List of Schedules:

Collective Schedule: The schedule graph describes the list of p2p and collective operations that need to be posted and completed for a given collective operation. Each operation is represented by a schedule node.

Two types of schedule nodes are supported
- Topology based schedule
- Reactive based schedule

Hierarchical Task: Sequence of operations that
Reactive Task:

```
struct ucc_<component_name>_collective_schedule {
    uint64_t          num_nodes;

}
```

```
struct ucc_<component_name>_task {
        ucc_request_t              *request;
        uint64_t                    incoming_num_edges;
        ucc_schedule_type_t *sch_type; /*p2p, collective */

};
```

```
struct ucc_<component_name>_hierarchical_task {
     ucc_schedule_node super; // Pasha: ucc_schedule_node  - where it defined. Is this struct
ucc_<component_name>_task ?
     tl_team                   *team;
     tl_team                   *fallback_team; // Pasha: what does it mean ?  - the task is being
launched via ucc_tl_collective_task and it may return UCC_UNSUPPORTED. Then we want to
have another team to run this task on. Example: task step is sharp_allreduce (team - is sharp
team), then it may not support all the datatypes and sharp_team will return unsupported status.
We want to have another team (e.g. team ucx) to still move the task.
     ucc_coll_args_t           args; // Pasha: Please expand ucc_coll_args_t  - coll args are
defined in ucc.h
     ucc_schedule_node_t *next; // is
};
```

ucc_status_t ucc_schedule_create_node ();
ucc_status_t ucc_schedule_destroy_node ();

/*Val : the 2 functions below would allow building a hierarchical graph corresponding to a single collective. We also need a way to progress this graph */

*ucc_status_t ucc_*<component_name>*_schedule_create_graph(int n_nodes, ucc_schedule_graph_t **graph, (ucc_status_t progress)(ucc_schedule_graph_t *graph));*

*ucc_status_t ucc_schedule_add_node_to_graph(ucc_schedule_node_t *node,*
*ucc_schedule_graph_t *graph,*
*int position, int n_dependencies, int *dep_ids);*

ucc_status_t ucc_<component_name>_task_progress();

*ucc_status_t ucc_*<component_name>*_context_progress(ucc_context_t *context);*

THIS PART IS IDENTICAL TO UCP_REQUEST

```
typedef struct ucg_request {
    volatile uint32_t        flags;      /**< @ref enum ucg_request_common_flags */
    volatile ucs_status_t    status;     /**< Operation status */
} ucg_request_t;
```

THIS PART IS IDENTICAL TO UCG-SPECIFIC

```
struct ucg_builtin_request {
    ucg_request_t             super;
    ucg_builtin_op_step_t    *step;      /**< indicator of current step within the op */
    ucg_builtin_op_t         *op;        /**< operation currently running */
    ucg_request_t            *comp_req;  /**< completion status is written here */
    volatile uint32_t         pending;   /**< number of step's pending messages */
    ucg_builtin_header_step_t latest;    /**< request iterator, mostly here for
                                              alignment reasons with slot structs */
};

ucs_status_t static UCS_F_ALWAYS_INLINE
ucg_builtin_comp_step_cb(ucg_builtin_request_t *req,
                         ucg_request_t **user_req)
{
    /* Check if this is the last step */
    if (ucs_unlikely(req->step->flags & UCG_BUILTIN_OP_STEP_FLAG_LAST_STEP)) {
        ucs_assert(user_req == NULL); /* not directly from step_execute() */
        ucg_builtin_comp_last_step_cb(req, UCS_OK);
        return UCS_OK;
    }

    /* Mark (per-group) slot as available */
    ucs_container_of(req, ucg_builtin_comp_slot_t, req)->cb = NULL;

    /* Start on the next step for this collective operation */
    ucg_builtin_op_step_t *next_step = ++req->step;
    req->pending = next_step->fragments * next_step->phase->ep_cnt;
    req->latest.step_idx = next_step->am_header.msg.step_idx;

    return ucg_builtin_step_execute(req, user_req);
}


int static UCS_F_ALWAYS_INLINE
ucg_builtin_comp_step_check_cb(ucg_builtin_request_t *req)
{
    UCG_IF_PENDING_REACHED(req, 0, 1) {
        (void) ucg_builtin_comp_step_cb(req, NULL);
        return 1;
    }

    return 0;
}
```

```
typedef struct ucg_builtin_op_step {
    uint16_t                   flags;            /* @ref enum ucg_builtin_op_step_flags */
    uint8_t                    iter_ep;          /* iterator, somewhat volatile */
    uint8_t                    iter_calc;        /* iterator, somewhat volatile */
    ucg_offset_t               iter_offset;      /* iterator, somewhat volatile */
#define UCG_BUILTIN_OFFSET_PIPELINE_READY   ((ucg_offset_t)-1)
#define UCG_BUILTIN_OFFSET_PIPELINE_PENDING ((ucg_offset_t)-2)

    uct_iface_h                uct_iface;
    uct_md_h                   uct_md;
    ucg_builtin_plan_phase_t  *phase;

    int8_t                    *send_buffer;
    int8_t                    *recv_buffer;
    size_t                     buffer_length;
    ucg_builtin_header_t       am_header;
    uint16_t                   batch_cnt;
    uint8_t                    am_id;

    uint32_t                   fragments;        /* != 1 for fragmented operations */
    size_t                     fragment_length;  /* only for fragmented operations */
    /* To enable pipelining of fragmented messages, each fragment has a counter,
     * similar to the request's overall "pending" counter. Once it reaches zero,
     * the fragment can be "forwarded" regardless of the other fragments.
     * This optimization is only valid for "*_WAYPOINT" methods. */
#define UCG_BUILTIN_FRAG_PENDING ((uint8_t)-1)
    volatile uint8_t          *fragment_pending;

    /* Step-level callback functions (as opposed to Op-level callback functions) */
    ucg_builtin_step_calc_cb_t calc_cb;
    ucg_builtin_comp_recv_cb_t recv_cb;

    /* Fields intended for zero-copy */
    struct {
        uct_mem_h              memh;
        ucg_builtin_zcomp_t   *zcomp;
    } zcopy;
} ucg_builtin_op_step_t;

typedef struct ucg_builtin_comp_slot ucg_builtin_comp_slot_t;
struct ucg_builtin_op {
    ucg_op_t                 super;
    unsigned                 opt_cnt; /**< optimization count-down */
    ucg_builtin_op_optm_cb_t optm_cb; /**< optimization function for the operation */
    ucg_builtin_op_init_cb_t init_cb; /**< Initialization function for the operation */
    ucg_builtin_op_fini_cb_t fini_cb; /**< Finalization function for the operation */
    ucg_builtin_comp_slot_t *slots;   /**< slots pointer, for faster initialization */
    ucg_builtin_op_step_t    steps[]; /**< steps required to complete the operation */
};

struct ucc_reactive_task {
      ucc_schedule_node super;
    ucg_builtin_op_step_t    *step; // Pasha can you please expand ucg_builtin_op_step_t . Sounds
like very similar to the above “*next”. // alex: added just above - that’s the bulk of params
used during an individual “step” of a collective operation (for example: tree node has 4 steps
for all reduce, tree root has 2)
```

```
    ucg_builtin_op_t          *op;  // Pasha: can you please expand what is ucg_builtin_op_t.
Similar to the above, you have to carry arguments and type of operations. // alex: added just
above - that’s the bulk of params used during the entire collective operation - basically an
array of steps + some callback functions
};
```