

Use of cookies in real-time system development

M. Gleixner, M. Mc Guire

monika@tglx.de
die.frau@hofr.at

Abstract

While typical scientific works have focused on the technical aspects of real-time development and real-time systems this paper will focus on the caloric requirements that profoundly can impact development and notably scientific dissemination. Though the use of cookies and respective protocols in computer science are well documented [1] we will not cover security aspects, notably related to excessive accumulative effects of consuming large amounts of cookies, rather we will focus on there creation, deployment, assessment and finally there consumption and the positive impact on the real-time Linux community we were able to observe.

KeyWords: cookies, cookie life-cycle, assessment

1 Introduction

2 Underlying Principles

Traditionally cookie protocols in computer science have been bidirectional. An in depth analysis of historic records indicates though that the original protocols in fact were unidirectional - bidirectional protocols limited to the error case where cookies were rejected. Unidirectional cookie protocol have a lot of advantages:

- They are lock free
- Simple to implement
- Portable to almost all flavors of real-time developers

To optimize the use of cookies we found that a donation principle is advantageous, notably with respect to the speed of consumption. A donation protocol is typically quite simple and does not necessarily mandate a hand-shake. The only limitation we found was that early donation, that is donation of only partially initialized cookies can lead to negative response of the consumers as well as the provider. Interruption of cookie generation by cookie consumers

also has generally been found to degrade performance of production, thus this should be prevented by utilizing standard locking procedures, locking out consumers at this early stage - for this traditional locking was found to be sufficient.

3 Case study 1

3.1 The RT-Preempt acceleration cookie pool initialization

The RT-Preempt acceleration cookie initialization needs multiple setup functions which are serialized. In theory the initialization could be parallelized to a certain degree but no proof of higher efficiency has been found in the literature. The simplified source code of the initialization procedure is provided below for clarity. Inline functions and defines can be found in appendix A, further technical description in appendix B.

Note that the various error code pathes have been substituted by `panic()` function calls to simplify the code flow. For the unexperienced cookie pool administrator this seems to be the safest option. The recommendation for experienced cookie pool administrators is to patch the initialization functions with

common sense and handle the various error scenarios gracefully, albeit there is no known way to handle the `panic()` in `setup_cookie_incubator()`. As far as we know there is no ongoing research to find a solution for that problem, but it might be possible to add more incubator types. The default case which indi-

cates a non existing incubator will probably remain unresolved for the next decades.

Trivial cookie pool administrator helper functions have been omitted as well and can be retrieved from the standard admin library `lib_cookies-devel`[3].

Listing 1: init code

```
int rt_acc_cookies_init(struct temp_plate *cookie_pool,
                        int cookies_need_bits,
                        int cookies_need_pieces)
{
    struct cookie_mass mass;
    struct base_container *base_container;
    struct content *content;
    struct baselayer *baselayers;
    struct wrap *wraps;
    struct raw_cookie *raw_cookies;
    struct container *container;
    int nr_cookies = 0;

    mass = mass_of_cookies();
    if (check_storage(mass))
        return -EOUTOFSUPPLY;

    setup_cookie_incubator();

    container = find_container(20);

    base_container = init_base_container(mass);
    start_base_timer(BASE_RISING_TIME);

    content = init_content(mass);
    /*
     * Note: init_content() might have taken longer than
     * BASE_RISING_TIME. This is not a failure.
     */
    wait_for_base_timer();

    baselayers = init_baselayers(base_container);
    add_to_baselayers(baselayers, content);

    /*
     * The following functions are conditional and not necessary
     * for the basic variant, but the full flavoured cookie init
     * will call them.
     */
    if (cookies_need_bits)
        add_bits_on_content(baselayers);
    if (cookies_need_pieces)
        add_pieces_on_content(baselayers);

    /*
     * The next step is mandatory to avoid the cookie rejection

```

```

    * due to incompatibility with the consumers intrusion detection
    * system.
    */
    add_cookie_authorization(baselayers);

    /*
     * Convert the baselayers
     */
    wraps = wrap_baselayers(baselayers);

    /*
     * Init the container to ensure that the cookies can be
     * retrieved later
     */
    init_container(container);

    /*
     * Convert the wrapped base transport layers to raw cookies and
     * store them in the container.
     *
     * Note, that the advanced algorithm is to call the combined
     * split and save function split_wrap_and_save() which takes
     * wrap, nr_cookies and container as arguments Omitted here as
     * it requires well coordinated handling of the various
     * resources.
     */
    raw_cookies = split_wraps(wraps, 20);

    save_raw_cookies(raw_cookies, container);

    /*
     * Phew. The hard work is done!
     */
    add_to_incubator(incubator, container);

    /*
     * The conversion from raw_cookies to final cookies takes some
     * time. Set a timer which alerts us to avoid conversion to
     * unconsumable cookies.
     */
    start_incubator_timer(INCUBATOR_TIME);

    /*
     * Premature removal from incubator must be avoided
     */
    wait_for_incubator_timer();

    remove_from_incubator(incubator, container);

    nr_cookies = transfer_to_pool(container, cookie_pool, 20);
    return nr_cookies;
}

static void setup_cookie_incubator(void)
{

```

```

switch(incubator_type) {
case TYPE_C:
    init_c_incubator(160);
    break;
case TYPE_E:
    init_e_incubator(190);
    break;
case TYPE_G:
    init_g_incubator(3);
    break;
default:
    panic("No_incubator_found");
}
}

static struct container *find_container(int nr_cookies)
{
    struct container *available, *cur;

    available = get_available_containers();

    for_each_container(cur, available) {
        if (match_container(cur->capacity, nr_cookies))
            return cur;
    }
    panic("Hardware_error!");
}

static struct base *init_base(ssize_t mass)
{
    struct baseresource *res;

    res = alloc_base_resource(mass);
    if (!res)
        panic("check_storage_is_buggy!");
    return do_init_base(res);
}

static struct content *init_content(ssize_t mass)
{
    struct contentresource *res;

    res = alloc_content_resource(mass);
    if (!res)
        panic("check_storage_is_buggy!");
    strip_content_res(res);
    remove_core_from_content_res(res);
    return slice_content_res(res);
}

static struct baselayer *init_baselayers(struct base_container *bc)
{
    /*
     * Flattening the base needs to be done carefully to avoid
     * root holes in the cookie transport base layer.

```

```

        */
        return flatten_base(bc);
}

static void add_to_baselayers(struct baselayer *bl,
                             struct content *c)
{
    /*
     * Even distribution is important otherwise consumers might
     * reject the consumption due to missing content. Minor
     * unevenness is acceptable and provides the individual touch
     * of the final cookies.
     */
    distribute_content_evenly(bl, c);
}

static struct wrap *wrap_baselayers(struct baselayer *bl)
{
    /*
     * Wrap up the baselayers so content, bits, pieces and authcode
     * are safely encapsulated by the base transport layer
     */
    return do_wrap_baselayers(bl);
}

static void init_container(struct container *c)
{
    /*
     * cookies require the container to be formatted with FAT.
     */
    format_fat(container);
}

static struct raw_cookies *split_wraps(struct wrap *wraps,
                                       int nr_cookies)
{
    int length;

    length = wrap_length(wraps[0]) + wrap_length(wraps[1]);
    length /= nr_cookies;

    if (length < MIN_COOKIELENGTH)
        panic("Math_error!");

    if (length > MAX_COOKIELENGTH)
        panic("Math_error!");

    return do_split_wraps(wraps, length);
}

static void save_raw_cookies(struct raw_cookie *rc,
                             struct container *co)
{
    struct raw_cookie *cur;

```

```

        for_each_raw_cookie_safe(cur, rc) {
            list_del(&cur->list);
            store_in_container(co, cur);
            if (container_full(co))
                panic("Math_error!");
        }
    }

static int transfer_to_pool(struct container *co,
                           struct temp_plate *cookie_pool,
                           int nr_cookies)
{
    struct temp_plate temp_plate;
    struct cookie *cookies, *cur;
    int cnt = 0;

    cookies = dump_to_temp_plate(co, temp_plate);

    for_each_cookie_safe(cookies, cur) {
        del_from_temp_plate(cur);
        save_to_pool(cur, cookie_pool);
        cnt++;
    }

    if (cnt < nr_cookies)
        alert("Cookie_underrun_detected!");
    if (cnt > nr_cookies)
        alert("Cookie_overcommit_detected");

    return cnt;
}

```

3.2 The RT-Preempt acceleration cookie pool distribution

The distribution of these cookies is initiated by the allocation of the cookie pool template on a free accessible storage space and the cookie protocol server is initialized. The advanced server technology intercepts the HTCPCP protocol described in RFC2324[1].

When the developer initiates the GET method of HTCPCP he receives a notification that the Accept-Additions header field has been expanded and needs to be filled in on the request. The default selection for the cookie addition is set to "accept".

Most of the developers tend to confirm the default and therefor receive the cookie as an addition to the content requested over HTCPCP. If the developer who received the first cookie of the fresh cookie pool happens to be in a room with other developers then the advanced HTCPCP server experiences massive concurrent requests short time after the first delivery.

3.3 The RT-Preempt acceleration cookie pool underrun

The time between the start of the advanced HTCPCP cookie server and the pool underrun depends mostly on the number of developers who are able to connect to the server.

A single developer is mostly unable to empty the pool in one series of HTCPCP requests even if he tries hard, but while the cookies should be consumed rather fast due to the instability of the content it is possible to distribute the consumption across several sessions.

A developer team usually empties the pool pretty fast which is partially driven by the fact that each team member fears that it might miss the opportunity to consume enough cookies[6].

In the rare case that the number of developer is larger than the number of cookies per pool allocation the advanced HTCPCP server needs to be instructed by the pool administrator to split the cookies before serving until the first round of requests has been satisfied[7].

3.4 The RT-Preempt acceleration cookies side effects

The consumption of the cookies has shown very positive side effects. The rich content of the cookies requires the full attention of the developers[5] so there is no ability anymore for the usual chit-chat and other counterproductive office habits including phone calls.

Negative side effects are almost unknown except some unexpected prints[4] on the screen which can be traced back to the FAT formatting of the cookie container.

4 Case Study 2 - 1 byte cookies

While much of the core routines of Case study 2 are comparable to those of case study 1 we will focus on design issues and only briefly outline the main differences to the RT-Preempt related example in the previous section.

Case study 2 will focus on nano-kernel compliant solutions. At the lowest level the design parameters are:

- width of cookies - typically 4 bytes are used
- Allignment - one or two dimensional array
- selection strategy - Compare and swap (CAS or CAS2)
- minimizing overall complexity

4.1 Width of cookies

While no significant overhead for decoding 8 or 16 byte cookies was detected, throughput seems to be optimum at 4 byte cookies, we explain this with the fact that resorting to smaller types increases the overhead of access while larger types incur a penalty due to resource contention on the side of the consumer (i.e. increased efforts in garbage collection).

4.2 Aligment

Alignment primarily concerns with the allignment of containers which can impact the performance of key functions like `save_raw_cookies` and the utilization of the incubator. Obviously multibyte cookies require more thought on alignment, notably as unaligned cookies tend to cause exceptions that can infuriate producers. while one dimensional arrays are simple to use (even RT developers can handle this task) they degrade the utilization of incubators and thus two dimensional arrays with proper alligment should be at least considered.

4.3 Code discussion

These nano cookies have quite obvious performance advantages notably allowing unnoticed transition to multicookie protocols without raising exceptions in other consumers.

Listing 2: init code

```
int nano_cookies_init(int nr_cookies)
{
    struct cookie_mass mass;
    struct base *base;
    struct raw_cookie *raw_cookies;
    struct container *container;

    mass = mass_of_cookies(nr_cookies);
    if (check_storage(mass))
        return -ENOENT; /* strict POSIX! */

    setup_cookie_incubator();

    container = find_container(nr_cookies);

    base = init_base(mass);
    store_base(base);
    start_base_timer(base, BASE_REST_TIME);
    wait_for_base_timer(base);

    init_container(container);
}
```

```

raw_cookies = split_fold(base, nr_cookies);

save_raw_cookies(raw_cookies, container);

add_to_incubator(incubator, container);

/*
 * note that timing is critical due to * these being 1 byte
 * cookies.
 */
start_incubator_hrtimer(INCUB_TIME);

/* non-preemptible section */
spin_on_incubator_timer();

remove_from_incubator(incubator, container);

nr_cookies = transfer_to_pool(container, nr_cookies);
return nr_cookies;
}

```

Functions not described were reused from CS1 so they are not relisted here. Essentially the main difference can be seen in the `init_base()` function which is optimized for the case that resources are available, which is plainly assumed. A minor improvement is to introduce resource conservation by dumping the stack if limits are exceeded and thus operating loss-free by rebasing. Finally fine-tuning of access patterns, i.e. by prefetching have been introduced.

Listing 3: sys cookies core

```

static struct base *init_base(ssize_t mass)
{
    struct baseresource *res = NULL;

    /*
     * don't do this if you are the
     * only one at home !
     */
    while(!res)
        res = alloc_base_resource(mass);

    return do_init_base(res);
}

static struct raw_cookies *split_fold(int nr_cookies)
{
    int length;
    batch_t batch;

    batch = ACCESS_ONCE(base);
    length = base_length(base);
    length /= nr_cookies;

    /*
     * is this possible with 1 byte cookies ?
     */
    if (length < MIN_COOKIELENGTH)

```

```

        panic("Math_error!");

    if (length > MAX_COOKIE_LENGTH) {
        dump_stack();
        rebase(CURRENT | FUTURE);
    }

    return do_split(batch, length);
}

static struct raw_cookies *do_split(struct batch *batch, int length)
{
    int count = 0;
    struct raw_cookie *next, *list;

    while (list) {
        next = list->next;
        prefetch(next);
        csum_fold(next);
        list = next;
        if (++count >= limit(container))
            break;
    }

    if (!count)
        panic("Math_error!");
    else
        raise_batch_soft();
    return list;
}

```

5 Preliminary assessment

A first study of cookie impact was done during the 1st Real Time Linux Workshop in Vienna (TU-Wien 1999) with the general response being positive - in fact in some cases this seems to be the most persistent trace of the event in memory.

To assess the effectiveness of the proposed cookies, preliminary studies were conducted on isolated individuals in Germany and Austria. Positive effects could be found, though it was not clear if these could be generalized. To confirm the findings a mass-study at the 11th Real-Time Linux WorkShop was conducted by exposing a large set of randomly sampled developers to a set of pre-allocated cookies.

5.1 Performance

Unidirectional cookie protocols typically scale $O(N)$ which was found to be sufficient due to the inherent limit of consumers which lead to fairly consistent and strict bounds on N . While multicookie protocols have been observed generally greedy the algorithms were found not to be suitable for cookie protocols,

optimizing for these cases has thus been deferred.

Monitoring of low-water-marks on cookies is essential to achieving a good throughput, unfortunately the cookie production has still a too high latency to allow refill events with small periods, thus pre-allocated pools are the dominant strategy in use.

5.2 Optimization

Surprisingly diversification has shown to be an effective way of maximizing cookie consumption. One observation that could explain this is that we found that multiple concurrent consumers tend to sample different pools, showing a preferred pool depending on specific (local) parameters. Thus the most effective optimization strategy, empirically found, is to initialize sets of diverse cookies, and allow effective parallel access via lock-free mechanisms.

We also found a small number of consumers defaulted to a grab-and-run strategy - while this can't be quantified effectively yet, this strategy does seem to be quite effective.

6 Conclusion

A vast improvement of productivity can be achieved by an effective use of unidirectional cookie protocols for real-time Linux development. Although it can be speculated that such could be deployed in other fields, our current study was limited to real-time developers. Both an increase of productivity as well as biological storage capacity (notably of participating

individuals) could be observed, leading to the conclusion that cookies have a general positive impact on development. Thus we conclude that the use of unidirectional cookie protocols is advantageous and further development of these capabilities is needed.

Though greedy protocols have been discussed in the past, we found that considering these has negative impacts on developers long term and thus are deprecated.

7 Appendix A

Listing 4: header

```
#ifndef OPSYS_METRIC
# error "Operator_lives_in_the_wrong_universe"
#endif

struct cookie_mass {
    grams_t    wheat_flour;
    ml_t       milk;
    grams_t    yeast;
    pieces_t   eggs;
    pieces_t   apples;
    grams_t    sugar;
    some_t     butter;
    optional_t cinnamon;
    optional_t raisins;
};

static inline struct cookie_mass mass_of_cookies(void)
{
    struct cookies_mass mass;

    mass.wheat_flour = 500;
    mass.milk = about(250);
    mass.yeast = 20;
    mass.eggs = 2;
    mass.apples = about(6);
    mass.sugar = about(100);

    return mass;
}

/* Incubator types */
#define TYPE_C "convection_oven"
#define TYPE_E "electric_oven"
#define TYPE_G "gas_oven"

struct baseresource {
    grams_t    wheat_flour;
    ml_t       milk;
    grams_t    yeast;
    pieces_t   eggs;
```

```

};

static inline struct base_container *do_init_base(struct baseresource *r)
{
    struct bowl *small_bowl, *large_bowl;
    struct base_ball *ball;
    struct board *board;

    small_bowl = get_small_bowl();
    put_yeast_into_bowl(small_bowl, r->yeast);
    add_some_lukewarm_water_to_bowl(small_bowl);
    stir_yeast_and_water(small_bowl);
    put_bowl_aside_for_10min(small_bowl);

    if (!twiddle_thumbs())
        do_something_useful();

    large_bowl = get_large_bowl();
    put_wheat_flour_to_bowl(large_bowl, r->wheat_flour);
    add_eggs_to_flour(large_bowl, r->eggs);
    add_yeast_mix_to_flour(large_bowl, small_bowl);
    move_to_dishwasher(small_bowl);

    do {
        if (!r->milk) {
            if (milk_availabe())
                refill(r->milk);
            else
                refill_with_water(r->milk);
        }
        add_some_of_the_milk_to_flour(large_bowl, r->milk);
        knead_ingredients(large_bowl);
    } while(flour_not_fully_absorbed(large_bowl));

    board = get_board();
    base_ball = remove_base_ball_from_bowl(large_bowl);
    put_soft_dough_on_floured_board(board, base_ball);
    knead_until_dough_is_smooth(board, base_ball);
    put_smooth_dough_ball_back(base_ball, large_bowl);
    cover_bowl_with_dish_towel(large_bowl);
    put_bowl_aside_to_warm_place(large_bowl);

    return (struct base_container) large_bowl;
}

#define BASE_RISING_TIME until_base_has_doubled_size

struct contentresource {
    pieces_t    apples;
    grams_t     sugar;
    optional_t  cinnamon;
    optional_t  raisins;
};

static inline void strip_content_res(struct contentresource *r)

```

```

{
    strip(r->apples);
}

static inline void remove_core_from_content_res(struct contentresource *r)
{
    remove_core(r->apples);
}

static inline struct content *slice_content_res(struct contentresource *r)
{
    struct content *slices;

    slices = slice_into_thin_slices(r->apples);

    return slices;
}

static inline struct baselayer *flatten_base(struct base_container *bc)
{
    struct base_ball *dough;
    struct baselayer *doughs;
    int i;

    dough = get_the_dough_from_bowl(bc);
    doughs = split_dough_in_half(dough);

    for (i = 0; i < 2; i++) {
        put_dough_on_floured_board(doughs[i]);
        do {
            apply_rolling_pin_to_dough(doughs[i]);
        } while (length != 40cm && width != 30cm);

        brush_a_bit_liquid_butter_on_flat_dough(doughs[i]);
    }

    move_to_dishwasher(base_container);

    return doughs;
}

static inline void
distribute_content_evenly(struct baselayer *bl, struct content *c)
{
    distribute_apple_slices_on_baselayers(bl, c);
}

static inline void add_bits_on_content(struct baselayer *bl)
{
    add_cinnamon_if_you_like_the_taste(bl);
}

static inline void add_pieces_on_content(struct baselayer *bl)
{
    add_raisins_if_you_like_the_taste(bl);
}

```

```

}

static inline void add_cookie_authorization(struct baselayer *bl)
{
    /*
     * The amount of sugar depends on the sourness of the apples.
     *
     * Note that the dough contains no sugar to have a better
     * contrast between dough and apples.
     */
    distribute_sugar(bl);
}

static inline struct wrap *do_wrap_baselayer(struct baselayer *bl)
{
    int i;

    for (i = 0; i < 2; i++)
        bl[i] = wrap_baselayer_to_long_roll(bl[i]);

    return (struct wraps*) bl;
}

static inline void format_fat(struct container *c)
{
    butter_the_casserole(c);
}

#define MIN_COOKIELENGHT about(4cm)
#define MAX_COOKIELENGHT about(4cm)

static struct raw_cookies *do_split_wraps(struct wrap *wraps, int length)
{
    return cut_the_rolls_into_pieces(wraps, length);
}

static inline void
store_in_container(struct container *co, struct raw_cookie *rc)
{
    roll_the_raw_cookie_in_liquid_butter(rc);
    put_roll_in_casserole_on_flat_side(co, rc);
}

#define INCUBATOR_TIME about(20min)

```

8 Appendix B

Technical description of RT-preempt acceleration cookies

Model name: Apfelschnecken

Alternative model name: Apfelnudeln

Origin: South Germany

9 Appendix C

Listing 5: header

```

/* times in seconds */
#define INCUB_TIME      8*60
#define BASE_REST_TIME  2*60*60

struct baseresource {
    grams_t    wheat_flour;
    grams_t    vanille_sugar;
    grams_t    almonds;
    grams_t    butter;
    pieces_t    eggs;
};

static inline struct cookie_mass mass_of_cookies(int nr_cookies)
{
    int m = nr_cookies / 40;
    struct cookies_mass mass;

    if (!m)
        panic("nr_cookies_to_low!");

    mass.wheat_flour = m * 180;
    mass.butter = m * 100;
    mass.eggs = m * 1;
    mass.vanille_sugar = m * 50;
    mass.almons = m * 50;

    return mass;
}

static inline struct base *do_init_base(struct baseresource *r)
{
    bowl_t bowl;

    bowl = get_large_bowl();
    push(wheat_flour, bowl);
    push(split_egg(EGG_JOKE), bowl);

    /*
     * notably if child processes are involved
     */
    set(IGNORE_MESS);

    do {
        push(grind(mass.almonds), bowl);
        kneed_ingredients(bowl);
    } while(STICKY | MESSY | !SMOOTH);

    return dough_in_large_bowl;
}

```

References

[1] <http://www.encyclopedia.com/doc/1G1-128446233.html>

- [2] <http://www.faqs.org/rfcs/rfc2324.html>
- [3] <http://www.powells.com/biblio?isbn=9780764554209>
- [4] <http://www.funnyforwards.com/fortuneteller1.htm>
- [5] http://www.waltsense.com/storage/articles/20090707_MouthFull.jpg
- [6] http://www.amazing-planet.net/slike/rodents/prairie_dog_food_fight.jpg
- [7] http://www.osthessen-news.de/Media/09/08/News090804_3_WettessenHerren.jpg