubscriptionState::HFSubscriptionState() : HeaderField("Subscription-State")
    {
//        header_params_.append("adaptive-min-rate");
//        header_params_.append("expires");
//        header_params_.append("max-rate");
//        header_params_.append("min-rate");
//        header_params_.append("reason");
//      header_params_.append("retry-after");
    }

    void HFSubscriptionState::generate_values()
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
    }

    int HFSubscriptionState::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }

    HFAuthenticationInfo::HFAuthenticationInfo() : HeaderField("Authentication-Info")
    {
//      header_params_.append("cnonce");
//      header_params_.append("nc");
//      header_params_.append("nextnonce");
//      header_params_.append("qop");
//      header_params_.append("rspauth");
    }

    void HFAuthenticationInfo::generate_values()
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
    }

    int HFAuthenticationInfo::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }

    void HFMinSE::generate_values()
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
    }

    int HFMinSE::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }

    HFProxyAuthenticate::HFProxyAuthenticate() : HFBase_4_("Proxy-Authenticate", true)
    {
//      header_params_.append("algorithm");
//      header_params_.append("domain");
//      header_params_.append("nonce");
//      header_params_.append("opaque");
//      header_params_.append("qop");
//      header_params_.append("realm");
//      header_params_.append("stale");
    }

    void HFWarning::generate_values()
    {
        char sym = ',';
        std::ostringstream o;

        for (auto &it : warn_vals_)
            o << it << sym;

        values_ = o.str();

        remove_tail_symbol(sym);
        std::stringstream p;
        p << header_params_;
        values_ += p.str();
    }

    int HFWarning::parse(std::string &msg, size_t &pos)
    {
        bool run = true, in_dquote = false;
        std::string buffer;
        size_t index = 0;

        while (msg.at(pos) == ' ' || msg.at(pos) == '\t') pos++;

        while (run)
        {
            switch (msg.at(pos))
            {
                CASE_TOKEN
                case '(':
                case ')':
                case ']':
                case '[':
                case '<':
                case '>':
                {
                    buffer += msg.at(pos++);
                    break;
                }
                case '\t':
                {
                    pos++;
                    buffer.clear();
                    break;
                }
                case ' ':
                {
                    if (in_dquote)
                    {
                        buffer += msg.at(pos++);
                        break;
                    }

                    if (buffer.size())
                    {
                        if (index >= warn_vals_.size())
                            warn_vals_.resize(warn_vals_.size()+1);

                        if (warn_vals_.at(index).code_.empty())
                            warn_vals_.at(index).code_ = buffer;
                    }

                    pos++;
                    buffer.clear();
                    break;
                }
                case ',':
                {
                    if (in_dquote)
                    {
                        buffer += msg.at(pos++);
                        break;
                    }

                    if (buffer.size())
                    {
                        if (index >= warn_vals_.size())
                            warn_vals_.resize(warn_vals_.size()+1);
                        if (warn_vals_.at(index).text_.empty())
                            warn_vals_.at(index).text_ = buffer;
                        index++;
                    }

                    pos++;
                    buffer.clear();
                    break;
                }
                case '"':
                {
                    in_dquote = !in_dquote;

                    if (!in_dquote)
                    {
                        if (buffer.size())
                        {
                            if (index >= warn_vals_.size())
                                warn_vals_.resize(warn_vals_.size()+1);

                            if (warn_vals_.at(index).text_.empty())
                                warn_vals_.at(index).text_ = buffer;
                            index++;
                        }
                    }

                    pos++;
                    buffer.clear();
                    break;
                }
                case '\r':
                {
                    pos++;
                    break;
                }
                case '\n':
                {
                    if (buffer.size())
                    {
                        if (index >= warn_vals_.size())
                            warn_vals_.resize(warn_vals_.size()+1);

                        if (warn_vals_.at(index).text_.empty())
                            warn_vals_.at(index).text_ = buffer;
                        index++;
                    }

                    if (pos+1 >= msg.size()) { run = false; break; }
                    do_if_is_alpha(msg.at(pos+1), run = false)

                    pos++;
                    buffer.clear();
                    break;
                }
                default:
                {
                    std::cerr << __PRETTY_FUNCTION__ << " Unexpected '" << msg.at(pos) << '(' << (int)msg.at(pos) << ')' << "': " << bu
                    pos++;
                    buffer.clear();
                }
            }
        }
        return PROCEDURE_OK;
    }

    HFWWWAuthenticate::HFWWWAuthenticate() : HFBase_5_("WWW-Authenticate", true)
    {
//      header_params_.append("algorithm");
//        header_params_.append("domain");
//        header_params_.append("nonce");
//        header_params_.append("opaque");
//        header_params_.append("qop");
//        header_params_.append("realm");
//        header_params_.append("stale");
    }

    void HFRSeq::generate_values()
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
    }

    int HFRSeq::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }

    void HFContentLanguage::generate_values()
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
    }

    int HFContentLanguage::parse(std::string &msg, size_t &pos)
    {
        std::cout << __PRETTY_FUNCTION__ << '\n';
        return PROCEDURE_OK;
    }

    void HFMIMEVersion::generate_values()
    {
        values_ = dotted_value_;
        std::stringstream p;
        p << header_params_;
        values_ += p.str();
    }

    int HFMIMEVersion::parse(std::string &msg, size_t &pos)
    {
        bool run = true;
        std::string buffer;

        while (msg.at(pos) == ' ' || msg.at(pos) == '\t') pos++;

        while (run)
        {
            switch (msg.at(pos))
            {
                CASE_DIGIT
                case '.':
                {
                    buffer += msg.at(pos++);
                    break;
                }
                case '\r':
                {
                    pos++;
                    break;
                }
                case '\n':
                {
                    if (dotted_value_.empty())
                        dotted_value_ = buffer;

                    run = false;

                    pos++;
            buffer.clear();
            break;
          }
          default:
          {
            std::cerr << __PRETTY_FUNCTION__ << " Unexpected '" << msg.at(pos) << '(' << (int)msg.at(pos) << ')' << "': " << bu
            pos++;
            buffer.clear();
          }
        }
      }
      return PROCEDURE_OK;
    }

    void HeaderFields::init_allowed_fields()
    {
      allowed_fields_["Call-ID"]              = HF_CALLID;
      allowed_fields_["CSeq"]                 = HF_CSEQ;
      allowed_fields_["From"]                 = HF_FROM;
      allowed_fields_["To"]                   = HF_TO;
      allowed_fields_["Via"]                  = HF_VIA;
      allowed_fields_["Alert-Info"]           = HF_ALERT_INFO;
      allowed_fields_["Allow-Events"]         = HF_ALLOW_EVENTS;
      allowed_fields_["Date"]                 = HF_DATE;
      allowed_fields_["Contact"]              = HF_CONTACT;
      allowed_fields_["Organization"]         = HF_ORGANIZATION;
      allowed_fields_["Record-Route"]         = HF_RECORD_ROUTE;
      allowed_fields_["Retry-After"]          = HF_RETRY_AFTER;
      allowed_fields_["Subject"]              = HF_SUBJECT;
      allowed_fields_["Supported"]            = HF_SUPPORTED;
      allowed_fields_["Timestamp"]            = HF_TIMESTAMP;
      allowed_fields_["User-Agent"]           = HF_USER_AGENT;
      allowed_fields_["Answer-Mode"]          = HF_ANSWER_MODE;
      allowed_fields_["Priv-Answer-Mode"]     = HF_PRIV_ANSWER_MODE;
      allowed_fields_["Accept"]               = HF_ACCEPT;
      allowed_fields_["Accept-Contact"]       = HF_ACCEPT_CONTACT;
      allowed_fields_["Accept-Encoding"]      = HF_ACCEPT_ENCODING;
      allowed_fields_["Accept-Language"]      = HF_ACCEPT_LANGUAGE;
      allowed_fields_["Authorization"]        = HF_AUTHORIZATION;
      allowed_fields_["Call-Info"]            = HF_CALL_INFO;
      allowed_fields_["Event"]                = HF_EVENT;
      allowed_fields_["In-Reply-To"]          = HF_IN_REPLY_TO;
      allowed_fields_["Join"]                 = HF_JOIN;
      allowed_fields_["Priority"]             = HF_PRIORITY;
      allowed_fields_["Privacy"]              = HF_PRIVACY;
      allowed_fields_["Proxy-Authorization"]  = HF_PROXY_AUTHORIZATION;
      allowed_fields_["Proxy-Require"]        = HF_PROXY_REQUIRE;
      allowed_fields_["P-OSP-AuthToken"]      = HF_P_OSP_AUTHTOKEN;
      allowed_fields_["PAsserted-Identity"]   = HF_PASSERTED_IDENTITY;
      allowed_fields_["PPreferred-Identity"]  = HF_PPREFERRED_IDENTITY;
      allowed_fields_["Max-Forwards"]         = HF_MAX_FORWARDS;
      allowed_fields_["Reason"]               = HF_REASON;
      allowed_fields_["Refer-To"]             = HF_REFER_TO;
      allowed_fields_["Referred-By"]          = HF_REFERRED_BY;
      allowed_fields_["Reply-To"]             = HF_REPLY_TO;
      allowed_fields_["Replaces"]             = HF_REPLACES;
      allowed_fields_["Reject-Contact"]       = HF_REJECT_CONTACT;
      allowed_fields_["Request-Disposition"]  = HF_REQUEST_DISPOSITION;
      allowed_fields_["Require"]              = HF_REQUIRE;
      allowed_fields_["Route"]                = HF_ROUTE;
      allowed_fields_["Rack"]                 = HF_RACK;
      allowed_fields_["Session-Expires"]      = HF_SESSION_EXPIRES;
      allowed_fields_["Subscription-State"]   = HF_SUBSCRIPTION_STATE;
      allowed_fields_["AuthenticationInfo"]   = HF_AUTHENTICATIONINFO;
      allowed_fields_["Error-Info"]           = HF_ERROR_INFO;
      allowed_fields_["Min-Expires"]          = HF_MIN_EXPIRES;
      allowed_fields_["Min-SE"]               = HF_MIN_SE;
      allowed_fields_["Proxy-Authenticate"]   = HF_PROXY_AUTHENTICATE;
```

```
        allowed_fields_["Server"]               = HF_SERVER;
        allowed_fields_["Unsupported"]          = HF_UNSUPPORTED;
        allowed_fields_["Warning"]              = HF_WARNING;
        allowed_fields_["WWW-Authenticate"]     = HF_WWW_AUTHENTICATE;
        allowed_fields_["RSeq"]                 = HF_RSEQ;
        allowed_fields_["Allow"]                = HF_ALLOW;
        allowed_fields_["Content-Encoding"]     = HF_CONTENT_ENCODING;
        allowed_fields_["Content-Length"]       = HF_CONTENT_LENGTH;
        allowed_fields_["Content-Disposition"]  = HF_CONTENT_DISPOSITION;
        allowed_fields_["Content-Language"]     = HF_CONTENT_LANGUAGE;
        allowed_fields_["Content-Type"]         = HF_CONTENT_TYPE;
        allowed_fields_["Expires"]              = HF_EXPIRES;
        allowed_fields_["MIME-Version"]         = HF_MIME_VERSION;
      }

    HeaderFields::HeaderFields()
    {
    }

    HeaderFields::~HeaderFields()
    {
    }

} // namespace EasySip
```

```
/*
 * src/timer.cpp
 */
#include "timer.h"

namespace EasySip
{
    bool operator== (struct itimerval &a, struct itimerval &b)
    {
        return timercmp(&a.it_interval, &b.it_interval, ==)
                && timercmp(&a.it_value, &b.it_value, ==);
    }

    bool operator!= (struct itimerval &a, struct itimerval &b)
    {
        return !(timercmp(&a.it_interval, &b.it_interval, ==)
                && timercmp(&a.it_value, &b.it_value, ==));
    }

    std::ostream& operator<< (std::ostream &o, struct timeval &a)
    {
        return o << "[" << a.tv_sec << ", " << a.tv_usec << "]";
    }

    std::ostream& operator<< (std::ostream &o, struct itimerval &a)
    {
        return o << a.it_value << " : " << a.it_interval;
    }

    std::ostream& operator<< (std::ostream &o, struct timespec &a)
    {
        return o << "[" << a.tv_sec << ", " << a.tv_nsec << "]";
    }

    std::ostream& operator<< (std::ostream &o, struct itimerspec &a)
    {
        return o << a.it_value << " : " << a.it_interval;
    }

    std::string Time::now()
    {
        time_t buf = time(0);
        std::string fmt("%a, %d %b %G %H:%M:%S GMT");
        char sbuf[30] = {0};

        strftime(sbuf, sizeof(sbuf), fmt.c_str(), gmtime(&buf));

        return std::string(sbuf);
//      return std::string(asctime(gmtime(&buf)));
    }
} // namespace EasySip
```

```
/*
 * src/dialog.cpp
 */
#include "dialog.h"


namespace EasySip
{
 Dialog::Dialog(Dialog &dia)
 {
  *this = dia;
 }

 Dialog::Dialog(RequestMessage &in_msg)
 : secure_flag_(false), confirmed_(false)
 {
  if (false /*TODO: sent over TLS && in_msg.req_line_->request_uri_ has sip URI */)
  {
   secure_flag(true);
  }

  if (in_msg.record_route_.size())
  {
   routes(in_msg.record_route_);
   std::reverse(routes().begin(), routes().end());
  }
  else
  {
   routes().clear();
  }

  if (in_msg.cseq_.size())
  {
   remote_seq(*in_msg.cseq_.last());
  }
//  local_seq_ = UNSET;
  if (in_msg.call_id_.size())
  {
   id().call_id(*in_msg.call_id_.last());
  }

  if (in_msg.to_.size())
  {
   id().local_tag(in_msg.to_.last()->tag());
   local_uri(in_msg.to_.last()->uri());
  }

  if (in_msg.from_.size())
  {
   id().remote_tag(in_msg.from_.last()->tag());
   remote_uri(in_msg.from_.last()->uri());
  }
 }

 Dialog::Dialog(ResponseMessage &in_msg)
 : secure_flag_(false), confirmed_(false)
 {
  if (false /*TODO: sent over TLS && in_msg.req_line_->request_uri_ has sip URI */)
  {
   secure_flag(true);
  }

  if (in_msg.record_route_.size())
  {
   routes(in_msg.record_route_);
   std::reverse(routes().begin(), routes().end());
  }
  else
  {
   routes().clear();
  }

  for (auto &it : in_msg.contact_)
  {
   remote_target().append(it->cons());
  }

//  remote_seq(UNSET);
  if (in_msg.cseq_.size())
  {
   local_seq(*in_msg.cseq_.last());
  }
  if (in_msg.call_id_.size())
  {
   id().call_id(*in_msg.call_id_.last());
  }
  if (in_msg.to_.size())
  {
   id().remote_tag(in_msg.to_.last()->tag());
   remote_uri(in_msg.to_.last()->uri());
  }
  if (in_msg.from_.size())
  {
   id().local_tag(in_msg.from_.last()->tag());
   local_uri(in_msg.from_.last()->uri());
  }
 }

 std::ostream& operator<< (std::ostream &o, Dialog &dia)
 {
  return o << dia.id()
   << "local_seq: " << dia.local_seq()
   << "remote_seq: " << dia.remote_seq()
   << "local_uri: " << dia.local_uri() << '\n'
   << "remote_uri: " << dia.remote_uri() << '\n'
   << dia.remote_target()
   << "secure_flag: " << dia.secure_flag() << '\n'
   << dia.routes()
   << "confirmation: " << (dia.is_confirmed() ? "true" : "false") << '\n';
 }

 Dialog* Dialogs::create_dialog()
 {
  append_item();
  std::cout << "dialogs size: [" << size() << "]\n";
  return last();
 }

 Dialog* Dialogs::create_dialog(Dialog &dialog)
 {
  append_item(dialog);
  std::cout << "dialogs size: [" << size() << "]\n";
  return last();
 }

 void Dialogs::cancel_dialog(DialogId val)
 {
  for (iterator it = begin(); it != end();)
  {
   if (val == (*it)->id())
   {
    erase(it);
    std::cout << "cancel dialog: \n[\n" << **it << "]\n";
    break;
   }
   else
   {
    it++;
   }
  }
  std::cout << "dialogs size: [" << size() << "]\n";
 }

 Dialog* Dialogs::get_dialog_by_id(DialogId &val)
 {
  for (iterator it = begin(); it != end(); it++)
  {
   if (val == (*it)->id())
   {
    return *it;
   }
  }

  return 0;
 }

 Dialog* Dialogs::operator[] (DialogId val)
 {
  return get_dialog_by_id(val);
 }


} // namespace EasySip
```

```
/*
 * src/Element/element.cpp
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#include "Element/element.h"

namespace EasySip
{
    Element::Element()
    : run_(true), stateful_(false)
    {
        HeaderFields::init_allowed_fields();
        init_allowed_methods();
        init_allowed_responses();
    }

    Element::~Element()
    {
    }

    void Element::init_allowed_methods()
    {
        allowed_methods_.insert(METHOD_INVITE);
        allowed_methods_.insert(METHOD_CANCEL);
        allowed_methods_.insert(METHOD_ACK);
        allowed_methods_.insert(METHOD_BYE);
        allowed_methods_.insert(METHOD_REGISTER);
        allowed_methods_.insert(METHOD_OPTIONS);
        allowed_methods_.insert(METHOD_SUBSCRIBE);
        allowed_methods_.insert(METHOD_NOTIFY);
        allowed_methods_.insert(METHOD_MESSAGE);
        allowed_methods_.insert(METHOD_INFO);
        allowed_methods_.insert(METHOD_UPDATE);
        allowed_methods_.insert(METHOD_REFER);
        allowed_methods_.insert(METHOD_PRACK);
    }

    void Element::init_allowed_responses()
    {
        allowed_responses_.insert(SIP_RESPONSE_TRYING);
        allowed_responses_.insert(SIP_RESPONSE_RINGING);
        allowed_responses_.insert(SIP_RESPONSE_FORWARDING);
        allowed_responses_.insert(SIP_RESPONSE_QUEUED);
        allowed_responses_.insert(SIP_RESPONSE_SESSION_PROGRESS);
        allowed_responses_.insert(SIP_RESPONSE_SUCCESSFUL);
        allowed_responses_.insert(SIP_RESPONSE_ACCEPTED);
        allowed_responses_.insert(SIP_RESPONSE_MULTI_CHOICES);
        allowed_responses_.insert(SIP_RESPONSE_MOVE_PERM);
        allowed_responses_.insert(SIP_RESPONSE_MOVE_TEMP);
        allowed_responses_.insert(SIP_RESPONSE_USE_PROXY);
        allowed_responses_.insert(SIP_RESPONSE_ALTER_SERVICE);
        allowed_responses_.insert(SIP_RESPONSE_BAD_REQUEST);
        allowed_responses_.insert(SIP_RESPONSE_UNAUTHORIZED);
        allowed_responses_.insert(SIP_RESPONSE_REQUIRE_PAYMENT);
        allowed_responses_.insert(SIP_RESPONSE_FORBIDDEN);
        allowed_responses_.insert(SIP_RESPONSE_NOT_FOUND);
        allowed_responses_.insert(SIP_RESPONSE_METHOD_NOT_ALLOWED);
        allowed_responses_.insert(SIP_RESPONSE_NOT_ACCEPTABLE);
        allowed_responses_.insert(SIP_RESPONSE_REQUIRE_PROXY_AUTHENTICATION);
        allowed_responses_.insert(SIP_RESPONSE_REQUIRE_REQUEST_TIMEOUT);
        allowed_responses_.insert(SIP_RESPONSE_RESOURCE_NOT_AVAIL);
        allowed_responses_.insert(SIP_RESPONSE_REQUEST_ENTITY_TOO_LARGE);
        allowed_responses_.insert(SIP_RESPONSE_REQUEST_URI_TOO_LONG);
        allowed_responses_.insert(SIP_RESPONSE_UNSUPPORTED_MEDIA_TYPE);
        allowed_responses_.insert(SIP_RESPONSE_UNSUPPORTED_URI_SCHEME);
        allowed_responses_.insert(SIP_RESPONSE_BAD_EXTENSION);
        allowed_responses_.insert(SIP_RESPONSE_REQUIRE_EXTENSION);
        allowed_responses_.insert(SIP_RESPONSE_INTERVAL_TOO_BRIEF);
        allowed_responses_.insert(SIP_RESPONSE_UNAVAIL_TEMP);
        allowed_responses_.insert(SIP_RESPONSE_CALL_OR_TRANSACTION_NOT_EXIST);
        allowed_responses_.insert(SIP_RESPONSE_LOOP_DETECTED);
        allowed_responses_.insert(SIP_RESPONSE_TOO_MANY_HOPS);
        allowed_responses_.insert(SIP_RESPONSE_ADDRESS_INCOMPLETE);
        allowed_responses_.insert(SIP_RESPONSE_AMBIGUOUS_URI);
        allowed_responses_.insert(SIP_RESPONSE_BUSY);
        allowed_responses_.insert(SIP_RESPONSE_REQUEST_TERMINATED);
        allowed_responses_.insert(SIP_RESPONSE_NOT_ACCEPTABLE_HERE);
        allowed_responses_.insert(SIP_RESPONSE_REQUEST_PENDING);
        allowed_responses_.insert(SIP_RESPONSE_UNDECIPHERABLE);
        allowed_responses_.insert(SIP_RESPONSE_SERVER_INTERNAL_ERROR);
        allowed_responses_.insert(SIP_RESPONSE_FUNC_NOT_IMPLEMENTED);
        allowed_responses_.insert(SIP_RESPONSE_BAD_GATEWAY);
        allowed_responses_.insert(SIP_RESPONSE_SERVICE_UNAVAIL);
        allowed_responses_.insert(SIP_RESPONSE_SERVICE_TIMEOUT);
        allowed_responses_.insert(SIP_RESPONSE_UNSUPPORTED_VERSION);
        allowed_responses_.insert(SIP_RESPONSE_MESSAGE_TOO_LARGE);
        allowed_responses_.insert(SIP_RESPONSE_GLOBAL_BUSY);
        allowed_responses_.insert(SIP_RESPONSE_CALLEE_DECLINE);
        allowed_responses_.insert(SIP_RESPONSE_GLOBAL_NOT_EXIST);
        allowed_responses_.insert(SIP_RESPONSE_GLOBAL_NOT_ACCEPTABLE);
    }

    void Element::send_msg(RequestMessage &msg)
    {
        udp_.send_buffer(msg.create().Msg());
    }

    void Element::send_msg(ResponseMessage &msg)
    {
        udp_.send_buffer(msg.create().Msg());
    }

    void Element::echo(RequestMessage &in_msg)
    {
        ResponseMessage rep(in_msg);

        rep.SipVersion(SIP_VERSION_2_0);
        rep.ResponseCode(SIP_RESPONSE_SUCCESSFUL);

        rep.append_userdata("Echo from Dr.Who");
        rep.add_content_length();

        send_msg(rep);
    }

    int Element::on_receive_message(std::string &msg)
    {
        int ret;

        if (METHOD_INVITE.code() <= (ret = Message::get_method_from_buffer(allowed_methods_, msg)))
        {
            return on_receive_req(msg, ret);
        }

        if (SIP_RESPONSE_TRYING.code() <= (ret = Message::get_response_code_from_buffer(allowed_responses_, msg)))
        {
            return on_receive_resp(msg, ret);
        }
        //TODO throw exception ??
        return PROCEDURE_ERROR;
    }

    void Element::simple_response(const RespCode &rc, RequestMessage &in_msg)
    {
        ResponseMessage rep(in_msg);
        rep.SipVersion(SIP_VERSION_2_0);
        rep.ResponseCode(rc);

        send_msg(rep);
    }

    int Element::on_receive_req(std::string &msg, const int code)
    {
        int ret = PROCEDURE_OK;
        RequestMessage in_msg(msg);

        if (false /* TODO: pending a request on demand*/)
        {
            ResponseMessage rep(in_msg);
            rep.SipVersion(SIP_VERSION_2_0);
            rep.ResponseCode(SIP_RESPONSE_REQUEST_TERMINATED);

            send_msg(rep);
            return MESSAGE_PROCESSED;
        }

        if (SIP_RESPONSE_UNSUPPORTED_URI_SCHEME.code() == (ret = in_msg.parse()))
        {
            simple_response(SIP_RESPONSE_UNSUPPORTED_URI_SCHEME, in_msg);
            return MESSAGE_PROCESSED;
        }

        if (in_msg.max_forwards_.size())
        {
            if (in_msg.max_forwards_.last()->is_zero_forward())
            {
                if (METHOD_ID_OPTIONS != code)
                {
                    simple_response(SIP_RESPONSE_TOO_MANY_HOPS, in_msg);
                    return MESSAGE_PROCESSED;
                }
            }
        }

        // TODO: loop detection
        if (false)
        {
            simple_response(SIP_RESPONSE_LOOP_DETECTED, in_msg);
            return MESSAGE_PROCESSED;
        }

        if (in_msg.proxy_require_.size())
        {
            std::vector<std::string> tags = in_msg.proxy_require_.last()->misunderstand_tags();

            if (tags.size())
            {
                ResponseMessage rep(in_msg);
                rep.SipVersion(SIP_VERSION_2_0);
                rep.ResponseCode(SIP_RESPONSE_BAD_EXTENSION);

                rep.add_unsupported()
                ->add_value(tags);

                send_msg(rep);
                return MESSAGE_PROCESSED;
            }
        }

        if (in_msg.proxy_authorization_.size())
        {
            //TODO: inspection NOTE: 96/269
        }
```

```
if (METHOD_ID_INVITE != code
&& METHOD_ID_REGISTER != code)
{
   Dialog dialog(in_msg);
   if (dialog_preprocess<RequestMessage>(dialog, in_msg))
      return PROCEDURE_OK;
}

switch (code)
{
   case METHOD_ID_INVITE:
   {
      return on_invite_request(in_msg);
   }
   case METHOD_ID_REGISTER:
   {
      return on_register_request(in_msg);
   }
   case METHOD_ID_CANCEL:
   {
      return on_cancel_request(in_msg);
   }
   case METHOD_ID_ACK:
   {
      return on_ack_request(in_msg);
   }
   case METHOD_ID_BYE:
   {
      return on_bye_request(in_msg);
   }
   case METHOD_ID_OPTIONS:
   {
      return on_options_request(in_msg);
   }
   case METHOD_ID_SUBSCRIBE:
   {
      return on_subscribe_request(in_msg);
   }
   case METHOD_ID_NOTIFY:
   {
      return on_notify_request(in_msg);
   }
   case METHOD_ID_MESSAGE:
   {
      return on_message_request(in_msg);
   }
   case METHOD_ID_INFO:
   {
      return on_info_request(in_msg);
   }
   case METHOD_ID_UPDATE:
   {
      return on_update_request(in_msg);
   }
   case METHOD_ID_REFER:
   {
      return on_refer_request(in_msg);
   }
   case METHOD_ID_PRACK:
   {
      return on_prack_request(in_msg);
   }
   default:
   {
      std::cerr << "Unexpected request: " << code << '\n';
   }
}
```

```
        return ret;
    }

    int Element::on_receive_resp(std::string &msg, const int code)
    {
        ResponseMessage in_msg(msg);
        in_msg.parse();

        Dialog dialog(in_msg);

        if (in_msg.is_resp2invite())
        {
            if (in_msg.is_1xx_resp())
            {
                dialogs_.create_dialog(dialog);
            }
            else if (in_msg.is_2xx_resp())
            {
                dialogs_[dialog.id()]->is_confirmed(true);

                AckMessage ack(in_msg);
                ack.SipVersion(SIP_VERSION_2_0);
                ack.RequestURI(udp_.Addr());

                send_msg(ack);
            }
            else
            {
                bye_request();
                // TODO: invite req failed feedback
//              dialogs_.cancel_dialog(dialog.id());
                std::cerr << "Unable to establish session due to \n[\n"
                        << in_msg << "]\n";
            }
        }
        else if (in_msg.is_resp2register())
        {
   switch (code)
   {
    default:; //TODO for each code
   }
  }
//      else if ((ret = dialog_preprocess<ResponseMessage>(dialog, in_msg)))
//      {
//          return ret;
//      }
        // TODO: else
        if (dialogs_[dialog.id()])
        {
            switch (code)
            {
                case 408:
                case 481:
                {
                    std::cout << "Receive response: " << code << ", cancelling dialog\n";
                    dialogs_.cancel_dialog(dialog.id());
                    return MESSAGE_PROCESSED;
                }
                default:;
            }
        }

        return PROCEDURE_OK;
    }

    int Element::fetch_msg()
    {
```

```
    if (0 > udp_.recv_buffer(0))
        return PROCEDURE_ERROR;

    std::cout << "peer: <" << udp_.Addr() << ":" << udp_.Port() << ">\n";
    std::string msg(udp_.Message());
    udp_.clear_msg();
    on_receive_message(msg);

    return PROCEDURE_OK;
}

int Element::start()
{
    try
    {
        while (run_)
        {
            if (0 > udp_.recv_buffer(0)) continue;
            // TODO: log peer
            std::cout << "peer: <" << udp_.Addr() << ":" << udp_.Port() << ">\n";
            std::string msg(udp_.Message());
            udp_.clear_msg();
            on_receive_message(msg);
        }
    }
    catch (std::exception e)
    {
        std::cout << "exception: " << e.what() << '\n';
        // TODO: log it
    }

    return PROCEDURE_OK;
}

int Element::invite_request()
{
    InviteMessage req;

    req.SipVersion(SIP_VERSION_2_0);
    req.RequestURI(udp_.Addr());

    req.add_from()
    ->add_name("zex")
    .add_uri("sip:zex@"+udp_.SelfAddr())
    .add_param("tag", "293!hsj@df");

    req.add_to()
    ->add_name("\"Big Boss\"")
    .add_uri("sip:bigboss@paris.agg.oo");

    req.add_cseq()
    ->cseq("1")
    .method(req.Method());

    req.add_via()
    ->add_proto(SIP_VERSION_2_0_UDP)
    .add_sentby(udp_.SelfAddr());

    req.add_call_id()
    ->id("sundo@1311bili");

    if (false /*TODO: is_sips(req.req_line_.request_uri_) */)
    {
        req.add_contact()
        ->add_uri("sips:zex@"+udp_.SelfAddr());
    }
    else
    {
            req.add_contact()
            ->add_uri("sip:zex@"+udp_.SelfAddr());
        }

        // TODO: check for re-invite
        if (!dialogs_.empty())
        {
        }

        send_msg(req);
        ivt_.state(T_FSM_CALLING);
//        msgq_.push(req.Msg());
        // TODO: 64*T1 start
        return PROCEDURE_OK;
    }

    int Element::register_request()
    {
        RegisterMessage req;

        req.SipVersion(SIP_VERSION_2_0);
        req.RequestURI("sip:nick@uuac.com");

        req.add_to()
        ->add_uri(udp_.SelfAddr())
        .add_name("ook");

        req.add_from()
        ->add_uri(udp_.SelfAddr())
        .add_name("ook");

        req.add_call_id()
        ->id("987kk");

        req.add_cseq()
        ->cseq("1")
        .method(req.Method());

        req.add_contact()
        ->add_uri("tel:+1-972-555-2222");
        //->add_uri(udp_.SelfAddr());

        req.add_route()
        ->add_uri("129.99.0.32");

        req.add_via()
        ->add_proto(SIP_VERSION_2_0_UDP)
        .add_sentby(udp_.SelfAddr());

        send_msg(req);

        return PROCEDURE_OK;
    }

    int Element::bye_request()
    {
        if (dialogs_.empty())
            return PROCEDURE_OK;

        ByeMessage req;
        req.SipVersion(SIP_VERSION_2_0);

        req.add_to()
        ->add_name("Big Boss\"")
        .add_uri(dialogs_.last()->remote_uri());

        if (dialogs_.last()->id().remote_tag().size())
            req.to_.last()->add_param("tag", dialogs_.last()->id().remote_tag());
        req.add_from()
        ->add_name("zex")
        .add_uri(dialogs_.last()->local_uri());

        if (dialogs_.last()->id().local_tag().size())
            req.from_.last()->add_param("tag", dialogs_.last()->id().local_tag());

        req.add_call_id()
        ->id(dialogs_.last()->id().call_id().id());

        std::string seq;

        if (!dialogs_.last()->local_seq().cseq().empty())
        {
            dialogs_.last()->local_seq().inc_seq();
            seq = dialogs_.last()->local_seq().cseq();
        }

        if (seq.empty())
        {
            seq = "1"; // TODO: choose a seq, 32bits
        }

        req.add_cseq()
        ->cseq(seq)
        .method(req.Method());

        if (dialogs_.last()->remote_target().size())
            req.RequestURI(dialogs_.last()->remote_target().last()->uri());

        if (dialogs_.last()->routes().size())
        {
            if (dialogs_.last()->routes().last()->cons_.last()->has_param("lr"))
            {
//                      if (dialogs_.last()->remote_target().size())
                req.RequestURI(dialogs_.last()->remote_target().last()->uri());

                req.add_route();

                if (dialogs_.last()->routes().size())
                {
                    req.route_.last()->cons_ = dialogs_.last()->routes().last()->cons_;
                }
            }
            else
            {
                req.RequestURI(dialogs_.last()->routes().last()->cons_.last()->uri());

                req.add_route();

                ContactList::iterator from = dialogs_.last()->routes().last()->cons_.begin();
                from++;

                req.route_.last()->cons_.append(from, dialogs_.last()->routes().last()->cons_.end());
                req.route_.last()->cons_.append(dialogs_.last()->remote_target());
            }
        }

        req.add_via()
        ->add_proto(SIP_VERSION_2_0_UDP)
        .add_sentby(udp_.SelfAddr());


        if (false /* TODO: is_sips(req.req_line_.request_uri_) */
        || false /* TODO: is_sips(req.req_line_.request_uri_) */)
        {
            req.add_contact()->add_uri("sips:utoc@ir.cx");
        }

        send_msg(req);
//        msgq_.push(req.Msg());
//-------------------------------------------------------------
        dialogs_.cancel_dialog(dialogs_.last()->id());

        return PROCEDURE_OK;
    }

    int Element::cancel_request()
    {
        CancelMessage req;
        req.SipVersion(SIP_VERSION_2_0);

        if (false /* TODO: 1xx resp not yet received */)
        {    /* wait until 1xx resp received then send */
            return PROCEDURE_ERROR;
        }

        send_msg(req);
        return PROCEDURE_OK;
    }

    int Element::update_request()
    {
        return PROCEDURE_OK;
    }

    int Element::info_request()
    {
        return PROCEDURE_OK;
    }

    int Element::ack_request()
    {
        AckMessage req;

        req.SipVersion(SIP_VERSION_2_0);

        if (dialogs_.size())
        {
            req.add_to()
            ->add_name("Big Boss\"")
            .add_uri(dialogs_.last()->remote_uri());

            if (dialogs_.last()->id().remote_tag().size())
                req.to_.last()->add_param("tag", dialogs_.last()->id().remote_tag());

            req.add_from()
            ->add_name("zex")
            .add_uri(dialogs_.last()->local_uri());

            if (dialogs_.last()->id().local_tag().size())
                req.from_.last()->add_param("tag", dialogs_.last()->id().local_tag());

            req.add_call_id()
            ->id(dialogs_.last()->id().call_id().id());

            std::string seq;

            if (!dialogs_.last()->local_seq().cseq().empty())
            {
                dialogs_.last()->local_seq().inc_seq();
                seq = dialogs_.last()->local_seq().cseq();
            }

            if (seq.empty())
            {
                seq = "1"; // TODO: choose a seq, 32bits
            }

            req.add_cseq()
            ->cseq(seq)
            .method(req.Method());

            if (dialogs_.last()->remote_target().size())
                req.RequestURI(dialogs_.last()->remote_target().last()->uri());

            if (dialogs_.last()->routes().size())
            {
                if (dialogs_.last()->routes().last()->cons_.last()->has_param("lr"))
                {
    //                  if (dialogs_.last()->remote_target().size())
                    req.RequestURI(dialogs_.last()->remote_target().last()->uri());

                    req.add_route();

                    if (dialogs_.last()->routes().size())
                    {
                        req.route_.last()->cons_ = dialogs_.last()->routes().last()->cons_;
                    }
                }
                else
                {
                    req.RequestURI(dialogs_.last()->routes().last()->cons_.last()->uri());

                    req.add_route();

                    ContactList::iterator from = dialogs_.last()->routes().last()->cons_.begin();
                    from++;

                    req.route_.last()->cons_.append(from, dialogs_.last()->routes().last()->cons_.end());
                    req.route_.last()->cons_.append(dialogs_.last()->remote_target());
                }
            }
        }

        req.add_via()
        ->add_proto(SIP_VERSION_2_0_UDP)
        .add_sentby(udp_.SelfAddr());


        if (false /*is_sips(req.req_line_.request_uri_) */
        || false /*is_sips(req.req_line_.request_uri_) */)
        {
            req.add_contact()->add_uri("sips:utoc@ir.cx");
        }

        send_msg(req);
        return PROCEDURE_OK;
    }

    int Element::message_request()
    {
        MessageMessage req;

        req.SipVersion(SIP_VERSION_2_0);
        req.RequestURI(udp_.Addr());

        if (dialogs_.size())
        {
            req.add_to()
            ->add_name("Big Boss\"")
            .add_uri(dialogs_.last()->remote_uri());
            if (dialogs_.last()->id().remote_tag().size())
                req.to_.last()->add_param("tag", dialogs_.last()->id().remote_tag());

            req.add_from()
            ->add_name("zex")
            .add_uri(dialogs_.last()->local_uri());

            if (dialogs_.last()->id().local_tag().size())
                req.from_.last()->add_param("tag", dialogs_.last()->id().local_tag());

            req.add_call_id()
            ->id(dialogs_.last()->id().call_id().id());

            std::string seq;

            if (!dialogs_.last()->local_seq().cseq().empty())
            {
                dialogs_.last()->local_seq().inc_seq();
                seq = dialogs_.last()->local_seq().cseq();
            }

            if (seq.empty())
            {
                seq = "1"; // TODO: choose a seq, 32bits
            }

            req.add_cseq()
            ->cseq(seq)
            .method(req.Method());

            if (dialogs_.last()->remote_target().size())
                req.RequestURI(dialogs_.last()->remote_target().last()->uri());

            if (dialogs_.last()->routes().size())
            {
                if (dialogs_.last()->routes().last()->cons_.last()->has_param("lr"))
                {
    //                  if (dialogs_.last()->remote_target().size())
                    req.RequestURI(dialogs_.last()->remote_target().last()->uri());

                    req.add_route();

                    if (dialogs_.last()->routes().size())
                    {
                        req.route_.last()->cons_ = dialogs_.last()->routes().last()->cons_;
                    }
                }
                else
                {
                    req.RequestURI(dialogs_.last()->routes().last()->cons_.last()->uri());

                    req.add_route();

                    ContactList::iterator from = dialogs_.last()->routes().last()->cons_.begin();
                    from++;

                    req.route_.last()->cons_.append(from, dialogs_.last()->routes().last()->cons_.end());
                    req.route_.last()->cons_.append(dialogs_.last()->remote_target());
                }
            }
        }
        else
        {
            req.RequestURI(udp_.Addr());

            req.add_to()
            ->add_name("Big Boss\"")
            .add_uri(udp_.Addr());
            req.add_from()
            ->add_name("zex")
            .add_uri(udp_.SelfAddr());

            req.add_cseq()
            ->cseq("1")
            .method(req.Method());

            req.add_call_id()
            ->id("54235jd"); // TODO: generate it

        }

        req.add_priority()
        ->add_value("emergency");

        req.add_content_type()
        ->type("application")
        .subtype("pkcs7-mime")
        .HeaderParam("smime-type", "enveloped-data")
        .HeaderParam("name", "smime.p7m");

        req.add_www_authenticate()
        ->add_value("Digest")
        .add_param("realm", "\"biloxi.com\"")
        .add_param("qop", "\"auth,auth-int\"")
        .add_param("nonce", "\"d928j8mms349q\"")
        .add_param("opaque", "\"5ccc8372dsvnlk\"");

        req.add_organization()
        ->add_value("ieee.org blenisa,asirel dlg,");

        req.add_subject()
        ->add_value("wekkwida asdfgnb adun38-vn kdi");

        req.add_date()
        ->add_value("Sat, 13 Nov 2010 23:29:00 GMT");

        req.add_accept_language()
        ->add_value("da")
        .add_param("q", "0.8")
        .add_value("en-gb")
        .add_param("q", "0.7")
        .add_value("en")
        .add_param("q", "0.1");

        req.add_accept_encoding()
        ->add_value("da")
        .add_param("q", "0.8")
        .add_value("en-gb")
        .add_param("q", "0.2");

        req.add_content_disposition()
        ->add_value("session")
        .HeaderParam("handling", "optional");

        req.add_via()
        ->add_proto(SIP_VERSION_2_0_UDP)
        .add_sentby(udp_.SelfAddr());

        if (false /*is_sips(req.req_line_.request_uri_) */
        || false /*is_sips(req.req_line_.request_uri_) */)
        {
            req.add_contact()->add_uri("sips:utoc@ir.cx");
        }

        req.append_userdata("bigo digo reading");
```

```
    req.add_content_length();

    send_msg(req);

    return PROCEDURE_OK;
}

int Element::subscribe_request()
{
    SubscribeMessage req;

    req.SipVersion(SIP_VERSION_2_0);
    req.RequestURI(udp_.Addr());

    send_msg(req);
    return PROCEDURE_OK;
}

int Element::notify_request()
{
    NotifyMessage req;

    req.SipVersion(SIP_VERSION_2_0);
    req.RequestURI(udp_.Addr());

    send_msg(req);
    return PROCEDURE_OK;
}

int Element::refer_request()
{
    ReferMessage req;

    req.SipVersion(SIP_VERSION_2_0);
    req.RequestURI(udp_.Addr());

    send_msg(req);

    return PROCEDURE_OK;
}

int Element::options_request()
{
    OptionsMessage req;

    req.SipVersion(SIP_VERSION_2_0);

    if (dialogs_.size())
    {
        req.add_to()
        ->add_name("Big Boss\"")
        .add_uri(dialogs_.last()->remote_uri());

        if (dialogs_.last()->id().remote_tag().size())
            req.to_.last()->add_param("tag", dialogs_.last()->id().remote_tag());

        req.add_from()
        ->add_name("zex")
        .add_uri(dialogs_.last()->local_uri());

        if (dialogs_.last()->id().local_tag().size())
            req.from_.last()->add_param("tag", dialogs_.last()->id().local_tag());

        req.add_call_id()
        ->id(dialogs_.last()->id().call_id().id());

        std::string seq;
        if (!dialogs_.last()->local_seq().cseq().empty())
        {
            dialogs_.last()->local_seq().inc_seq();
            seq = dialogs_.last()->local_seq().cseq();
        }

        if (seq.empty())
        {
            seq = "1"; // TODO: choose a seq, 32bits
        }

        req.add_cseq()
        ->cseq(seq)
        .method(req.Method());

        if (dialogs_.last()->remote_target().size())
            req.RequestURI(dialogs_.last()->remote_target().last()->uri());

        if (dialogs_.last()->routes().size())
        {
            if (dialogs_.last()->routes().last()->cons_.last()->has_param("lr"))
            {
//                  if (dialogs_.last()->remote_target().size())
                req.RequestURI(dialogs_.last()->remote_target().last()->uri());

                req.add_route();

                if (dialogs_.last()->routes().size())
                {
                    req.route_.last()->cons_ = dialogs_.last()->routes().last()->cons_;
                }
            }
            else
            {
                req.RequestURI(dialogs_.last()->routes().last()->cons_.last()->uri());

                req.add_route();

                ContactList::iterator from = dialogs_.last()->routes().last()->cons_.begin();
                from++;

                req.route_.last()->cons_.append(from, dialogs_.last()->routes().last()->cons_.end());
                req.route_.last()->cons_.append(dialogs_.last()->remote_target());
            }
        }
    }
    else
    {
        req.RequestURI(udp_.Addr());

        req.add_to()
        ->add_name("Big Boss\"")
        .add_uri(udp_.Addr());

        req.add_from()
        ->add_name("zex")
        .add_uri(udp_.SelfAddr());

        req.add_cseq()
        ->cseq("1")
        .method(req.Method());

        req.add_call_id()
        ->id("54235jd"); // TODO: generate it
    }

    req.add_via()
    ->add_proto(SIP_VERSION_2_0_UDP)
```

```
        .add_sentby(udp_.SelfAddr());

        if (false /*is_sips(req.req_line_.request_uri_) */
        || false /*is_sips(req.req_line_.request_uri_) */)
        {
            req.add_contact()->add_uri("sips:utoc@ir.cx");
        }

        send_msg(req);
//      msgq_.push(req.Msg());
//-------------------------------------------------------------

        return PROCEDURE_OK;
    }

    int Element::prack_request()
    {
        PrackMessage req;

        req.SipVersion(SIP_VERSION_2_0);
        req.RequestURI(udp_.Addr());

        send_msg(req);
        return PROCEDURE_OK;
    }

    int Element::on_invite_request(RequestMessage &in_msg)
    {
        ResponseMessage rep(in_msg);
        rep.SipVersion(SIP_VERSION_2_0);

        Dialog dialog(in_msg);

        // check for a re-invite request
        if (dialogs_[dialog.id()] && dialogs_[dialog.id()]->is_confirmed())
        {
            // TODO: update dialog
        }


        dialogs_.create_dialog(dialog);

        rep.add_contact()
        ->add_uri("sip:ag@"+udp_.Addr());

        if (in_msg.record_route_.size())
            rep.record_route_ = in_msg.record_route_;

        std::cout << "----------\n" << *dialogs_.last() << "-----------\n";

        rep.ResponseCode(SIP_RESPONSE_RINGING);

        send_msg(rep);
        dialogs_[dialog.id()]->still_ringing(true);

        // TODO: timeout here

        // dummy --------------->
        int i = 7;
        PROGRESS_WITH_FEEDBACK("ringing", i--, sleep(0.5); send_msg(rep))
        // dummy ---------------|

        if (false /* TODO: need redirect */)
        {
            rep.ResponseCode(SIP_RESPONSE_MULTI_CHOICES);
//          rep.ResponseCode(SIP_RESPONSE_MOVE_PERM);
//          rep.ResponseCode(SIP_RESPONSE_MOVE_TEMP);
            send_msg(rep);
            // TODO: start redirect

            return PROCEDURE_OK;
        }

        if (false /* TODO: get reject signal */)
        {
            if (false /* TODO: no one, really, will take this */)
                rep.ResponseCode(SIP_RESPONSE_GLOBAL_BUSY);
            else
                rep.ResponseCode(SIP_RESPONSE_BUSY);

            send_msg(rep);

            return PROCEDURE_OK;
        }

        rep.ResponseCode(SIP_RESPONSE_SUCCESSFUL);

        rep.add_allow();

        for (auto &it : allowed_methods_)
            rep.allow_.last()->add_value(it.name());


        rep.add_supported()
        ->add_value("100rel");

        send_msg(rep);

        // TODO: timeout here for ACK

        return PROCEDURE_OK;
    }

    int Element::on_register_request(RequestMessage &in_msg)
    {
        ResponseMessage rep(in_msg);
        rep.SipVersion(SIP_VERSION_2_0);
        rep.ResponseCode(SIP_RESPONSE_SUCCESSFUL);

        /*
         * TODO: Expires <= 2^32-1
         *     if Expires is illegal, then use 3600
         */

        /*
         * NOTE: A UA SHOULD NOT refresh bindings set up by
         * other UAs.
         * TODO: add bindings for AOR, check preference priority by `q`
         *  sip:xxxxxx
         *  tel:xxxxx
         *  mailto:xxxxx
         */

        /*
         * TODO: add current bindings list to rep
         */
        rep.add_date()
        ->add_value(Time::now());
//      ->add_value("Sat, 13 Nov 2014 23:29:00 GMT");
        send_msg(rep);

        return PROCEDURE_OK;
    }

    int Element::on_bye_request(RequestMessage &in_msg)
```

```
    {
        ResponseMessage rep(in_msg);

        Dialog dialog(in_msg);

//      if (in_msg.record_route_.size())
//          rep.record_route_ = in_msg.record_route_;

        dialogs_.cancel_dialog(dialog.id());

        return PROCEDURE_OK;
    }

    int Element::on_cancel_request(RequestMessage &in_msg)
    {
        Dialog dialog(in_msg);

        if (dialogs_[dialog.id()])
        {
            if (dialogs_[dialog.id()]->still_ringing())
            {
                // TODO: cancel it
            }
            else
            {
                ResponseMessage rep(in_msg);

                rep.SipVersion(SIP_VERSION_2_0);
                rep.ResponseCode(SIP_RESPONSE_REQUEST_TERMINATED);

                send_msg(rep);
            }
        }

        return PROCEDURE_OK;
    }

    int Element::on_ack_request(RequestMessage &in_msg)
    {
        Dialog dialog(in_msg);

        if (dialogs_[dialog.id()])
        {
            dialogs_[dialog.id()]->is_confirmed(true);
        }

        return PROCEDURE_OK;
    }

    int Element::on_options_request(RequestMessage &in_msg)
    {
        ResponseMessage rep(in_msg);

        rep.SipVersion(SIP_VERSION_2_0);
        rep.ResponseCode(SIP_RESPONSE_SUCCESSFUL);

        rep.add_accept()
        ->add_value("text", "plain")
        .add_value("text", "html")
        .add_value("application/sdp")
        .add_param("level", "1")
        .add_value("multipart/sdp");

        rep.add_allow();

        for (auto &it : allowed_methods_)
            rep.allow_.last()->add_value(it.name());
//        rep.add_error_info()
//        ->add_uri("<sip:mary238@4usnmn4.s49s.lsdj.org>")
//        .add_uri("<sip:yem.kkk.ei3m.com>");

        send_msg(rep);

        return PROCEDURE_OK;
    }

    int Element::on_subscribe_request(RequestMessage &in_msg)
    {
        echo(in_msg);
        return PROCEDURE_OK;
    }

    int Element::on_notify_request(RequestMessage &in_msg)
    {
        echo(in_msg);
        return PROCEDURE_OK;
    }

    int Element::on_info_request(RequestMessage &in_msg)
    {
        echo(in_msg);
        return PROCEDURE_OK;
    }

    int Element::on_update_request(RequestMessage &in_msg)
    {
        echo(in_msg);
        return PROCEDURE_OK;
    }

    int Element::on_refer_request(RequestMessage &in_msg)
    {
        echo(in_msg);
        return PROCEDURE_OK;
    }

    int Element::on_message_request(RequestMessage &in_msg)
    {
        echo(in_msg);
        return PROCEDURE_OK;
    }

    int Element::on_prack_request(RequestMessage &in_msg)
    {
        echo(in_msg);
        return PROCEDURE_OK;
    }

} // namespace EasySip
```

```
/*
 * src/Element/registar.cpp
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#include "Element/registar.h"

namespace EasySip
{
 Registar::Registar()
 {
  // TODO: configurable
  udp_.SelfAddr(Socket::get_ip_addr());
  udp_.SelfPort(5163);
  udp_.setup_server();
 }

// int Registar::invite_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Registar::register_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Registar::bye_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Registar::cancel_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Registar::update_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Registar::info_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Registar::ack_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Registar::message_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Registar::subscribe_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Registar::notify_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Registar::refer_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Registar::options_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Registar::prack_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Registar::on_invite_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  ResponseMessage rep(SIP_RESPONSE_TRYING);
//  udp_.send_buffer(rep.Msg());
//
//  InviteMethod invite(in_msg);
//  invite.parse();
//
//  return PROCEDURE_OK;
// }
//
 int Registar::on_register_request(RequestMessage &in_msg)
 {
  in_msg.parse();

  ResponseMessage rep(in_msg);
  rep.SipVersion(SIP_VERSION_2_0);

  // TODO: looking for server, determine whether proxy (Request-URI)

  // TODO: Check HFRequire for extensions

  // TODO: authenticate the UAC, if no auth-mechanism available , check HFFrom address

  // TODO: check if the authenticated user is authorized to modfy registrations for AOR.
  //  check the database where map user names to a list of AOR.
  //  if not authorized, reply with 403 response code and quit
  rep.ResponseCode(SIP_RESPONSE_FORBIDDEN);
  udp_.send_buffer(rep.create().Msg());
  return PROCEDURE_OK;

  // TODO: get AOR from HFTo.
  //  if AOR not valid for domain in Request-URI, reply with 404 response code and quit
  rep.ResponseCode(SIP_RESPONSE_NOT_FOUND);
  udp_.send_buffer(rep.create().Msg());
  return PROCEDURE_OK;

  // check HFContact
  if (in_msg.contact_.size())
  {
   if (1 < in_msg.contact_.size())
   {
    rep.ResponseCode(SIP_RESPONSE_BAD_REQUEST);
    udp_.send_buffer(rep.create().Msg());
    return PROCEDURE_OK;
   }

   for (auto &it : in_msg.contact_.at(0)->cons_)
   {
    if (it->uri() == "*")
    {
     if (in_msg.expires_.size() && in_msg.expires_.at(0)->digit_value_ != "0")
     {
      rep.ResponseCode(SIP_RESPONSE_BAD_REQUEST);
      udp_.send_buffer(rep.create().Msg());
      return PROCEDURE_OK;
     }
    }
   }

  // TODO: check HFCallId, whether agrees with each binding stored
  //  if not, remove the binding
  //  else
  //   if the in_msg.cseq_ > binding.cseq_
  //   else abort update, request failed

   int seconds;

   std::string expire = in_msg.contact_.at(0)->header_params_.get_value_by_name("expires");

   if (expire.empty())
   {
    if (in_msg.expires_.size())
    {
     expire = in_msg.expires_.at(0)->expire();
    }
    else
    {
     // TODO: expire = local expireation
    }
   }

   std::istringstream is(expire);
   is >> seconds;

   if (seconds > 0 && seconds < ONE_HOUR/* TODO && expire < local-min-registrar-timeout */)
   {
    rep.ResponseCode(SIP_RESPONSE_INTERVAL_TOO_BRIEF);
    rep.add_min_expires()->add_value("45");/* TODO: min-expire value*/
    udp_.send_buffer(rep.create().Msg());
    return PROCEDURE_OK;
   }
  }

  rep.ResponseCode(SIP_RESPONSE_SUCCESSFUL);
  // TODO: append HFContact in current bindings with expires param
  //  append HFDate
  udp_.send_buffer(rep.create().Msg());
  return PROCEDURE_OK;
 }
//
// int Registar::on_bye_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Registar::on_cancel_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Registar::on_ack_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Registar::on_options_request(RequestMessage &in_msg)
// {
```

```
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Registar::on_subscribe_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Registar::on_notify_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Registar::on_info_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Registar::on_update_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Registar::on_refer_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Registar::on_message_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Registar::on_prack_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Registar::on_response(Message &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Registar::on_rx_req_exception(RequestMessage &in_msg)
// {
//  // ---------------------------------------------
//  ResponseMessage resp_msg = in_msg;
//
//  resp_msg.RespStatus(SIP_RESPONSE_METHOD_NOT_ALLOWED);
//
//  resp_msg.allow_.append_field();
//
//  for (MethodMapList::iterator it = allowed_methods_.begin(); it != allowed_methods_.end(); it++)
//   resp_msg.allow_.append_value(it->Name());
//
//  // ---------------------------------------------
//
//  return PROCEDURE_OK;
// }
```

```
//
} // namespace EasySip
```

```
/*
 * src/Element/uaclient.cpp
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#include "Element/uaclient.h"

namespace EasySip
{
 UAClient::UAClient()
 {
  udp_.SelfAddr(Socket::get_ip_addr());
  udp_.SelfPort(2039);
  udp_.setup_server();
  udp_.Addr(Socket::get_ip_addr());
  udp_.Port(1971);
//  udp_.NeedBind(false);
 }

} // namespace EasySip
```

```
/*
 * src/Element/proxy.cpp
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#include "Element/proxy.h"

namespace EasySip
{
 Proxy::Proxy()
 {
  // TODO: configurable
  udp_.SelfAddr(Socket::get_ip_addr());
  udp_.SelfPort(7831);
  udp_.setup_server();
 }

// int Proxy::invite_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Proxy::register_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Proxy::bye_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Proxy::cancel_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Proxy::update_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Proxy::info_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Proxy::ack_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Proxy::message_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Proxy::subscribe_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Proxy::notify_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Proxy::refer_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Proxy::options_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Proxy::prack_request()
// {
//  return PROCEDURE_OK;
// }
//
// int Proxy::on_invite_request(RequestMessage &in_msg)
// {
//  return PROCEDURE_OK;
// }
//
// int Proxy::on_register_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Proxy::on_bye_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Proxy::on_cancel_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Proxy::on_ack_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Proxy::on_options_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Proxy::on_subscribe_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Proxy::on_notify_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Proxy::on_info_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Proxy::on_update_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Proxy::on_refer_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Proxy::on_message_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Proxy::on_prack_request(RequestMessage &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
//
// int Proxy::on_response(Message &in_msg)
// {
//  echo(in_msg);
//  return PROCEDURE_OK;
// }
} // namespace EasySip
```

```
/*
 * src/Element/uaserver.cpp
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#include "Element/uaserver.h"

namespace EasySip
{
 UAServer::UAServer()
 {
  // TODO: configurable
  udp_.SelfAddr(Socket::get_ip_addr());
  udp_.SelfPort(1971);
  udp_.setup_server();
 }

} // namespace EasySip
```

```
/*
 * src/socket.cpp
 *
 * Author: Zex <top_zlynch@yahoo.com>
 */
#include "socket.h"
#include "buffer.h"

// port reuse
//unsigned int yes = 1;
//setsockopt(socket, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes));

namespace EasySip
{
    std::string Socket::get_ip_addr()
    {
        std::string ret;
        struct ifaddrs *ifaddrs = NULL, *ifaddr = NULL;
        void *tmpAddrPtr = NULL;
        int prot, len;

        getifaddrs(&ifaddrs);

        for (ifaddr = ifaddrs; ifaddr; ifaddr = ifaddr->ifa_next)
        {
            if (!ifaddr->ifa_addr)
            {
                continue;
            }

            if (ifaddr->ifa_addr->sa_family == AF_INET)
            {
                tmpAddrPtr = &((struct sockaddr_in *)ifaddr->ifa_addr)->sin_addr;
                len = INET_ADDRSTRLEN;
                prot = AF_INET;
            }
            else if (ifaddr->ifa_addr->sa_family == AF_INET6)
            {
                tmpAddrPtr = &((struct sockaddr_in6 *)ifaddr->ifa_addr)->sin6_addr;
                len = INET6_ADDRSTRLEN;
                prot = AF_INET6;
                break;
            }
            else
            {
                continue;
            }


            Buffer addr_buf(len);
            inet_ntop(prot, tmpAddrPtr, addr_buf.data(), addr_buf.len());
            std::cout << "IF: " << ifaddr->ifa_name << " IP: " << addr_buf.data() << '\n';
            ret = addr_buf.data();
        }

        if (ifaddrs)
        {
            freeifaddrs(ifaddrs);
        }

        return ret;
    }

    int Socket::set_timeout(int sec)
    {
        int ret;

        struct timeval tv;
        tv.tv_sec = sec;
        tv.tv_usec = 0;

        if (0 > (ret = setsockopt(sk_, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv))))
                std::cerr << "socket: " << strerror(errno) << '\n';

        return ret;
    }

    SocketIp4UDP::SocketIp4UDP()
    : SocketIp4(SOCK_DGRAM), binded_(false), need_bind_(true)
    {
    }

    SocketIp4UDP::SocketIp4UDP(std::string addr, int port)
    : SocketIp4(SOCK_DGRAM), binded_(false), need_bind_(true)
    {
        SelfAddr(addr);
        SelfPort(port);
    }

    SocketIp4UDP::~SocketIp4UDP()
    {
    }

    void SocketIp4UDP::send_buffer(const std::string msg)
    {
        sendto(sk_,  msg.c_str(), msg.size(), 0,
            (sockaddr*)&sk_addr_, sizeof(sk_addr_));
    }

    int SocketIp4UDP::setup_server()
    {
        int ret;

        if (!binded_ && need_bind_)
        {
            if (0 > (ret = bind(sk_, (sockaddr*)&self_sk_addr_, sizeof(self_sk_addr_))))
            {
                // TODO: throw exception
                std::cerr << "bind: " << strerror(errno) << '\n';
                return ret;
            }
            binded_ = true;
        }

        return ret;
    }

    int SocketIp4UDP::recv_buffer(int selfloop)
    {
        int ret;
        Buffer buf(max_rx_);
        socklen_t len = sizeof(sk_addr_);

//      fd_set r_fds;
//      struct timeval tv;

        do
        {
//          FD_ZERO(&r_fds);
//          FD_SET(sk_, &r_fds);
//          tv.tv_sec = 3;
//          tv.tv_usec = 10;
//
//          select(sk_+1, &r_fds, 0, 0, &tv);
//
//          if (FD_ISSET(sk_, &r_fds))
```

```
            {
                if ((ret = recvfrom(sk_, buf.data(), buf.len(), 0,
                    (sockaddr*)&sk_addr_, &len)) == 0)
                {
                //    break;
                }
                else if (ret < 0)
                {
                    if (errno == EAGAIN) break;

                    std::cerr << "recvfrom: " << strerror(errno) << '\n';
                    break;
                }
                else
                {
                    addr_ = inet_ntoa(sk_addr_.sin_addr);
                    msg_ = buf.data();
                }
            }
        } while (selfloop);

        return ret;
    }
} // namespace EasySip
```

```
#include <utility>
#include <iostream>
#include <set>

int main()
{
 std::pair<int, int> fo;
 std::set<std::pair<int, int> > fos;

 fo = std::make_pair(100, 33);
 std::pair<int, int> foe = std::make_pair(100, 33);

 std::cout << fo.first << ' ' << fo.second << '\n';
 fos.insert(fo);

 std::set<std::pair<int, int> >::iterator it = fos.find(foe);
 std::cout << (it == fos.end()) << '\n';

 std::cout << fo.first << ' ' << fo.second << '\n';
}
```

```
#include <iostream>
#include "ts-thr-timer.h"

void on_timeup()
{
 std::cout << __PRETTY_FUNCTION__ << '\n';
}

typedef void (*func_cb)();

int main()
{
 int i=1000;
 timer<int, func_cb>(i, &on_timeup);

 while(1);

 return 0;
}
```

```
#include <stdio.h>
#include <sys/types.h>
#include <ifaddrs.h>
#include <netinet/in.h>
#include <string.h>
#include <arpa/inet.h>

int main (int argc, const char * argv[])
{
    struct ifaddrs * ifAddrStruct=NULL;
    struct ifaddrs * ifa=NULL;
    void * tmpAddrPtr=NULL;

    getifaddrs(&ifAddrStruct);

    for (ifa = ifAddrStruct; ifa != NULL; ifa = ifa->ifa_next) {
        if (!ifa->ifa_addr) {
            continue;
        }
        if (ifa->ifa_addr->sa_family == AF_INET) { // check it is IP4
            // is a valid IP4 Address
            tmpAddrPtr=&((struct sockaddr_in *)ifa->ifa_addr)->sin_addr;
            char addressBuffer[INET_ADDRSTRLEN];
            inet_ntop(AF_INET, tmpAddrPtr, addressBuffer, INET_ADDRSTRLEN);
            printf("%s IP Address %s\n", ifa->ifa_name, addressBuffer);
        } else if (ifa->ifa_addr->sa_family == AF_INET6) { // check it is IP6
            // is a valid IP6 Address
            tmpAddrPtr=&((struct sockaddr_in6 *)ifa->ifa_addr)->sin6_addr;
            char addressBuffer[INET6_ADDRSTRLEN];
            inet_ntop(AF_INET6, tmpAddrPtr, addressBuffer, INET6_ADDRSTRLEN);
            printf("%s IP Address %s\n", ifa->ifa_name, addressBuffer);
        }
    }
    if (ifAddrStruct!=NULL) freeifaddrs(ifAddrStruct);
    return 0;
}
```

```
#include <signal.h>
#include <time.h>
#include <string.h>
#include <error.h>
#include <iostream>
#include <sys/time.h>

//union sigval {          /* Data passed with notification */
//           int     sival_int;         /* Integer value */
//           void   *sival_ptr;         /* Pointer value */
//       };
//
//       struct sigevent {
//           int          sigev_notify; /* Notification method */
//           int          sigev_signo;  /* Notification signal */
//           union sigval sigev_value;  /* Data passed with
//                                         notification */
//           void       (*sigev_notify_function) (union sigval);
//                            /* Function used for thread
//                               notification (SIGEV_THREAD) */
//           void        *sigev_notify_attributes;
//                            /* Attributes for notification thread
//                               (SIGEV_THREAD) */
//           pid_t        sigev_notify_thread_id;
//                            /* ID of thread to signal (SIGEV_THREAD_ID) */
//       };

extern int errno;

void sigev_notify_cb(union sigval sv)
{
 std::cout << "sigev_notify_function: " << sv.sival_int << '\n';
}

//           struct timespec {
//               time_t tv_sec;                /* Seconds */
//               long   tv_nsec;               /* Nanoseconds */
//           };
//
//           struct itimerspec {
//               struct timespec it_interval;  /* Timer interval */
//               struct timespec it_value;     /* Initial expiration */
//
//       int timer_settime(timer_t timerid, int flags,
//                         const struct itimerspec *new_value,
//                         struct itimerspec * old_value);
//  int timer_gettime(timer_t timerid, struct itimerspec *curr_value);

//int main()
//{
// int ret = 0;
//
// struct sigevent sige;
// timer_t tid;
//
// sige.sigev_notify_function = sigev_notify_cb;
// sige.sigev_notify = SIGEV_THREAD;//SIGEV_SIGNAL;
// sige.sigev_signo = SIGRTMIN;
//
//    std::cout << "timer_create: " << (ret = timer_create(CLOCK_REALTIME, &sige, &tid)) << '\n';
//
// struct itimerspec itmspec, *itmspec_cur = new itimerspec;
// struct timespec tmspec_intv, tmspec_expir;
//
// tmspec_intv.tv_sec = 3;
// tmspec_intv.tv_nsec = 0;
// tmspec_expir.tv_sec = 3;
// tmspec_expir.tv_nsec = 0;
//
// itmspec.it_interval = tmspec_intv;
// itmspec.it_value = tmspec_expir;
//
// std::cout << "timer_gettime: " << (ret = timer_gettime(tid, itmspec_cur)) << '\n';
// std::cout << "timer_settime: " << (ret = timer_settime(tid, TIMER_ABSTIME, &itmspec, itmspec_cur)) << '\n';
// std::cout << "timer_delte: " << (ret = timer_delete(tid)) << '\n';
//
// delete itmspec_cur;
//
// return ret;
//}
//

//struct itimerval {
//              struct timeval it_interval; /* next value */
//              struct timeval it_value;    /* current value */
//          };
//
//          struct timeval {
//              time_t      tv_sec;         /* seconds */
//              suseconds_t tv_usec;        /* microseconds */
//          };
// bool cb1()
// {
//  std::cout << "cb1 signo received, time's up\n";
//  return false;
// }
//
struct itimerval it_a;

void sigalrm_cb(int signo)
{
 std::cout << signo << " signo received, time's up\n";
 //signal(SIGALRM, SIG_DFL);
 std::cout << "settimer: " << setitimer(ITIMER_REAL, 0, &it_a) << '\n';
}

int main()
{
 signal(SIGALRM, sigalrm_cb);

 struct itimerval it_a;
 struct timeval tm_cur, tm_next;

 tm_cur.tv_sec = 1;
 tm_cur.tv_usec = 0;

 tm_next.tv_sec = 3;
 tm_next.tv_usec = 0;

 it_a.it_interval = tm_next;
 it_a.it_value = tm_cur;

// std::cout << "settimer: " << setitimer(ITIMER_REAL, &it_a, 0) << '\n';
 char c;
 std::cin.get(c);
// while(1);
 std::cout << "timercmp(&tm_cur, tm_next, ==) " << timercmp(&tm_cur, &tm_next, ==) << '\n';

 return 0;
}
```

```
#include "thread.h"

using namespace EasySip;

void* t1_loop(void* arg)
{
 int a = *(int*)arg;

 std::cout << a << '\n';

 return 0;
}

class X
{
 int num;
public:
 X(int n) : num(n) {};

 void* show()
 {
  std::cout << num << '\n';

 // std::cout << "start " << pthread_yield() << '\n';
  return 0;
 }
};


int main()
{
 int arg = 1098;
 X x(132);
 //Thread t1(t1_loop, (void*)&arg);
 Thread t1 = Thread(&X::show, &x);

// t1.join();

 char c;
 std::cin.get(c);

 return 0;
}
```

```
#include "timer.h"
#include <thread>

using namespace EasySip;

void startt(Timer *t)
{
 t->start();
}


int main()
{
 Timer t1(3, 0);
 Timer t2(5, 0);

 std::thread th1(startt, &t1);
 std::thread th2(startt, &t2);

 th1.join();
 th2.join();

 char c;
 std::cin.get(c);

 return 0;
}
```

```
#include <iostream>
#include <memory>

class A
{
public:
 int n_;
 A(int n) : n_(n)
 { std::cout << n_ << " construct\n"; }
 ~A()
 { std::cout << n_ << " destruct\n"; }
};
int main()
{
 std::shared_ptr<A> a;

 a = std::make_shared<A>(3);
 std::cout << a->n_ << ">>>>>>>>>\n";
 a = std::make_shared<A>(100);
 std::cout << a->n_ << ">>>>>>>>>\n";

 return 0;
}
```

```
/*
 * lsof -i
 * netstat -lptu
 * netstat -tulpn
 */
#include <iostream>
#include "socket.h"
#include "Element/uaserver.h"
#include <thread>

using namespace EasySip;

UAServer server;

void rxd()
{
 server.start();
}

void txd()
{
 char c;
 int run = 1;

 while (run)
 {
  std::cout << "input command: ";
  std::cin >> c;

  switch (c)
  {
   case 'i': server.invite_request(); break;
   case 'r': server.register_request(); break;
   case 'b': server.bye_request(); break;
   case 'c': server.cancel_request(); break;
   case 'u': server.update_request(); break;
   case 'f': server.info_request(); break;
   case 'a': server.ack_request(); break;
   case 'm': server.message_request(); break;
   case 's': server.subscribe_request(); break;
   case 'n': server.notify_request(); break;
   case 'e': server.refer_request(); break;
   case 'o': server.options_request(); break;
   case 'k': server.prack_request(); break;
   case 'q': std::cout << "shutdown ...\n"; while(server.run()) server.run(false); run = 0; break;
   default:
   {
    std::cerr << "Unexpected command '" << c << "(" << int(c) << ")\n";
   }
  }
 }
}

int main()
{
 std::thread tx(txd);
 std::thread rx(rxd);

 tx.join();
 rx.join();

 return 0;
}
```

```
#include <thread>
#include <chrono>

//struct f_op
//{
// void operator()() const {
//  std::cout << __PRETTY_FUNCTION__ << '\n';
// }
//};

template<typename Dua, typename Func>
//struct timer//(Dua const & d, Func const & f)
//{
void timer(Dua const & d, Func const & f)
 {
 std::thread([d, f](){
  std::chrono::milliseconds dur(d);
  std::this_thread::sleep_for(dur);
  f();
 }).detach();
 }
//};
```

```
#include "Element/uaclient.h"
#include <iostream>
#include <thread>

using namespace EasySip;

UAClient client;

void rxd()
{
 client.start();
}

void txd()
{
 char c;
 int run = 1;

 while (run)
 {
  std::cout << "input command: ";
  std::cin >> c;

  switch (c)
  {
   case 'i': client.invite_request(); break;
   case 'r': client.register_request(); break;
   case 'b': client.bye_request(); break;
   case 'c': client.cancel_request(); break;
   case 'u': client.update_request(); break;
   case 'f': client.info_request(); break;
   case 'a': client.ack_request(); break;
   case 'm': client.message_request(); break;
   case 's': client.subscribe_request(); break;
   case 'n': client.notify_request(); break;
   case 'e': client.refer_request(); break;
   case 'o': client.options_request(); break;
   case 'k': client.prack_request(); break;
   case 'q':
    PROGRESS_WITH_FEEDBACK("shutdown", client.run(), client.run(false))
    run = 0; break;
   default:
   {
    std::cerr << "Unexpected command '" << c << "(" << int(c) << ")\n";
   }
  }
 }
}

int main()
{
 std::thread tx(txd);
 std::thread rx(rxd);

 tx.join();
 rx.join();

 return 0;
}
```

```
#include "../include/header_field.h"
#include <iostream>

using namespace EasySip;

int main()
{
 std::cout << HFFrom().Field()<< '\n';
 return 0;
}
```

```
#include <iostream>
#include <string.h>
#include <memory>
#include <locale>
#include <unordered_map>
#include <algorithm>

class A
{
 unsigned long value;
public:
 A(A &a)
 : value(a.Value())
 {
  std::cout << __PRETTY_FUNCTION__ << "\n";
 }
 A()
 : value(3)
 {
  std::cout << __PRETTY_FUNCTION__ << "\n";
 }
 A(unsigned int val)
 {
  value = val;
  std::cout << __PRETTY_FUNCTION__ << "\n";
 }
 void show()
 {
  std::cout << __PRETTY_FUNCTION__ << "\n";
 }
 ~A()
 {}

 void Value(unsigned long val)
 {
  value = val;
 }

 unsigned long Value()
 {
  std::cout << __PRETTY_FUNCTION__ << "\n";
  return value;
 }

 unsigned long operator* (unsigned long val)
 {
  std::cout << __PRETTY_FUNCTION__ << "\n";
  return (value*val);
 }

 A operator= (A a)
 {
  std::cout << __PRETTY_FUNCTION__ << "\n";
  A ret(a.Value());
  return ret;
 }

 friend std::ostream& operator<< (std::ostream &o, A a)
 {
  o << __PRETTY_FUNCTION__ << "\n";
  return o;
 }
// void operator() (A a)
// {
//  std::cout << __PRETTY_FUNCTION__ << "\n";
//  A ret(a.Value());
//  return ret;
// }
```

```
};

class B : public A
{
public:
 B(unsigned int val)
 : A(val)
 {
  std::cout << __PRETTY_FUNCTION__ << "\n";
  show();
 }

 ~B()
 {}

 int operator[] (int val)
 {
  return 3310;
 }

 int operator[] (std::string hello)
 {
  return 928;
 }
 int operator[] (A a)
 {
  a.show();
  return 309;
 }
};

 #define STRDQUOTE """'\""""'
 #define STRQUOTE """'\""""
 #define STRBSLASH """'\\"""'



int main()
{
// B b(100);
 B b2(B(100)*3);
// std::cout << b.Value() << '\n';
 std::cout << b2.Value() << '\n';
 std::cout << b2[3] << '\n';
 std::cout << b2["ok"] << '\n';
 A a(333);
 std::cout << b2[a] << '\n';

 std::shared_ptr<A> p;
 p = std::make_shared<A>();
 p->show();

 std::unordered_map<std::string, std::string> buck;

 buck["hello"] = "now";
 std::string hstr("hello");

 std::cout << buck.hash_function()(hstr) << '\n';

 std::vector<int> digits, buf;

 for (int i = 0; i < 10; i++)
 {
  buf.push_back(i);
 }

 digits = buf;
 std::reverse(digits.begin(), digits.end());

 for (auto &i : digits)
  std::cout << i << ';';

 digits.insert(digits.end(), buf.begin(), buf.end());

 for (auto &i : digits)
  std::cout << i << ';';


 return 1;
}
```

```
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include <osip2/osip.h>

#define MESSAGE_MAX_LENGTH 4000
#define MAX_ADDR_STR 128
#define MESSAGE_ENTRY_MAX_LENGTH 256
#define SIP_PORT 5060
#define EXPIRES_TIME_INSECS 3600

#define USER_ID "7929"
#define SIP_PROXY "sip:10.1.8.10"
#define SIP_FROM "sip:7929 at 10.1.7.110"
#define SIP_TO "sip:7929 at 10.1.7.110"
#define SIP_CONTACT "sip:7929 at 10.1.7.110"
#define LOCAL_IP "10.1.7.110"

osip_t *osip;
int sipSock;

int networkInit()
{
 struct sockaddr_in address;
 if((sipSock = socket(PF_INET, SOCK_DGRAM, 0)) < 0){
  perror("networkInit: error opening socket");
  return -1;
 }
 address.sin_family = PF_INET;
 address.sin_addr.s_addr = htonl(INADDR_ANY);
 address.sin_port = htons(SIP_PORT);
 printf("sipSock = %d/n",sipSock);
 if(bind(sipSock,(struct sockaddr *)&address,sizeof(address)) < 0){
  perror("networkInit: error binding socket");
  return -1;
 }

 return 0;
}

int networkMsgSend(int sock,char *msgP,int msgLen,char *host,int port)
{
 struct sockaddr_in address;
 address.sin_family = PF_INET;
 address.sin_addr.s_addr = inet_addr(host);
 address.sin_port = htons(port);
 if(sendto(sock,msgP,msgLen,0,(struct sockaddr *)&address,sizeof(address)) < 0){
  perror("networkMsgSend: sendto error");
  return -1;
 }
 return 0;
}

int networkMsgRecv(int sock,char *msgP,int msgLen,struct sockaddr_in *address)
{
 int addrLen;
 int dataLen;

 dataLen = recvfrom(sock,msgP,msgLen,0,(struct sockaddr *)address,&addrLen);
 if(dataLen < 0){
  perror("networkMsgRecv: recvfrom error");
  return -1;
 }
 return dataLen;
}
```

```
int SendMsg(osip_transaction_t *tr,osip_message_t *sip, char *host,int port, int out_socket)
{
 int len = 0;
 char *msgP;
 int msgLen;
 int i;
 int status;

 printf("SendMsg/n");

 if((i = osip_message_to_str(sip, &msgP, &msgLen)) != 0){
  OSIP_TRACE(osip_trace(__FILE__,__LINE__,OSIP_BUG,NULL,"failed to convert message/n"));
  return -1;
 }
 if(!networkMsgSend(sipSock,msgP,strlen(msgP),host,port))
  OSIP_TRACE(osip_trace(__FILE__,__LINE__,OSIP_INFO1,NULL,"Time: Udp message sent: /n%s/n",msgP));

 return 0;
}


void cb_rcvICTRes(int type, osip_transaction_t *pott,osip_message_t *pomt)
{
 printf("cb_rcvICTRes/n");
}

void cb_rcvNICTRes(int type, osip_transaction_t *pott,osip_message_t *pomt)
{
 printf("cb_rcvNICTRes/n");
}

void cb_rcvreq(int type, osip_transaction_t *pott,osip_message_t *pomt)
{
 printf("cb_rcvreq/n");
}


void setCallbacks(osip_t *osip)
{
 osip_set_cb_send_message(osip,SendMsg);
 osip_set_message_callback(osip,OSIP_ICT_STATUS_1XX_RECEIVED,cb_rcvICTRes);
 osip_set_message_callback(osip,OSIP_NICT_STATUS_1XX_RECEIVED,cb_rcvNICTRes);
 osip_set_message_callback(osip,OSIP_IST_INVITE_RECEIVED,cb_rcvreq);
}

int AddSupportedMethods(osip_message_t *msgPtr)
{
 osip_message_set_allow(msgPtr, "INVITE");
 osip_message_set_allow(msgPtr, "INFO");
 osip_message_set_allow(msgPtr, "ACK");
 osip_message_set_allow(msgPtr, "CANCEL");
 osip_message_set_allow(msgPtr, "BYE");

 return 0;
}

int bSipSend(
       osip_message_t    *msgPtr,
       osip_fsm_type_t   transactionType)
{
 int status;
 osip_transaction_t *transactionPtr;
 osip_event_t       *sipeventPtr;

 if ( (status = osip_transaction_init(&transactionPtr,transactionType,osip,msgPtr)) != 0 ){
  printf("Failed to init transaction %d",status);
  return -1;
 }

 if((sipeventPtr = osip_new_outgoing_sipmessage(msgPtr)) == NULL){
  printf("Can't allocate message");
  osip_message_free(msgPtr);
  return -1;
 }

 sipeventPtr->transactionid =  transactionPtr->transactionid;


 if((status = osip_message_force_update(msgPtr)) != 0){
  printf("Failed force update",status);
  osip_message_free(msgPtr);
  return -1;
 }

 if((status = osip_transaction_add_event(transactionPtr, sipeventPtr)) != 0){
  printf("Can't add event");
  osip_message_free(msgPtr);
  return -1;
 }

 return 0;

}

int bSipRegisterBuild(osip_message_t **regMsgPtrPtr)
{
 static int gSeqNum = 1;
 int status;
 char *callidNumberStr = NULL;
 char *seqNumStr = NULL;
 osip_call_id_t *callidPtr;
 char temp[MESSAGE_ENTRY_MAX_LENGTH];
 char sipPort[MESSAGE_ENTRY_MAX_LENGTH];
 osip_cseq_t *cseqPtr;
 unsigned int number;
 osip_message_t     *regMsgPtr;
 char expires[10];

 if((status = osip_message_init(&regMsgPtr)) != 0){
  OSIP_TRACE(osip_trace(__FILE__,__LINE__,OSIP_BUG,NULL,"Can't init message!/n"));
  return -1;
 }
 osip_message_set_method(regMsgPtr, osip_strdup("REGISTER"));

 osip_uri_init(&(regMsgPtr->req_uri));
 if ( ( status = osip_uri_parse(regMsgPtr->req_uri, SIP_PROXY) ) != 0)
 {
  OSIP_TRACE(osip_trace(__FILE__,__LINE__,OSIP_BUG,NULL,"uri parse failed!/n"));
  osip_message_free(regMsgPtr);
  return -1;
 }
 osip_message_set_version(regMsgPtr, osip_strdup("SIP/2.0"));
 osip_message_set_status_code(regMsgPtr, 0);
 osip_message_set_reason_phrase(regMsgPtr, NULL);

 osip_message_set_to(regMsgPtr, SIP_TO);
 osip_message_set_from(regMsgPtr, SIP_FROM);

 if((status = osip_call_id_init(&callidPtr)) != 0 ){
  OSIP_TRACE(osip_trace(__FILE__,__LINE__,OSIP_BUG,NULL,"call id failed!/n"));
  osip_message_free(regMsgPtr);
  return -1;
 }
 callidNumberStr = (char *)osip_malloc(MAX_ADDR_STR);
 number = osip_build_random_number();
 sprintf(callidNumberStr,"%u",number);
 osip_call_id_set_number(callidPtr, callidNumberStr);

 osip_call_id_set_host(callidPtr, osip_strdup("10.1.1.63"));

 regMsgPtr->call_id = callidPtr;

 if((status = osip_cseq_init(&cseqPtr)) != 0 ){
  OSIP_TRACE(osip_trace(__FILE__,__LINE__,OSIP_BUG,NULL,"seq init failed!/n"));
  osip_message_free(regMsgPtr);
  return -1;
 }
 gSeqNum++;
 seqNumStr = (char *)osip_malloc(MAX_ADDR_STR);
 sprintf(seqNumStr,"%i", gSeqNum);
 osip_cseq_set_number(cseqPtr, seqNumStr);
 osip_cseq_set_method(cseqPtr, osip_strdup("REGISTER"));
 regMsgPtr->cseq = cseqPtr;

 osip_message_set_max_forwards(regMsgPtr, "70");

 sprintf(sipPort, "%i", SIP_PORT);
 sprintf(temp, "SIP/2.0/%s %s;branch=z9hG4bK%u", "UDP",LOCAL_IP,osip_build_random_number() );
 osip_message_set_via(regMsgPtr, temp);

 osip_message_set_contact(regMsgPtr, SIP_CONTACT);
 sprintf(expires, "%i", EXPIRES_TIME_INSECS);
 osip_message_set_expires(regMsgPtr, expires);

 osip_message_set_content_length(regMsgPtr, "0");

 osip_message_set_user_agent(regMsgPtr, "TotalView 1.0");

 AddSupportedMethods(regMsgPtr);
 *regMsgPtrPtr = regMsgPtr;
 return 0;
}

int bSipRegister(void *cookie)
{
 osip_message_t *regMsgPtr;

 if(bSipRegisterBuild(&regMsgPtr) != 0){
  printf("Error building register message!");
  return -1;
 }

 if (bSipSend(regMsgPtr,NICT) != 0){
  printf("Error sending message!");
  return -1;
 }
 return 0;
}

void processSipMsg()
{
 int port;
 char host[256];
 char msg[MESSAGE_MAX_LENGTH];
 int msgLen;
 osip_event_t *sipevent;
 osip_transaction_t *transaction = NULL;
 struct sockaddr_in sa;
 int status;

 if((msgLen = networkMsgRecv(sipSock,msg,MESSAGE_MAX_LENGTH,&sa)) > 0){
  printf("processSipMsg: RECEIVED MSG/n");
  printf("%s/n",msg);
  sipevent = osip_parse(msg,msgLen);
  if((sipevent==NULL)||(sipevent->sip==NULL)){
   printf("Could not parse SIP message/n");
   osip_event_free(sipevent);
   return;
  }
 }
 osip_message_fix_last_via_header(sipevent->sip,(char *)inet_ntoa(sa.sin_addr),ntohs(sa.sin_port));
 if((status = osip_find_transaction_and_add_event(osip,sipevent)) != 0){
  printf("New transaction!/n");
  if(MSG_IS_REQUEST(sipevent->sip)){
   printf("Got New Request/n");;
  }else if(MSG_IS_RESPONSE(sipevent->sip)){
   printf("Bad Message:%s/n",msg);
   osip_event_free(sipevent);
  }else{
   printf("Unsupported message:%s/n",msg);
   osip_event_free(sipevent);
  }
 }
}

int main()
{
 int i,result;
 fd_set readfds;
 struct timeval tv;
 printf("Initializing OSIP/n");
 TRACE_INITIALIZE(END_TRACE_LEVEL,NULL);
 if(networkInit() < 0){
  printf("ERROR Initializing NETWORK/n");
  return -1;
 }
 i=osip_init(&osip);
 if (i!=0)
  return -1;
 printf("Setting Callbacks/n");
 setCallbacks(osip);
 printf("Entering Main loop 1/n");
 OSIP_TRACE(osip_trace(__FILE__,__LINE__,OSIP_BUG,NULL,"Check OSIP_TRACE init/n"));
 bSipRegister("This is Test Cookie");
 while(1){
  FD_ZERO(&readfds);
  FD_SET(sipSock,&readfds);
  tv.tv_sec = 0;
  tv.tv_usec = 100000;
  result = select(FD_SETSIZE,&readfds,0,0,&tv);
  if(result < 0){
   perror("main: select error");
   exit(1);
  }
  if(FD_ISSET(sipSock,&readfds)){
   printf("main: Received SIP message/n");
   processSipMsg();
  }
  osip_ict_execute(osip);
  osip_ist_execute(osip);
  osip_nict_execute(osip);
  osip_nist_execute(osip);
  osip_timers_ict_execute(osip);
  osip_timers_ist_execute(osip);
  osip_timers_nict_execute(osip);
  osip_timers_nist_execute(osip);
 }
 return 0;
}
```

/--

Hi,

I am new to OSIP Stack.
I am writing a small user agent.
I have initialised osip stack and formed a sip message.
I have initialised a transaction, but after this how do I sent this message to the UAS.
Please let me know the API used to send the message to the UAS.

Is there a transport layer in OSIP Stack or not?

Should I create a socket to the Server and send?

```
#include "Element/proxy.h"
#include <iostream>
#include <thread>

using namespace EasySip;

Proxy proxy;

void rxd()
{
 proxy.start();
}

void txd()
{
 char c;
 int run = 1;

 while (run)
 {
  std::cout << "input command: ";
  std::cin >> c;

  switch (c)
  {
   case 'i': proxy.invite_request(); break;
   case 'r': proxy.register_request(); break;
   case 'b': proxy.bye_request(); break;
   case 'c': proxy.cancel_request(); break;
   case 'u': proxy.update_request(); break;
   case 'f': proxy.info_request(); break;
   case 'a': proxy.ack_request(); break;
   case 'm': proxy.message_request(); break;
   case 's': proxy.subscribe_request(); break;
   case 'n': proxy.notify_request(); break;
   case 'e': proxy.refer_request(); break;
   case 'o': proxy.options_request(); break;
   case 'k': proxy.prack_request(); break;
   case 'q':
    PROGRESS_WITH_FEEDBACK("shutdown", proxy.run(), proxy.run(false))
    run = 0; break;
   default:
   {
    std::cerr << "Unexpected command '" << c << "(" << int(c) << ")\n";
   }
  }
 }
}

int main()
{
 std::thread tx(txd);
 std::thread rx(rxd);

 tx.join();
 rx.join();

 return 0;
}
```